OPTIMIZING THE CONFIGURATION OF SOFTWARE PRODUCT LINE VARIANTS

By

Christopher Jules White

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

December, 2008

Nashville, Tennessee

Approved:

Professor Douglas C. Schmidt

Professor Aniruddha Gokhale

Professor Janos Sztipanovits

Professor Gautam Biswas

Professor Jeff Gray

This dissertation is dedicated to Lisa and Dad for
their continual encouragement, advice, and support.

ACKNOWLEDGMENTS

I would like to express my profound gratitude to Prof. Douglas C. Schmidt for not only the knowledge he has passed on to me but the opportunities he has provided. Of course, Doug's ability to give me the chance to serve as a panel moderator, conference organizer, or invited paper author is a direct result of the exceptional status he has acheived in the field. I can not imagine that it is possible to find an advisor and friend who is more dedicated to ensuring that his graduate students not only grow but flourish.

I first met Prof. Jean Bezivin in Genoa, Italy in 2006. From the first day I met Jean, he has supported my work on the Generic Eclipse Modeling System and provided excellent guidance during our research discussions. Jean is an individual whose generosity to young researchers continues to ensure that new ideas are drawn to the modeling community. I look forward to working with Jean in the years to come and thank him for supporting my research and helping make this dissertation possible.

Since my very first days at Vanderbilt's Institute for Software Integrated Systems, Prof. Aniruddha Gokhale has offered me a welcome combination of research and humor. It is not unusual for Andy to point out a key insight into my research. It is also not unusual for the next sentence to be a hearty chuckle and humorous anecdote about a conference I will be attending. I thank Andy for having a unique blend of research acumen and wit that make research fun.

As a young researcher, it is always critical to get outside advice on research, jobs, and publications. Prof. Jeff Gray has continually been willing to lend a hand in guiding my research and career. There are few people as busy as Jeff who reply immediately to career questions. I sincerely thank Jeff for his advice and support. I hope that we have numerous opportunities to collaborate in the future.

I would like to thank Prof. Janos Sztipanovits and Prof. Gautam Biswas for serving as members of my dissertation committee. The advice and guidance that they have provided

me with on my dissertation has been extremely helpful and will inspire future work. Having personally observed the rigors of their research and travel schedules, I am very appreciative of the time that they have lent me to help improve this dissertation.

My dad and Lisa have provided more support and advice than I could possibly hope to describe in a short acknowledgment. If a way that they might have helped me is not listed here, you can assume they provided support in that manner. Considering the brevity of this paragraph, you can infer how much their love and care has done for me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Software Product-Lines (SPLs) are a technique for creating software applications composed from reusable parts that can be re-targeted for different requirement sets. For example, in the automotive domain, an SPL can be created that allows a car's software to provide Anti-lock Braking System (ABS) capabilities or simply standard braking. Each unique configuration of an SPL is called a *variant*.

SPL variants cannot be constructed arbitrarily, *e.g.*, a car cannot have both ABS and standard braking software controllers. A key step in building an SPL is therefore creating a model of the SPL's variability and the constraints on variant configuration. An effective technique for capturing these configuration constraints is *feature modeling* [82], which documents SPL variability and configuration rules via *features*. Each feature represents an increment in product functionality. A feature model can capture different types of variability, ranging from *SPL variability* (*e.g.*, variations in customer requirements) to *software variability* (*e.g.*, variations in software implementation) [98].

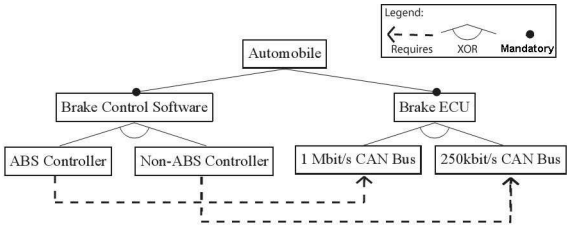


Figure I.1: Simple Feature Model for an Automobile

SPL variants can be specified as a selection or configuration of features. Feature models of SPLs are arranged in a tree-like structure where each successively deeper level in the tree corresponds to a more fine-grained configuration option for the product-line variant, as shown by the feature model in Figure I.1. The parent-child and cross-tree relationships

Figure I.2: Overview of Research Approach

capture the constraints that must be adhered to when selecting a group of features for a variant.

Overview of Research Challenges

Although SPLs help to facilitate software reuse and lower development costs across a number of software development projects, there is a significant amount of complexity associated with the configuration of an SPL variant. The two key challenges that pervade most SPL configuration activities are that:

- SPLs commonly have 1,000s of variable parts or features and an equal number of associated constraints. Fully configuring a valid SPL variant can take several days or months. Moreover, the complexity of configuring a variant makes variant derivation a significant cost and burden for SPL developers.

- Variants need to be optimized for a specific set of requirements. Before SPLs were

used, tightly-coupled and brittle software solutions were developed for each requirement set. The key advantage of this approach was that the software was highly optimized for the requirement set. In order to provide comparable cost and performance, developers need to carefully configure SPL variants to optimize key properties, such as cost.

In order to help improve the speed at which variants can be configured, decrease the complexity of the manual variant derivation steps, and improve the optimality of variants, methods are needed for automating and optimizing variant configuration. Automated variant construction mechanisms have a number of challenges to overcome, such as:

1. The optimization and automation mechanisms must support some level of human interaction. There are certain steps in configuration, such as translating verbal customer specifications into feature selections, that cannot be automated. A key problem is determining how to help optimize the manual modeling steps in the configuration process.

2. Because some portion of most configurations will be entered manually, automation mechanisms must be able to operate on and complete partial solutions. The automation mechanism cannot simply throw away the decisions made during the manual modeling process.

3. Frequently, two partial configurations need to be integrated into a single and complete solution. For example, two developers may make configuration decisions in parallel and an automation mechanism needs to be able to take these two partial specifications and produce a complete configuration that incorporates the two sets of decisions and resolves conflicts.

4. Because configuration is an extremely complex process, any automation mechanism must support some form of automated debugging or diagnosis. Manual configuration

steps may produce incompatible partial configurations, invalid full configurations, or partial configurations for which there is no valid completion. If a configuration cannot be completed, it will be extremely difficult for developers to figure out why and how best to remedy the error. Thus, any automated configuration mechanism must be able to provide automated diagnostics.

Overview of Research Approach

To help address the challenges of configuring good SPL variants, we propose a research approach that uses constraint-based programming techniques to help model engineers create optimal or good SPL variant configurations. The approach we propose, as shown in Figure I.2, uses constraint solving algorithmic techniques as follows:

1. **Model Intelligence** integrates a constraint solver with a modeling tool and uses visual queues to show developers the optimal ways of completing modeling actions. For example, when a modeler initiates a connection between two modeling elements, the constraint solver can solve a Constraint Satisfaction Problem (CSP) [77] representing the semantics of the connection and show the modeler the optimal endpoint by highlighting ideal connection endpoints.

2. **Automated Configuration Integration** techniques that take one or more configurations specified in parallel by a group of developers and derive the values for the intermediate configuration choices that need to be made to properly integrate the set of partial configurations. For example, one developer may select the data access objects to manipulate persistent data in the application, another developer selects the database type that will be used to store persistent data, and the solver derives the appropriate database driver and supporting libraries to allow the data access objects to interact with the database.

3. **Automated Configuration** techniques to take a partially specified configuration and

derive a valid and complete configuration from it. For example, a developer may specify some number of configuration choices that are necessitated by customer requirements and the solver will deduce values for the remaining configuration decisions based on the configuration constraints of the SPL and the developer's optimization goals.

4. **Automated Debugging** techniques which provide developers with recommendations on the optimal set of modifications that can be made to a flawed configuration to make the configuration valid. For example, an automated debugging mechanism could suggest the minimal number of configuration changes to make in order to bring an invalid configuration to a valid state.

Overview of Research Contributions

Summary of Research Accomplishments:

1. 22 Accepted papers, 5 first author journal papers, 10 conference papers, 1 book chapter, and 6 workshop publications, including one "best paper" award

2. Best Paper Award SPLC 2008 "Automated Diagnosis of Product-line Configuration Errors in Feature Models" was selected as the best of 30 papers at SPLC 2008

My research on product-line configuration spans a number of areas in Software Engineering and Distributed Systems, including Model-driven and Aspect-oriented Software Engineering, Software Product-lines for Distributed and Embedded Systems, Autonomic Distributed Systems, and Component-based Distributed Systems. In each area, I have combined new constraint and heuristic based configuration and optimization techniques to produce novel systems. In each research endeavor, I have been keenly aware of the risk of solving a complex problem at the expense of introducing another system development complexity. Although I have separated modeling into its own research area, I have combined modeling with many of the other research areas to reduce the complexity of applying

my contributions to the analysis, configuration, and optimization of distributed and embedded systems. Table l.1 describes my key research contributions obtained from applying SPL configuration techniques to a number of areas of software engineering and distributed systems:

The remainder of this section describes my contributions for each research area in more detail.

Constraint-based Modeling Guidance

Many current software systems are so large and complex that manually producing a correct model of the system is extremely hard. For example, building a model of how software components in an automobile are deployed to hardware components requires adhering to a large number of complex non-functional constraints, such as resource limitations and safety properties. Building an automotive software deployment model that satisfies these complex constraint sets is extremely difficult. My research on model-driven engineering focuses on how constraint-solvers and inference engines can be used to help guide developers towards correct modeling solutions and automate the construction of complex models. My primary contributions in this research area are:

Constraint-based modeling guidance [70, 150, 162]: We have developed and prototyped two different techniques that transform model instances into equivalent constraint programming problems and use constraint-solvers to derive valid ways of completing individual modeling actions. For both techniques, we have developed the constraint programming paradigms and the modeling technologies to make the paradigms usable. The translation techniques are performed on-the-fly as developers incrementally build models. developers are shown visual cues indicating the correct ways of completing individual modeling actions, such as highlighting the valid endpoints for a connection that a user is creating. The modeling guidance techniques also allow users to partially construct models

Research Area	Primary Contributions	Publications
Model-driven and Aspect-Oriented Software Engineering	1. Modeling guidance using constraint solvers 2. Constraint-based model weaving	[146], [162], [107], [150], [70], [160][20], [106]
Software Product-lines for Distributed and Embedded Systems	1. Automated constraint-based configuration of product-line variants with resource constraints and optimization 2. Diagnosis of product-line configuration errors	[161], [157], [149], [156], [145], [153]
Autonomic Systems	1. Configuration healing using constraint-solvers 2. Modeling and simulation of autonomic component-based systems	[163], [152], [155], [159], [151]
Component-based Distributed Systems	1. Deployment and configuration modeling and automation	[144], [154], [147], [106]

Table I.1: Summary of Research Accomplishments

and then use a constraint solver to autonomously complete a large number of modeling actions, such as creating a series of connections between model elements, to bring the model to a valid state.

Constraint-based model weaving [107, 146]: Our research approach created a technique, called constraint-based weaving, that maps model weaving to a constraint satisfaction problem (CSP) and uses a constraint-solver to deduce the appropriate weaving strategy. By mapping model weaving to a CSP and leveraging a constraint solver, our technique (1) generates solutions that are correct with respect to the weaving constraints, (2) can incorporate complex global weaving constraints, (3) can provide weaving solutions that are optimal with respect to a weaving cost function, and (4) can eliminate manual effort that would normally be required to specify pointcuts and maintain them as target models change.

Resource-constrained Software-product Line Variant Configuration for Distributed and Embedded Systems

Research focus: Software Product-Lines (SPLs) are software architectures built on a set of reusable components that can be reconfigured for different requirement sets. A key requirement of an SPL is a specification of the variability in the architecture and how the points of variability affect each other. The most common method of documenting this variability is with a Feature Model. A feature model uses a tree-like structure to describe the points of variability in an SPL and the possible values for the variability points. Feature models can contain thousands of features and complex constraints making finding a good or valid configuration hard. We have devised techniques for (1) automatically deriving configurations that maximize a goal function with a constraint solver and (2) diagnosing errors in feature model configurations. These techniques allow SPL developers to significantly reduce the complexity of both finding a good configuration and pinpointing errors in manual configuration decisions. My primary contributions in this research area are:

Constraint-based automated configuration subject to resource constraints [145, 149, 153, 156, 161]: SPLs designed for systems with resource constraints, such as mobile devices, create a unique challenge for automated product variant selection engines since

deriving valid configurations subject to resource constraints is NP-Hard. Previously, automation techniques did not incorporate configuration resource consumption constraints into variant selection and did not address how a SPL could be designed to improve automated variant selection speed. Through our research work, we have developed CSP and knapsack-based configuration techniques whose input is (1) the requirements of SPL construction and (2) the resources available to the configuration and whose output is the optimal SPL configuration that will fit into the resource limits. These techniques provide automatic configuration selection based on configuration rules and resource constraints. These techniques also ensure that the configuration is optimal with regard to a configurable cost function.

CSP-based feature configuration error diagnosis [157]: Configuration of large feature models can involve multiple stages and participants, which makes it hard to avoid conflicts and errors. Our research provided three contributions to debugging feature model configurations: (1) we created a technique for transforming a flawed feature model configuration into a CSP and showed how a constraint solver can derive the minimal set of feature selection changes to fix an invalid configuration, (2) we created methods for using this technique to automatically resolve conflicts between configuration participant decisions, and (3) we conducted experiments that show that our technique scales to models with over 5,000 features, which is well beyond the size used to validate other automated techniques.

Model-based Healing in Distributed Autonomic Systems

Research focus: Developing and maintaining enterprise applications is hard, due in part to their complexity and the impact of human operator error, which have shown to be a significant contributor to distributed system repair and down time. The aim of autonomic computing is to create distributed applications that have the ability to self-manage, self-heal, self-optimize, self-configure, and self-protect, thereby reducing human interaction with the system to minimize down-time from operator error. Although the benefits

of autonomic computing are significant, the pressures of limited development timeframes and inherent/accidental complexities of large-scale software development have discouraged the integration of sophisticated autonomic computing functionality into distributed applications. My primary contributions in this research area are:

Fine-grained component healing [163]: For each potential error state that an application could enter, most existing MDE adaptation techniques require explicitly modeling both the error state and the numerous actions to transition from the error state to a correct state. For large enterprise applications, there are usually a significant number of potential error states and complex nuanced considerations (e.g. availability of other services, database locks held, transaction states, etc.) that make it very difficult to create a model for application healing. Rather than explicitly modeling error states and recovery actions, we developed a technique, called Refresh, thta uses feature models to capture the rules for determining what is or is not a correct configuration/error state. Feature models provide a mechanism for validating and deriving valid configurations without explicitly specifying every recovery path. Our research has shown how 1) feature models can be used to identify errors, 2) a constraint solver can be used to derive a new and valid application configuration, and 3) the applicationŠs component container can be used to safely abort transactions, release locks, and reboot the failed subsystem with the new valid configuration.

Domain-specific modeling techniques for autonomic systems [151, 152, 159]: To reduce the complexity of developing autonomic component-based systems, we developed a modeling language and toolsuite, called J2EEML. J2EEML provides a high-level modeling notation that helps simplify the development of autonomic systems by providing notations and abstractions that are aligned with autonomic computing, QoS, and EJB terminology, rather than low-level features of operating systems, middleware platforms, and third-generation programming languages. Our toolsuite also includes a customized implementation of the QSim algorithm that allows developers to perform model validation of key autonomic design decisions related to continuous system properties.

Automating the Deployment and Configuration of Component-based Distributed and Embedded Systems

Research focus: Distributed real-time and embedded (DRE) systems are increasingly being built using component-based technologies. Component technologies facilitate software reuse across applications by allowing the dynamic assembly of applications at deployment time via configuration scripts. The late-binding properties of component technologies allow application developers to reuse existing software and reduce costs by leveraging commercial-off-the-shelf (COTS) components. Application developers have traditionally used tightly-coupled proprietary solutions to handle the tight requirements and resource restrictions of DRE systems. Composing a component-based application from components that are not specifically designed for the individual application poses a number of challenges. For example, highly specialized components can make assumptions, such as what type of underlying operating system will be used, that reusable components cannot make. These assumptions can help improve performance (e.g., using specialized APIs) at the cost of reusability. Because DRE systems often operate in environments with little resource slack, being unable to make these key assumptions makes it difficult to find a configuration that meets the required timeliness, safety, and other non-functional properties. My research has focused on automated techniques for dynamically configuring and optimizing component-based applications that are subject to resource constraints. My primary contribution in this research area is:

Dynamic constraint-based component configuration and optimization at application launch [144, 154, 155]: At the heart of our research approach to solve the problems associated with deploying and configuration component-based applications is a MDE tool called Fresh. Fresh is designed to reduce the complexity of deriving a correct application configuration and implementing the configuration in configuration scripts. Fresh simplifies and improves the correctness of configuring DRE component-based applications by: (1)

Capturing configuration rules through feature models, which describe application variability in terms of differences in functionality; (2) Translating an application's feature models into a CSP and using a constraint solver to automatically derive a correct application configuration for a requirements set; (3) Facilitating configuration optimization for a requirements set by providing a configurable cost function to the constraint solver to select optimal configurations; and (4) Providing an XML configuration file annotation language that allows it to directly inject configuration decisions into configuration scripts at application launch.

Dissertation Organization

Each chapter describes a single focus area, the unresolved challenges in the area, and our solution or proposed solution to the challenges. The remainder of this dissertation is organized as follows: Chapter II presents a taxonomy of existing research related to optimizing the configuration of SPL variants; Chapter III explores the automated configuration of SPL variants; Chapter IV delves into the integration of two partial EJB configurations; Chapter V investigates the automated configuration of Corba Component Model (CCM) applications; Chapter VI presents a technique for optimizing and automating aspect weaving. Chapter VII evaluates the optimization of manual modeling steps; Chapter VIII presents an approach for automating and optimizing the healing of service configurations using feature models; Chapter IX presents a heuristic technique for solving large scale configuration problems with resource constraints; Chapter X presents an approach to optimizing hardware and software configuration in tandem; and Chapter XI presents our proposed solution to debugging invalid configurations.

CHAPTER II

RELATED WORK ON CONFIGURATION OPTIMIZATION

This chapter categorizes and examines existing research efforts related to optimizing the configuration of SPL variants. The research is divided into categories based on: customizing SPL variants based on hardware and other non-functional concerns; automated configuration integration; automating the weaving of aspects, which are often used to help implement SPLs; healing configurations when SPL components fail; and debugging SPL configuration errors.

Customizing SPL Variants for Non-functional Concerns

A key challenge in SPL variants is determining how to customize a variant based on a set of non-functional requirements. One area where variant customization is particularly difficult is the customization of software for mobile devices, such as a cell phone. This section examines existing research in the area of software customization for mobile devices.

In [93], Mannion et al present a method for specifying SPL compositional requirements using first-order logic. Each feature is modeled as a boolean variable and the selection of a feature is tied to a number of logical implications. The logical implications define the constraints imposed on the SPL when a particular feature is selected. The validity of a variant can be checked by transforming its feature selection into a set of values for these boolean variables and ensuring there are no contradictions. In [94], Mannion et al. enhance this approach to allow valid product variants to be derived using SAT solvers. The key limitation of this approach is that it is geared towards checking the correctness of the SPL composition with respect to the feature model and not the correctness of the composition with respect to other non-functional requirements. Although non-functional requirements can be used to inform the construction of the feature model or augment the

predicate logic as further boolean variables, the approach cannot handle integer properties, such as cost. This limitation makes it difficult to perform optimization, such as cost or memory minimization. Further discussion on these limitations versus a constraint-based approach is available in [22].

A mapping from feature selection to a CSP is provided by Benavides et al. [22]. Many of the research approaches in this dissertation use this same reduction but also extend it with the capability to handle references and resource constraints. Resource constraints are a key requirement type in mobile devices with limited capabilities. Constraint-based configuration approaches, such as Benavides', have exponential worst case time complexity. Benavides et al. do not address how an SPL can be designed to avoid this worst case behavior and ensure that automatic variant configuration is possible.

In [88], Lemlouma et. al, present a framework for adapting and customizing content before delivering it to a mobile device. Their strategy takes into account device preferences and capabilities. The approach of customizing software is somewhat comparable in that each attempts to deliver customized data to a device based on the device's capabilities and preferences. A key limitation of Lemlouma's approach is that it does not handle resource constraints. Resource constraints are a critical factor when selecting software features for a device with extremely limited hardware resources.

Many complex modeling tools are available for describing and solving combinatorial constraint problems, such as those presented in [33, 40, 57, 99, 129]. These modeling tools provide mechanisms for describing domain-constraints, a set of knowledge, and finding solutions to the constraints. These tools, however, do not provide a high-level mechanism to capture non-functional requirements and SPL composition rules geared towards mobile devices. These tools also do not dictate exactly how an SPL is modeled using constraint-based programming. Benavides et al. [22] have laid out one approach to building a constraint-based model of SPL configuration, but as we pointed out earlier, resource constraints and

SPL design decisions to improve solving performance have not been investigated sufficiently.

Integrating Partial SPL Configurations

Pure::variants [23] is a commercial tool that provides feature modeling capabilities. Pure::variants allows developers to specify features and feature constraints, validate feature selections, and to derive required completions of a feature selection. Pure::variants requires developers to manually specify how features from one feature model affect features in another feature model. Pure::variants does not automate the synchronization of feature models, which is an important capability for distributed development. The lack of model synchronization and integration capabilities prevents developers from working in a distributed fashion.

BigLever's *Software Gears* [31] is another commercial feature modeling and software variant management tool. Software Gears supports features similar to Pure::variants including: feature modeling, automated feature selection completion, and configuration injection. Software Gears requires manually developed mappings between features. BigLever suffers from the same drawbacks of Pure::variants in that it does not provide mechanisms for synchronizing and integration feature selections performed in parallel.

Various approaches [101, 120] have been devised to handle the complexity of configuring applications. Other techniques have also been proposed for variant configuration in SPLs based on configuration rules for application components [133]. This related work focuses on how a configuration problem can be formalized as a CSP. My work in this dissertation extends many of these ideas, particularly those that describe a generic model of configuration as a CSP [101]. With my work, however, modeling has been used to make these techniques practical for industrial SPLs. These approaches provide key building blocks of automated product configuration, but do not address the specific challenges related to decentralized SPL configuration.

Modeling Guidance Related Work

There are a plethora of technologies and standards available for building MDD tools. This section explores some of the main frameworks, tools, and specifications that are available to develop model-driven processes for software systems.

Domain-independent modeling languages. On one end of the MDD tool spectrum are Unified Modeling Language (UML) [58] based tools, such as IBM's Rational Rose [115], that focus on building UML and UML-profile [58] based models. When using UML, all models and languages must be specializations of the UML language. UML provides a single generic language to describe all domains. The advantage of the domain-independent approach of UML-based tools is the increased interoperability between modeling platforms that can be obtained by describing models using a single modeling language and the wide acceptance of the language. New languages can be constructed on top of UML by defining profiles, which are language extensions. UML is based on the MOF metamodel specified by the OMG.

Domain-specific modeling languages. On the other end of the MDD tool spectrum are domain-specific modeling language (DSML) [87] tools. In contrast to UML, DSML tools do not necessarily share a common metamodel or language format. This freedom allows DSMLs to have greater expressivity and handle domains (such as warehouse management, automotive design, and product line configuration), that contain concepts (such as spatial attributes) that are not easily expressed and visualized using UML-based tools. The drawback of DSMLs, however, is that choosing a language generally ties a development process not only to a specific way of representing the model but also generally to a specific tool. Although the loss of interoperability can be problematic, transformations can be written to convert between model formats and still achieve tool interoperability. In many cases, the greater expressivity gained by using a DSML can greatly improve the usability of the MDD tool.

Tools for building DSMLs. To build a DSML, a metamodeling language must be used to

define the syntax of the language. A metamodel describes the rules that determine the correctness of a model instance and specifies the types that can be created in the language. The OMG's current standard is the Meta-Object Facility (MOF) [60] language. MOF provides a metamodel language, similar to UML, that can be used to describe other new languages. MOF itself is recursively defined using MOF. MOF is a specification and therefore is not wedded to a particular tool infrastructure or language technology. Many DSMLs can be described using MOF.

Another popular metamodeling language is the Eclipse Modeling Framework's (EMF) [30] Ecore language. Ecore has nearly identical language constructs to MOF but is a concrete implementation rather than a standard specification. Developers can describe DSMLs using Ecore [30] and then leverage EMF to automatically generate Java data structures to implement the DSML. EMF also possesses the capability to generate basic tree-based graphical editing facilities for Eclipse that operate on the Java data structures produced by EMF.

Complex diagram-like visualizations of EMF-based modeling languages can be developed using the Graphical Editor Framework (GEF) for Eclipse [30]. GEF provides the fundamental patterns and abstractions for visualizing and interacting with a model. Editors can be developed using GEF that allow modelers to draw connections to create associations, nest elements to develop containment relationships, and edit element attributes. GEF editors are based on the Model, View, Controller (MVC) pattern [61]. GEF, however, requires complex graphical coding.

The Graphical Modeling Framework (GMF) [6], is higher level framework, built on top of GEF, that simplifies the development of graphical editors. GMF automates the construction of the controller portion of GEF editors and provides a set of reusable view classes.

MVC controllers are developed using GMF by creating complex XML files that map elements and their attributes to views in the model. GMF takes the XML mappings of elements to views and generates controllers that developers can use to synchronize the model and view of the MDD tool automatically.

Even with the powerful development frameworks presented thus far, developing a visual MDD tool requires significant effort. Meta-programmable modeling environments [87] help alleviate this effort by allowing developers to specify the metamodel for a DSML visually. After the visual specification for the language is complete, the meta-programmable modeling environment can automatically generate the appropriate code and configure itself to provide graphical editing capabilities for the modeling language.

Meta-programmable modeling environments also provide complex remoting, model traversal, library, and other capabilities that are hard to develop from scratch. Two examples of these environments are the Generic Modeling Environment (GME) [87], which is a Windows-based meta-programmable MDD tool, and the Generic Eclipse Modeling System (GEMS) [160], a part of the Eclipse Generative Modeling Technologies (GMT) project. The main tradeoff in using meta-programmable modeling environments is that they tend to provide less flexibility in the visualization of the model.

Many modeling techniques rely on a constraint specification language to provide correctness checking rules that are hard to concisely describe using a graphical language. Certain types of constraints that specify conditions over multiple types of modeling elements, not necessarily related through an interface or inheritance, are more naturally expressed using a textual constraint specification language. The constraint language rules are run against instances of the UML, EMF, or other models to ensure that domain constraints are met. Constraint failures are returned to the modeler through the use of popup windows or other visual mechanisms.

The OMG Object Constraint Language (OCL) [140] is a standard constraint specification for modeling technologies. OCL allows developers to specify invariants, pre-conditions, and post-conditions on types in the modeling language. For example, the OCL constraint:

```
context ECU
inv: self.hostedComponents->collect(x
          | x.requiredRAM)->sum() < self.RAM
```

can be used to check that the sum of the RAM demands of the components hosted by an electronic control unit (ECU) do not exceed the available RAM on the ECU. The first line of the OCL rule defines the context or the type to which the OCL rule should be applied. The second part of the rule, beginning with "inv," defines the invariant condition for the rule. When there is a change to a property of a modeling element of the context type, the invariant conditions for the rules applicable to the element must be checked. Invariants that do not hold after the modification are flagged as errors in the MDD tool.

OCL works well for localized constraints that check the correctness of the properties of a single modeling element. As described earlier, however, the rule can only be used to check the correctness of the state of a modeling element and not to derive valid states for a modeling element, which is a process called backward chaining. In a modeling context, backward chaining is a process whereby the MDD tool deduces correct modeling actions based on the domain constraints. For example, if it were possible to use the above OCL rule to backward chain, a MDD tool could not only determine whether or not an ECU was in a correct state but also, given the current state of an ECU, produce a list of components that could be hosted by the ECU without violating the rule.

For software systems with global constraints and large models, the inability of traditional modeling and constraint checking approaches, such as OCL, to not only flag errors but deduce solutions limits the utility of model-based development approaches. Backward

chaining (providing modeling guidance) becomes more important as domains become more complex, and where it is thus harder to handcraft solutions.

Deriving Solutions that meet a global constraint. The increasing proliferation of DRE systems is leading to the discovery of further hard modeling problems. These domains all tend to exhibit problems, such as scheduling with resource constraints, that are exponential in complexity since they are different types of NP problems. A key challenge in developing effective and scalable DSMLs and models for these domains is deriving the overall organization and architecture of MDD tools and software platforms that can simultaneously meet stringent resource, timing, or cost constraints.

Mobile devices are a domain that have become widely popular and typically exhibit tight resource constraints that must be considered when designing software. Software design decisions, such as the CPU demand of the application, often have physical impacts on the device as well. For example, the scheduling of and workload placed on the CPU can affect the power consumed by the device. Poor scheduling or resource allocation decisions can therefore limit battery life.

Determining the appropriate scheduling policies and application design decisions to handle the resource constraints of mobile devices is critical. Without the proper decisions, devices can have limited battery life and usability. Scheduling with resource constraints, however, is an NP problem and thus cannot be solved manually for non-trivial problems.

Adhering to non-functional requirements. Another challenge of DRE systems is that they often exhibit numerous types of non-functional QoS requirements that are hard to handle manually. For example, in automotive development, an application may have communication timing constraints on the real-time components (e.g., anti-lock braking control), resource constraints on components (e.g., infotainment systems), and feature requirements (e.g., parking assistance) [141]. In environments with this range of QoS requirements, a correct design must solve numerous complex problems and solve them in a layered manner so the solutions are compatible.

For example, the placement of two components on particular ECUs may satisfy a timing constraint but cause a resource constraint failure for another component, such as the infotainment system. Not only must modelers be able to solve numerous types of individually challenging problems, therefore, but they must be able to find solutions that meet all of the requirements.

Another area where complex constraints are common is in configuration management, which is key in emerging software development paradigms, such as product-lines [98] and feature modeling [81]. In these domains, applications are built from reusable software components that interact through a common set of interfaces or framework. Applications are assembled using existing software assets for specific requirement sets. For example, in mission critical avionics product lines, such as Boeing Bold Stroke [123], the correct software component to update the heads-up display (HUD) is selected based on the timing, memory, and other requirements of the particular airframe that the software is being deployed to. Configuration-driven domains exhibit the same characteristics of computationally complex constraints that drive overall system organization as other complex domains.

Debugging Related Work

In prior work [131], Trinidad et al. have shown how feature models can be transformed into diagnosis CSPs and used to identify *full mandatory features*, *void features*, and *dead feature models* [131]. Developers can use this diagnostic capability to identify feature models that do not accurately describe their products and to understand why not. The techniques described in this dissertation build on this idea of using a CSP for automated diagnosis. Whereas Trinidad focuses on diagnosing feature models that do not describe their products, we build an alternate diagnosis model to identify conflicts in feature configurations. Moreover, we provide specific recommendations as to the minimal set of features that can be selected or deselected to eliminate the error.

Batory et al. [17] also investigated debugging techniques for feature models. Their techniques focus on translating feature models into propositional logic and using SAT solvers to automate configuration and verify correctness of configurations. In general, their work touches on debugging feature models rather than individual configurations. The approach in this dissertation focuses on another dimension of debugging, the ability to pinpoint errors in individual configurations and to specify the minimal set of feature selections and deselections to remove the error. Furthermore, propositional logic-based approaches do not typically provide maximization or minimization as primitive functions provided by the solver. Since, the work in this dissertation uses a CSP-based approach, minimization/maximization diagnosis functionality is built-in.

Mannion et al. [93] present a method for encoding feature models as propositional formulas using first-order logic. These propositional formulas can then be used to check the correctness of a configuration. Mannion, however, does not touch on how incorrect configurations are debugged. In contrast, our work in this dissertation provides this capability and can therefore recommend the minimal feature modifications to rectify the problem.

Pure::variants [23], Feature Modeling Plugin (FMP) [46], FeAture Model Analyser (FAMA) [21], and Big Lever Software Gears [31] are tools developed to help developers create correct configurations of SPL feature models. These tools enforce constraints on modelers as the features are selected. None of these tools, however, addresses cases where feature models with incorrect configurations are created and require debugging.

CHAPTER III

AUTOMATED CONFIGURATION

Challenge Overview

This chapter motivates the need for automated configuration mechanisms that choose application configurations on a user's behalf. To illustrate the need for automated configuration mechanisms, the dynamic provisioning of software for mobile phones is used as an example. We show how our automated constraint-based configuration techniques address the gaps in existing automated variant configuration research.

Introduction

The increasing popularity and abundance of mobile and embedded devices is bringing the promise of pervasive computing closer to reality. A recent trend in mobile devices that makes pervasive computing more realistic is the proliferation of services that allow mobile devices to download software on-demand. Mobile phones, for example, can now access web-based applications, such as google mail, or download custom applications from services, such as Verizon's "Get It Now." Google delivers both a web-based interface to google mail and an application that can be downloaded to a mobile phone.

In a pervasive computing environment, the ability to download software on-demand will play a critical role in delivering custom services to users where and when they are needed. For example, when a mobile device enters a retail store, software for browsing back room inventory, displaying store circulars, and purchasing items can be downloaded by the mobile device. When exiting the store, the device may be carried onto a train, in which case applications for placing food orders, checking train schedules, and reserving further tickets could be downloaded by the mobile device.

Software product-lines (SPLs) [38] are a promising approach to help developers manage the complexity of the variability between mobile devices [14, 105, 165]. SPLs [38] enable the development of a group of software packages that can be retargeted for different requirement sets by leveraging common capabilities, patterns, and architectural styles. The design of a SPL is typically guided by scope, commonality, and variability (SCV) analysis [41]. SCV captures key characteristics of software product-lines, including their (1) *scope*, which defines the domains and context of the SPL, (2) *commonalities*, which describe the attributes that recur across all members of the family of products, and (3) *variabilities*, which describe the attributes unique to the different members of the family of products.

Using a SPL, developers can create software architectures that can be rapidly retargeted to the capabilities of different mobile devices. In a pervasive environment, however, the retargeting of a software application to produce a valid variant for a device must happen online. When a device enters a particular context, such as a retail store, the application provider service must very quickly deduce and create a variant for the device. With the large array of device types and rapid development speed of new devices and capabilities, the system will not be able to know about all device types *a priori*. As devices enter a context, their unique capabilities must be discovered and dealt with efficiently and correctly.

Current techniques for automating variant construction from component-based models or feature models, such as those presented in [22, 93, 101, 121, 133], do not sufficiently address various challenges of designing and implementing an automated approach to selecting a product variant for a mobile device. One common capability lacking in each of these approaches is the ability to consider resource consumption constraints, such as the total available memory consumed by the features selected for the variant must be less than 256 kilobytes. Resource constraints are important for mobile devices since resources are typically limited. Some resources, such as cellular network bandwidth, also have a measurable cost associated with them and must be conserved.

Another missing detail of these approaches is the architecture for how a device discovery service would be used to characterize a device's non-functional properties (such as OS, total RAM, etc.) so that a variant can be selected for them. A variant selection engine for mobile devices must have a way to interface with a discovery mechanism. Finally, to provide fast feature selection engines (which aids dynamic software delivery for mobile devices) more research is needed on how SPL design decisions impact the speed of different automation techniques.

To address these gaps in online mobile software variant selection engines, we have developed a tool called *Scatter* that first captures the requirements of a SPL and the resources of a mobile device and then quickly constructs a custom variant from a SPL for the device. this chapter presents the architecture and functionality of Scatter and provides the following contributions to research on custom application deployment in pervasive environments:

- We describe Scatter's graphical requirement and resource specification mechanisms and show how they facilitate the capture and analysis of a wide variety of requirement types.
- We discuss how Scatter transforms requirement specifications into a format that can be operated on by a constraint solver and how we extend existing constraint-based automation approaches [22] to include resource constraints.
- We describe the automated variant selection engine, based on a Constraint Logic Programming Finite Domain (CLP(FD)) solver [77,134] and show how it can rapidly produce both correct and optimal variants based on the requirements.
- We present data from experiments that show how SPL constraints impact variant selection time for a constraint-based variant selection engine.
- We describe SPL design rules that we have gleaned from our experiments that help to improve variant selection time when using a constraint-based approach.

Challenges of Automated Variant Selection for Mobile Devices

The following are three key challenges associated with creating an automated variant selector in a pervasive environment:

• **Unknown device signatures.** Although devices may share common communication protocols and resource description schemas, a variant selection service will not know all device signatures at design time. To provide on-demand variant selection when a new device is encountered, the selection mechanism must be fast. Moreover, devices may possess different signatures. On the one extreme, a laptop may be carried onto a train with a relatively powerful Intel Core Duo processor and a gigabyte or more of RAM. On the other extreme, a Treo mobile phone may be discovered with a 312mhz XScale processor and 64mb of RAM. A variant selector must be able to handle these diverse device descriptions.

• **Variant cost optimization.** Each variant may have a cost associated with it. There may be many valid variants that can be deployed and the variant selector must possess the ability to choose the best variant based on a cost formula. For example, if the variant selected is deployed to a device across a general packet radio service (GPRS) connection that is billed for the total data transferred, it is crucial that this cost/benefit tradeoff be analyzed when determining which variant to deploy. If one variant minimizes the amount of data transferred over thousands or hundreds of thousands of device deployments, it can provide significant cost savings.

• **Limited selection time.** A variant selection may need to occur rapidly. On a train, for instance, a variant selection engine may have tens of minutes or hours before the device exits (although the traveler may become irritated if variant selection takes this long). In a retail store, conversely, if customers cannot get a variant of a sales application quickly, they may become frustrated and leave. To provide a truly seamless pervasive environment, automated variant selection must happen rapidly. When combined with the challenge of not knowing device signatures *a priori* and the need for optimization, achieving quick selection times is even harder.

Capturing PLA and Mobile Device Requirements

Traditional processes of identifying valid PLA variants involve software developers manually determining the software components that must be in a variant, the components to configure, and how to compose and deploy the components. In addition to being infeasible in a pervasive environment (where the target device signatures are not known ahead of time and variant selection must be done on demand), such manual approaches are tedious and error-prone and are a significant source of system downtime [50]. Manual approaches also do not scale well and become impractical with the large solution spaces typical of PLAs.

One way to overcome the speed and correctness deficiencies of manual variant selection is to capture a formal model of the PLA's commonality and variability so that automation can take place. In addition to capturing the composition rules for building variants, a model is needed to analyze the non-functional requirements of a variant to avoid selecting variants that are compositionally correct, but whose functional requirements fail due to being deployed on incompatible or insufficient infrastructure. Figure III.1 shows the cycle of device discovery, variant selection based on requirements, and variant deployment on a train.

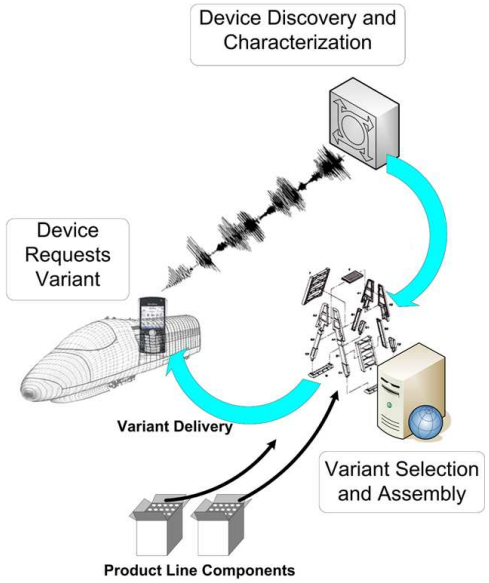


Figure III.1: Selecting a Train Ticket Reservation Service for a Device

For example, a ticket reservation service for a train may require 1 megabyte of memory and 256 kilobits of data transfer over a GPRS connection. If the reservation service is deployed to a device with insufficient free memory, it will not function properly even if it adheres to the PLA compositional rules. To properly configure and select a variant dynamically, therefore, both compositional and non-functional requirements must be considered and matched against the target device.

Capturing and relating composition and non-functional requirements to a mobile device is hard. The remainder of this section describes key challenges of building a compositional and non-functional requirements model of a PLA and outlines how our Scatter tool addresses them.

Solution Approach

The Scatter tool helps automate variant selection for mobile devices by providing:

1. A graphical modeling tool that defines a domain-specific modeling language (DSML) for specifying variant composition rules via a Visio-like interface, as shown in Figure III.2. Scatter allows developers to visually model (1) the components of their PLA, (2) the dependencies and composition rules of components, and (3) the non-functional requirements of each component.

2. A compiler that converts the graphical models from the Scatter modeling tool into both a Prolog knowledge base and a Constraint Satisfaction Problem (CSP) [77, 134] that can be operated on using a Prolog constraint solver. Scatter's formulation of the CSP is an extension of the model presented in [22], which includes resource constraints between components or features.

3. A remoting mechanism that allows a device discovery service to communicate discovered devices to Scatter's variant selection engine. The remoting mechanism allows the discovery service to report back key device non-functional properties, such as OS, memory, and CPU speed.

4. A variant selection engine, based on a Prolog constraint solver, that can automatically select a correct and optimal variant for a device. The Scatter selection engine feeds the device specification, provided by a discovery service, and Prolog knowledge base created by the Scatter compiler, to the constraint solver. The selection engine then translates the results from the constraint solving back into configuration decisions for the variant.

Scatter is implemented using the open-source Generic Eclipse Modeling System (GEMS) [151, 153], which is part of the Eclipse Generative Modeling Technologies (GMT) project. GEMS provides a convenient way to define the metamodel, *i.e.*, the visual syntax of the modeling language. Based on the metamodel, GEMS automatically generates a graphical editor that enforces the grammar specified in the metamodel. Scatter extends our previous work using Role-based Object Constraints (ROCs) and Model Intelligence [106, 148]. Models created in Scatter are transformed via the ROCs infrastructure into formats that can be operated on by a constraint solver.

Scatter Graphical PLA Models

To facilitate the analysis of the variant solution space requires a formal grammar to describe the structure, commonality, and variability (SCV) analysis of the PLA and its valid configurations. This customization grammar can then be used to automatically generate and explore the variant solution space. Scatter provides a visual modeling tool for capturing the SCV of a PLA, as seen in Figure III.2. This view allows developers to formalize which components are available in the PLA, what applications can be constructed, and how each application is composed. The components can be used as an abstraction to describe a

PLA both on system structure [95] or using feature modeling [22, 81]. In our approach, configurations of components or features can be modeled as variabilities using Scatter's SCV model.

To capture a formal definition of the PLA, the components on which it is based must be modeled. The *Component* element is the basic building block in the Scatter DSML that represents an indivisible unit of functionality, such as a Java class or specific feature. For instance, the various food ordering applications are *Components* in our train example.

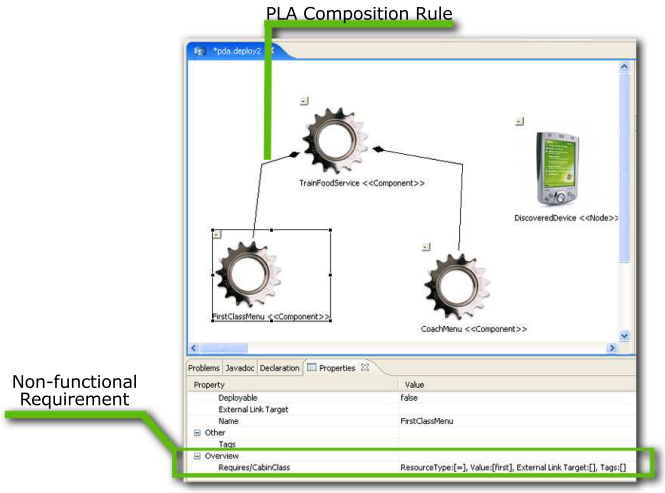


Figure III.2: Scatter PLA Composition and Non-functional Requirements

Dependencies between components can be created by specifying a composition predicate (Required, Exclusive OR, Cardinality, or Exclusion) and the *Components* to which the predicate should be applied. For our train example, the *FoodService* component is connected to the Exclusive OR predicate, which can be connected to the *first class* and *coach class menu* components. This composition indicates that the *FoodService* component can be deployed with exactly one of these menus. The same composition rule could also be specified using the *Cardinalty* predicate by specifying that 1..1 of the *first class* and *coach class menu* components can be deployed with the *FoodService* component.

Component dependencies can be constructed hierarchically from other components with dependencies to capture the compositional variability in a PLA. Components can

also have composition rules with predicates that refer to arbitrary other components in the model. This mechanism is identical to the concept of feature references [49]. To specify the compositional variability in the PLA, developers build *Component* and *Predicate* graphs that show the dependencies and composition rules of the applications and their constituent pieces.

By capturing PLA compositional variability, developers can formally specify how valid variants are composed. With a formal specification of the variant construction rules, Scatter can then automatically explore the variant solution space to discover all valid compositional variants of the PLA for a given device.

Non-functional Requirements Capture

One challenge when building a tool to model a PLA's non-functional requirements is providing a mechanism that not only allows modelers to express a wide variety of constraint types, but also captures them in a form that can be operated on by a constraint solver. At one end of the spectrum are textual specifications, such as "this component should only be deployed to devices located in the first-class cabin running Palm OS." Although these specifications are intuitive to produce and understand, they are imprecise in meaning and require manual translation to the format expected by a constraint solver.

At the other end of the spectrum are the native formats, such as matrices representing systems of linear equations or constraint networks, used by constraint solvers to specify requirements, such as required OS. These native constraint solver formats are easy to operate on with a constraint solver. It is hard, however, to map these formats back to the variant selection for mobile devices, which makes it hard for application developers and quality engineers to use.

Scatter provides a graphical modeling tool to address this challenge and allow developers to express requirements. To specify non-functional requirements, users drag-and-drop requirements from the palette onto components. The child requirement elements of

a component specify the non-functional requirements that must be satisfied by a device's resources. Each requirement has a *Name*, *Type*, and *Value* attribute associated with it:

- The *Name* specifies the name of the resource on the device that it is restricting.
- The *Type* specifies the type of requirement, either '>', '<', '=', '=<', '>=', or '−'.
- The Value indicates the target amount of the resource to which the constraint is being applied.

For example, if a JVM with a version greater than 1.2 is needed, the requirement would have the Name 'JVMVersion', Type '>', and Value '1.2'. For a Resource constraint, such as the amount of memory consumed by a software component, the '−' Type is used, *e.g.*, if a component consumed 200kb of memory, the constraint would be Name 'RAM', Type '−', and Value '200'.

Scatter's approach strikes a careful balance between expressivity and formalness outlined above by blending both the flexibility and intuitiveness of a textual approach with the concrete meaning of a constraint solver format. The Name can be any string and thus modelers can create meaning by providing very descriptive names. The Type provides a clear definition of how the constraint is compared to the resources available on a candidate device. The Type also indicates exactly which constraint solver must be used to analyze the constraint.

All types, except the '-' type, are local constraints governing the placement of one component and are solved by an inferencing engine. These constraints are considered local because their satisfaction is independent of the satisfaction of constraints for other components. For example, if a component requires a specific OS, that constraint does not restrict which other components it can be deployed with. If a component consumes a certain amount of memory, however, its placement on a device will restrict the other components that can be placed with it.

A key challenge in a pervasive environment is that variant selection must take into

account requirements based on business and context data. For example, on a train, the first-class and coach-class cabins may offer different meal services. In coach, travelers may be able to pre-order food via a mobile phone application, but still must physically go and pickup the food. In first-class, however, train staff may be required to deliver food orders to a traveler's seat.

For first class, therefore, a variant that provides a component for notifying the ordering system of where the traveler is sitting may be required while it would not be required in coach. Cabins may also offer different meal selections or meal prices, in which case the variant selection must account for the location-based rules when selecting which menu to deliver with the ordering service. This train variant selection scenario is shown in Figure III.3.

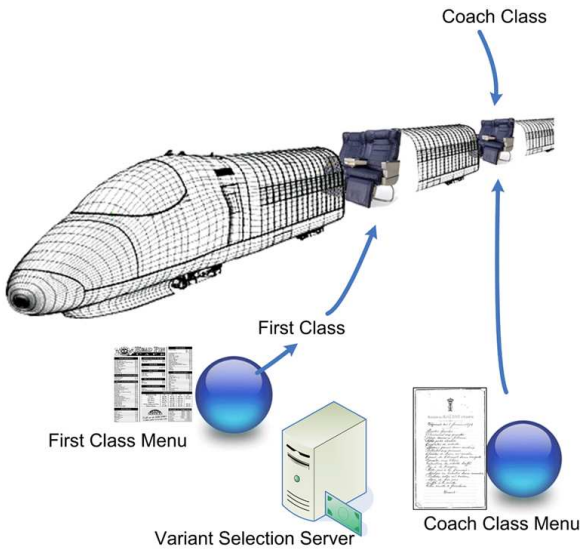


Figure III.3: Cabin Class Constraints for Train Menu Variant Selection

At one extreme, a tool can limit the types of constraints that can be solved to a small subset that is considered most important. At the other extreme, a tool can allow developers to capture any type of constraint, but provide no guarantee of having a way of deducing a variant that satisfies them. Capturing a wide variety of these types of non-functional business and location-based constraints is hard.

Scatter employs a strategy that focuses on allowing the datasources to change while the types of constraints remain constant. This strategy allows it to capture and solve a wide variety of constraint types. For example, a modeler could specify the constraints:

```
JVMVersion > 1.2
WifiCapable = true
CabinClass = first
CPU - 100
RAM - 200
DisplayHResolution > 128
DisplayVResolution > 64
```

This specification mixes multiple different types of domain constraints. A segment of a Scatter requirements model showing these constraints is seen in Figure III.4. The *JVMVersion* constraint relates to the software stack on the device, *CPU* and *RAM* are resource consumption constraints, *WifiCapable* and *DisplayXResolution* are hardware capability constraints, and *CabinClass* is a business/location based constraint.

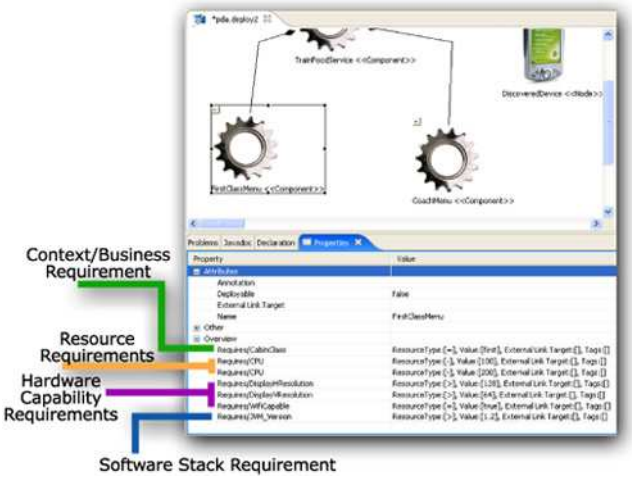


Figure III.4: Capturing Mixed Non-functional Requirement Types in Scatter

The restrictions imposed by the specification format are only on the types of comparisons that can be done and not on the data that the comparison is based upon. This freedom in constraint specification allows Scatter's variant selection to incorporate a large array of datatypes that a device discovery service could provide. This setup allows other services to pre-process the data used by the variant selector and thus allow it to operate on very complex data sets.

For example, context processors based on GPS or RFID can calculate a device's position or type and correlate cabin class. Business-rule engines can calculate customer priorities and provide business analysis. Scatter's architecture thus holds constant the complex portions of variant selection—the constraint solvers—while still allowing the incorporation of new datatypes from a discovery service. For scenarios where other types of constraints are needed, Scatter provides mechanisms for plugging in new types and solvers.

Discovery and Device Signatures

The non-functional properties of a device, such as *JVMVersion* and *CabinClass*, can be used by the variant selection engine to select a variant only if values are provided for them. The values for these variables can be obtained from a mobile device discovery service, as shown in Figure III.5.

Scatter exposes a SOAP-based web service and a CORBA remoting mechanism for remotely communicating device characterizations as they are discovered. The properties of a device are reported back to Scatter as key/value pairs. The keys match the names of the non-functional properties constrained by the non-functional requirements in the Scatter graphical model. These constraints and key/value pairs are used by the variant selection engine to filter the list of variants that can be deployed to a device.

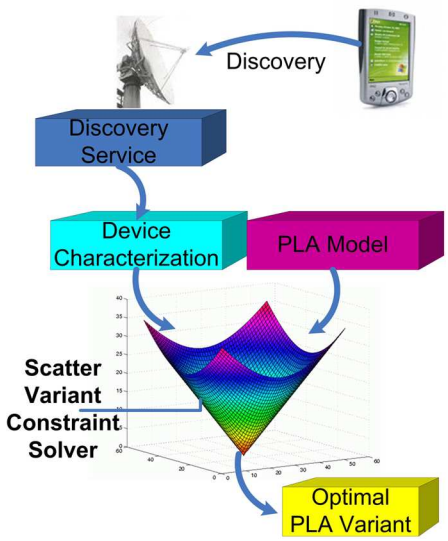


Figure III.5: Scatter Integration with a Discovery Service

Scatter Automated Variant Selector

Scatter provides an automated variant selector that leverages Prolog's inferencing engine and the Java Choco CLP(FD) constraint solver [2]. The Scatter solver uses a layered solving approach to help reduce the combinatorial complexity of satisfying the resource constraints. Scatter prunes the solution space using the SPL composition rules and the local non-functional requirements so that only variants that can run on the target infrastructure are considered. The resource constraints are a form of bin-packing an NP-Hard problem [39]. This layered pruning helps improve selection speed and enables more efficient solving.

Layered Solution Space Pruning

Initially, the variant solution space contains many millions or more possible component compositions. Solving the resource constraints is thus time consuming. To optimize this search, Scatter first prunes the solution space by eliminating components that cannot be deployed to the device because their non-functional requirements, such a JVMVersion or CabinClass, are not met. After pruning away these components, Scatter evaluates the SPL

composition rules to see if any components can no longer be deployed because one of their dependencies has been pruned in the previous step. After pruning the solution space using the SPL composition rules, the resource requirements are considered. After solving the resource constraints, Scatter is left with a drastically reduced number of deployment solutions to select from. At this point, if there is more than one valid variant remaining, Scatter uses a branch and bound algorithm to iteratively try and optimize a developer-supplied cost function by searching the remaining valid solutions.

The first two phases of the solution space pruning use a constraint solver based on standard Prolog inferencing. A rule is specified that only allows a component to be deployed to a device, if for every local non-functional requirement on the component, a resource is present that satisfies the requirement. For example, if a *Component* requires a JVMVersion greater than 1.2, the target *Device* must contain a *Resource* named JVMVersion with a value greater that 1.2 or the component is pruned from the solution space and not considered.

Using CLP(FD) to Solve Resource Constraints

After performing this initial pruning of the solution space, the resource and SPL composition constraints are turned into an input for a CLP(FD) solver. The transformation is an extension of the model proposed in [22] to include resource consumption constraints. The model is also extended to allow for feature references.

A Constraint Satisfaction Problem (CSP) is a problem that involves finding a labeling (a set of values) for a set of variables that adheres to a set of labeling rules (constraints). For example, with the constraint "$X < Y$", $X = 3, Y = 4$ is a correct labeling of the values for X and Y. Typically, the more variables and constraints that are involved in a CSP, the more complex it is to find a correct labeling of the variables.

Selecting a a product variant can be reduced to a CSP. Scatter constructs a set of variables $DC_0 \ldots DC_n$, with domain $[0,1]$, to indicate whether or not the ith component is present in a variant. A variant therefore becomes a binary string where the i^{th} position

represents if the i^{th} component (or feature) is present. Satisfying the CSP for variant selection is devising a labeling of $DC_0 \ldots DC_n$ such that the composition rules of the feature model are adhered to.

Resource consumption constraints are created by ensuring that the sum of the resource demands of a binary string representing a variant do not exceed any resource bound on the device (e.g. $\sum variant\_component\_resource\_demands < device\_resources$). For each *Component* C_i that is deployable in the SPL, a presence variable DC_i, with domain [0,1] is created to indicate whether or not the *Component* is present in the chosen variant. For every resource type in the model, such as CPU, the individual *Component* demands on that resource, $C_i(R)$, when multiplied by their prescence variables and summed cannot exceed the available amount of that resource, $Dvc(R)$, on the *Device*.

If the presence variable is assigned 0, indicating the component is not in the variant, the resource demand contributed by that component to the sum falls to zero. The constraint $\sum C_i(R) * DC_i < Dvc(R)$ is created to enforce this rule. Components that are not selected by the solver, therefore, will have $DC_i = 0$ and will not add to the resource demands of the variant.

The solver supports multiple types of composition relationships between *Components*. For each *Component* C_j that C_i depends on, Scatter creates the constraint: $C_i > 0 \rightarrow C_j = 1$. Scatter also supports a cardinality composition constraint that allows at least *Min* and at most *Max* components from the dependencies to be present. The cardinality operator creates the constraint: $C_i > 0 \rightarrow \sum C_j > Min, \sum C_j < Max$. The standard XOR dependencies from the metamodel are modeled as a special case of cardinality where $Min/Max = 1$. Finally, the solver supports component exclusion. For each *Component* C_n that cannot be present with C_i, the constraint $C_i > 0 \rightarrow C_n = 0$ is created. The variables that can be referred to by the constraints need not be direct children of a component or feature and thus are references.

To support optimization, a variable $Cost(V)$ is defined using the user supplied cost function. For example, $Cost(V) = DC_1 * GPRSC_1 + DC_2 * GPRSC_2 + DC_3 * GPRSC_3 \ldots DC_n * GPRSC_n$ could be used to specify the cost of a variant as the sum of the costs of transferring each component to the target device using a GPRS cellular data connection. This cost function would attempt to minimize the size of the variant deployed within the resource and SPL composition limits. Once the requirements have been translated into CLP(FD) constraints, Scatter asks the CLP solver for a labeling of the variables that maximizes or minimizes the variable $Cost(V)$, which allows the variant selector to choose components that not only adhere to the compositional and resource constraints but that maximize the value of the variant. The user therefore supplies a fitness criteria for selecting the best variant from the population of valid solutions.

Results

A key question is how fast Scatter performs and whether or not online variant selection is possible. To test Scatter's performance, we developed a series of progressively larger SPL models to evaluate solution time. The tests focused solely on the time taken by Scatter to derive a solution and did not involve deploying components. We also tested how various properties of SPL composition and local non-functional constraints affected the solution speed. Our tests were performed on an IBM T43 laptop, with an 1.86ghz Pentium M CPU and 1 gigabyte of memory.

Note that optimization and satisfaction of resource constraints is an NP-Hard problem, where it is always possible to play the role of an adversary and craft a problem instance that provides exponential performance [39]. Constraint satisfaction and optimization algorithms often perform well in practice, however, despite their theoretical worst-case performance. One challenge when developing a SPL that needs to support online variant selection is ensuring that the SPL does not induce worst-case performance of the selector.

We therefore attempted to model realistic SPLs and to test Scatter's performance and better understand the effects of SPL design decisions.

Pure Resource Constraints

We first tested the brute force speed of Scatter when confronting SPLs with no local non-functional or SPL composition requirements that could prune the solution space. We created models with 18, 21, 26, 30, 40, and 50 *Components*. Our models were built incrementally, so each successively larger model contained all of the components from the previous model. In each model, we ensured that not all of the components could be simultaneously supported by the device's resources. Our device was initially allocated 100 units of CPU and 16 megabytes of memory. Scatter's performance results on this model can be seen in Figure III.6.

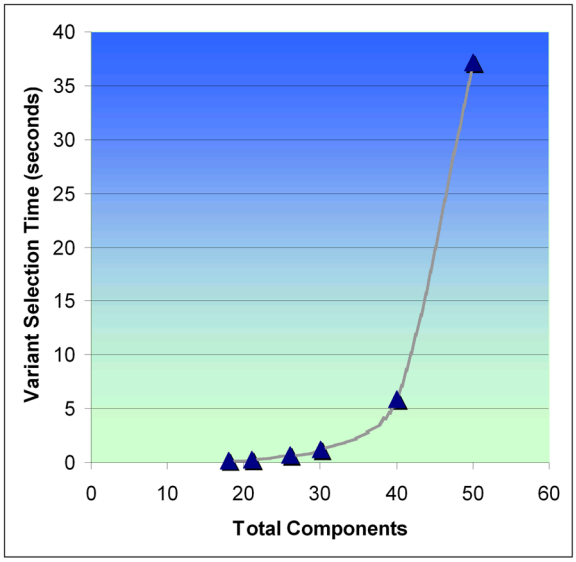


Figure III.6: Scatter Performance on Pure Resource Constraints

As can be seen from the large jump in time from the time to select a variant from 40 to 50 *Components*, solving for a variant does not scale well if resource constraints alone are considered.

Testing the Effect of Limited Resources

We next investigated how the tightness of the resource constraints affected solution time. We incrementally increased the available CPU on the modeled device from 100 to 2,500 units for the 50 Component model. The results can be seen in Figure III.7.

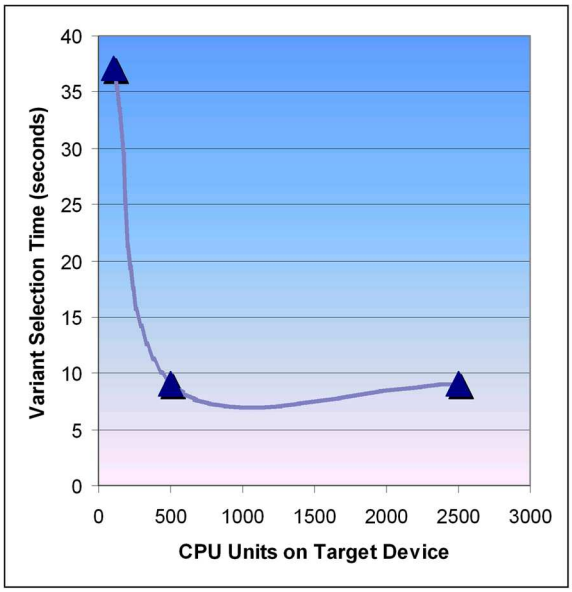


Figure III.7: Scatter Performance as CPU Resources Expand on Device

As shown in Figure III.7, expanding the CPU units from 100 to 500 units dramatically dropped the time required to solve for a variant. Moreover, after increasing the CPU units to 2,500, there was no increase in performance indicating that the tightness of the CPU resource constraints were no longer the limiting bottleneck.

We then proceeded to increase the memory on the device while keeping 2,500 units of CPU. The results are shown in Figure III.8.

Doubling the memory immediately halved the solution time. Doubling the memory again to 128 megabytes provided little benefit since the initial doubling to 64 megabytes made deployment of all of the components possible. As we had hypothesized initially, the solution speed when pure resource constraints are considered is highly dependent on how tight the resource constraints are.

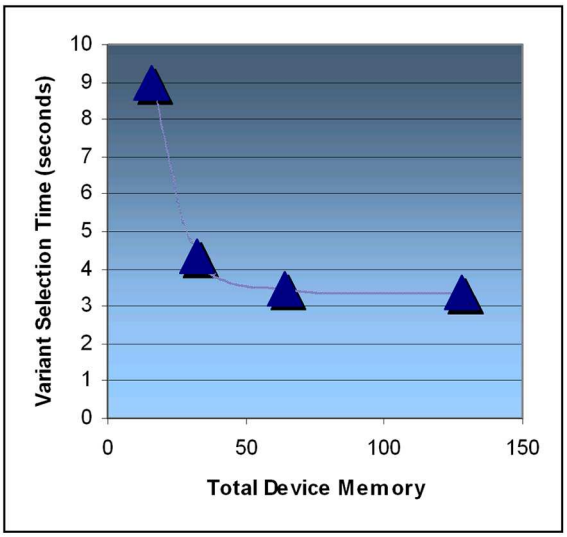


Figure III.8: Scatter Performance as Memory Resources Expand on Device

Testing the Effect of SPL Composition Constraints

Our next set of experiments evaluated how well the dependency constraints within a SPL could filter the solution space and reduce solution time. We modified our models so that the *Components* composed sets of applications that should be deployed together. For example, our *TrainTicketReservationService* was paired with the *TrainScheduleService* and other complementary components.

As with the first experiment III, we used our 50 component model as the initial baseline. We first constructed a tree of dependencies that tied 10 components into an application set that led the root of the tree, the reservation service, to only be deployed if all children were deployed. Each level in the tree depended on the deployment of the layer beneath it. The max depth of the tree was 5. We continued to create new dependencies between the components to produce trees and see the effect. The results are shown in Figure III.9.

As can be seen from the results in Figure III.9, by adding dependencies between components and creating a dependency tree, there was an immediate drop in selection time.

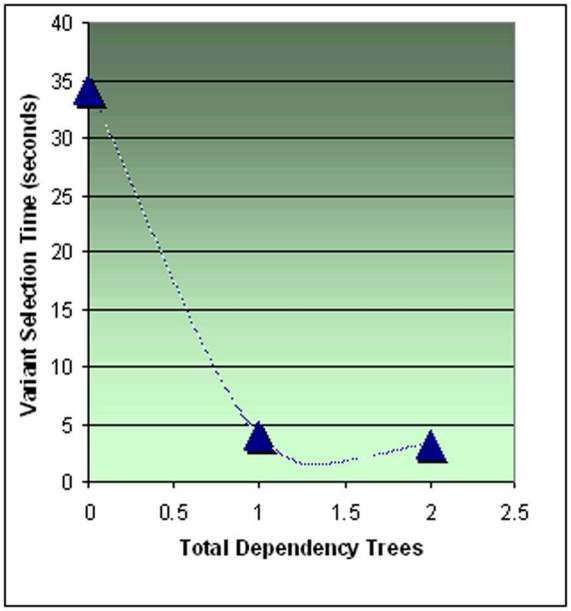


Figure III.9: Scatter Performance as SPL Dependency Trees are Introduced

This is presumably because it reduces the number of possible combinations of the components that must be considered for a variant. Adding more dependencies to the model to add other trees provided only a very small gain over the original large performance increase.

Results Analysis: Mobile SPL Design Strategies

Based on the results we collected from the experiments, we devised a set of mobile SPL design rules to help improve variant selection performance. The remainder of this section presents the lessons we learned from our results.

Exploit non-functional requirements Non-functional requirements can be used to further increase the performance of Scatter. Each component with an unmet non-functional requirement is completely eliminated from consideration. When SPL dependency trees are present, this pruning can have a cascading effect that completely eliminates large numbers of components. One SPL construction rule based on non-functional requirements that was particularly powerful and natural to implement in Scatter exploited the relative lack of variation in packaging of a SPL variant.

Prune using low-granularity requirements The requirements with the lowest granularity filter the largest numbers of variants. For example, when deploying variants, especially from a SPL with high configuration-based variability, such as varying input parameters, the disk footprint of various classes of variants can be used to greatly prune the solution space. If a SPL with 50 components is composed of 5 Java Archive Resource (JAR) files, although there are a large number of ways that the SPL can be composed, there are relatively few valid combinations of the JAR files.

Many variants may also require common sets of these JAR files with various footprints. These variants can be grouped based on their JAR configurations. For each group, a non-functional requirement can be added to the components to ensure that a target Device provide sufficient disk space or communication bandwidth to receive the JARs. For small devices that usually have little availabe disk space, where resource constraints are tighter and solving takes more time, large numbers of Components can be pruned solely due to the lack of packaging variability and need for disk space. This footprint-based strategy works even if there are few functional SPL dependencies between components.

Limit resource tightness Due to the increased cost of finding a variant for small devices where resources are more limited, we developed another design rule. To decrease the difficulty of finding a deployment on small devices, SPL developers should provide local non-functional constraints to immediately filter out unessential resource consumptive *Components* when the resource requirements of the deployable *Components* greatly exceed the available resources on the device. Although the cost function can be used to perform this tradeoff analysis and filter these *Components* during optimization, this method is time consuming. Filtering some components out ahead of time may lead to less optimal solutions but it can greatly improve solution speed. Even by selecting only the least valued components to exclude from consideration, performance can be increased significantly.

Create service classes Another effective mechanism for pruning the solution space with non-functional requirements is to provide various classes of service that divide the components into broad categories. In our train example, for instance, by annotating numerous *Components* with the *CabinClass* and other similar context-based requirements, the solution space can be quickly pruned to only search the correct class of service for the target device. In general, the more non-functional requirements that can be specified, the quicker Scatter can prune away invalid solutions and hone in on the correct configuration. Moreover, each non-functional requirement gives the solver more insight into how Components are meant to be used and thus reduces the likelihood of unanticipated variants that fail.

From our experiments, we have seen that when a SPL for a mobile device is properly specified with good constraints, Scatter can solve models involving 50 or fewer components in seconds. This performance should be more than adequate for many pervasive environments, particularly when device signature and variants are cached to eliminate repetitive solving for known solutions. In future work, we intend to test Scatter with larger models and evaluate more characteristics of SPLs that can be used to reduce variant selection time.

CHAPTER IV

AUTOMATED CONFIGURATION INTEGRATION IN JAVA

Challenge Overview

This chapter illustrates the need for automated configuration integration mechanisms, which are techniques for taking two manually specified partial configurations and deriving any intermediate configuration choices that need to be made to meld the two together. To illustrate the challenges of configuration integration, the chapter utilizes examples from the configuration of enterprise applications, such as enterprise Java applications.

Introduction

Enterprise applications are large-scale software programs, typically hosted on multiple application servers, that perform complex business processes. Enterprise applications commonly support thousands or more simultaneous users and are often written using component middleware, such as Enterprise Java Beans. Due to their large number of components, complicated XML-based configuration files, and complex interdependencies between components, enterprise applications are often hard to configure.

Enterprise application configuration is typically a decentralized process. Multiple development roles edit configuration files, install applications, and perform other configuration steps to deploy an enterprise application. Each role usually operates semi-independently from other roles and focuses on aspects of application configuration pertinent to requirements the role is responsible for. For example, database developers identify the best database vendor, database schema, and database configuration parameters to use; component developers determine what software components are needed to meet the functional requirements for the application; and IT administrators install and configure application servers on the appropriate nodes in data centers.

The diverse configuration decisions made by each role outlined above constrain the possible configuration decisions of other roles. For example, when database developers choose a database, component developers must use the appropriate database driver for that database. These configuration decisions are distributed across roles and configuration files and must ultimately be integrated to create a complete and valid configuration. When integration takes place, each role often performs other configuration steps (such as installing the correct database driver) necessitated by decisions made by other roles. This integration process may require adding new components to adapt the application to its target environment, loading extra libraries into the application server, or other types of configuration steps.

It is hard to keep track of and analyze an enterprise application's configuration decisions (configuration state) since these decisions are enacted by multiple roles, involve hundreds or more components, and are spread throughout numerous configuration files. Even after the configuration state is collected, the complex interdependencies and implications of the configuration decisions must be understood to check the validity of the configuration state and derive further configuration steps to perform. Finally, after a complete configuration for the application is derived, the configuration must be enacted by the multiple roles in numerous configuration files.

Configuration errors related to functional requirements have been shown to be a major contributor to enterprise application downtime and cost. In some studies, for example, misconfiguration from manual processes has been shown to cause over 50% of all application failures [50]. One approach to alleviating the complexity of configuring enterprise applications is to use model-driven development [125]. With a model-based approach, a model of the application's configuration rules and configuration state is first built. Configuration artifacts, such as XML configuration files, are generated from the model. By creating a model of application components and configuration requirements, algorithmic techniques

(such as constraint solvers) can be used to check configuration correctness and derive valid configurations.

Feature modeling [49, 81] is a promising modeling technique for representing the configuration state of enterprise applications. This technique can capture the configuration dependencies between roles and non-functional requirements for enterprise applications. Feature modeling provides a set of modeling formalisms that decompose an application based on functional and non-functional variations and formalize the rules by which these variabilities may be composed into an application variant. In the context of enterprise applications, feature modeling can be used to capture (1) what configuration decisions must be made to install an enterprise application, (2) what roles are responsible for what configuration steps (by having a separate feature model per role), (3) how each role's configuration steps affect other roles, and (4) how the target infrastructure and requirements limit the valid configuration possibilities.

To configure an application with a feature model, development team members (such as component developers, database developers, etc.) first identify a *feature selection*, which is a group of desired functional capabilities that constitute a complete configuration of an application and adhere to the constraints specified in the feature model. These participants must then determine what configuration actions, such as adding component IDs to application XML descriptors or installing a specific database, are required to enable and/or implement the functionality specified in the feature set. What we term *feature selection* is also often called *product configuration* [89]. To avoid confusion, we use the term *application configuration* to denote editing XML files, installing application servers, and other configuration related actions. Likewise, we define *feature selection* as the process of determining a valid set of configuration parameters (*i.e.*, filling in variabilities) with respect to a feature model's constraints.

The challenge with using existing model-based approaches, including feature models, for enterprise application configuration is that they often require a single large monolithic

model of the system [22, 44, 52, 62, 101, 120]. Enterprise configuration decisions are often spread across multiple files, developers, and hosts, however, so it is time consuming to build and maintain accurate feature models. Moreover, the decentralization of enterprise application configuration decisions makes it easy for monolithic models to drift out of sync with the actual configuration state.

Some approaches advocate the use of multiple models [23, 31] that contain references to each other. This multi-model organization better mirrors the decentralized structure of enterprise application configuration and improves developer concurrency. The multi-model approach, however, requires that each role manually specify how changes to other roles' models affect elements in its own model. Manually specifying these effects is thus tedious and error-prone.

This chapter describes how we created and applied an automated application configuration tool called *Fresh* to configure enterprise Java applications. Our Fresh approach uses a novel probe-based synchronization technique to allow each role to use its own feature model, while also not requiring manual cross-model effect specification and synchronization. Each probe is executable Java code that tests a property of the target environment (such as what libraries have been installed) and updates a role's feature model according to the results of the test (such as disabling or enabling a corresponding feature). As each role changes its feature selection and enacts changes on the application or target environment, Fresh probes translate the changes into feature modifications in any affected models. Roles synchronize models by describing how they affect and are affected by code and configuration changes to the application and target environment.

Fresh combines its multi-model approach with a constraint solver to reduce the complexity of enterprise application configuration. The key contribution of this chapter is showing how Fresh simplifies enterprise application configuration by:

1. Automatically collecting the application's distributed configuration state with probes, *e.g.* determine the database installed, etc.

2. Phrasing the completion of the application's feature selection as a constraint satisfaction problem.

3. Deriving any remaining required features by solving the constraint satisfaction problem with a constraint solver, *e.g.*, if a database driver is not installed determine which one is needed.

4. Rewriting the application's configuration files to include any new required features *e.g.*, add the database driver to the application configuration.

Example Enterprise Java Application: Pet Store

As a reference architecture of an enterprise Java application, we use the J2EE Pet Store application [9], which provides an example e-commerce site that allows customers to search for and purchase pets over the Internet. Pet Store was developed originally to showcase the benefits of J2EE technologies. Since its original release, nearly every major J2EE application server has included a refactored version of Pet Store as an example application. Microsoft has also reimplemented Pet Store (called Pet Shop) in .NET to highlight the differences between J2EE and .NET.

Since Pet Store is widely known and demonstrates the features of enterprise Java, we use it in this chapter to show the configuration challenges of enterprise Java applications. To show the application's numerous points of variability we built a feature model of Pet Store bundled with the Java Spring framework [79], which allows developers to create highly-modular and configurable enterprise Java applications. In particular, Spring uses (1) factory patterns [61] to instantiate and interconnect enterprise Java components (beans) and (2) Java reflection to shield application components from details of the configuration process. At launch, a factory is created and initialized using one or more XML configuration files, which determine what components it constructs and how they are wired together. In the process of constructing objects, the factory may associate crosscutting aspect advice with

them, generate dynamic proxies to perform remote invocations, load objects into a naming service, or perform numerous other complex application configuration tasks.

We bounded the scope of the feature model presented in this chapter to a group of features related to the data tier of Pet Store. For example, in the feature model shown in Figure IV.1, the Pet Store can use either a *CombinedDatabase* setup, where both order and product data is stored in the same database, or a *DualDatabase* setup where product and order data are stored in separate databases. Depending on which setup is chosen, the Pet Store's application configuration files must be changed to include the appropriate Data Access Objects (DAOs). If a DualDatabase setup is used, developers alter the Pet Store configuration files to instruct Spring to instantiate and use the *JtaDAOs* and wire them into the application.

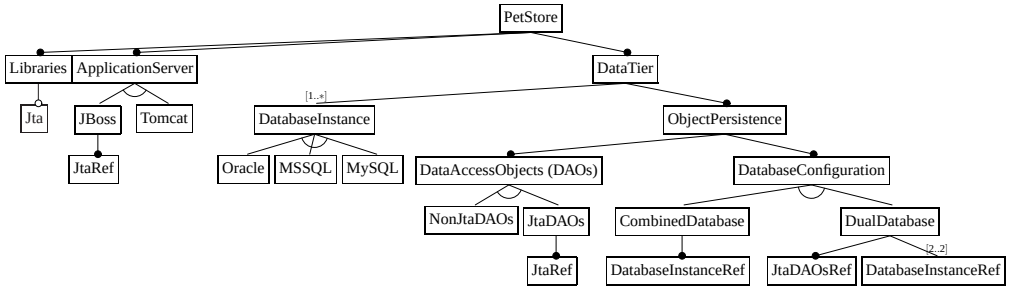


Figure IV.1: Feature Model of the Features Related to the J2EE Pet Store's Data Tier

The Complexity of Enterprise Java Feature Selection

In this section, we explore: (1) the varied participant roles involved in configuring a Spring application and where their decisions are reflected in the application, (2) the complex conflicting requirements and dependencies exposed by the roles, and (3) the difficulty of deriving a feature set that adheres to all of the functional requirements and non-functional requirements created by the roles.

Dimensions of Configuration

By identifying the key roles involved in feature selection, we can illuminate the types of requirements and preferences that will be involved and the points where they are likely to conflict. Furthermore, we can determine where each role implements its decisions so that they can be collected. For the majority of enterprise Java applications, the parties involved in feature selection can be divided into roughly six roles: enterprise bean (component) developer, web developer, client application developer, database developer, application assembler, and IT administrator (application deployer and administrator) [96].

To implement feature selections from a model, these various roles must rely on each other to perform configuration steps to select values for different points of variability in the application. These various configuration steps must be consistent with each other with respect to the feature model constraints. Enterprise Java configuration can be viewed along several broad dimensions:

1. *Feature Configuration*: A feature, component, or user requires a specific feature to be enabled, disabled, etc. The end user may require the component developer to enable email notification of completed customer orders.

2. *Attribute Configuration*: A component or feature requires that the value of an attribute on another component or feature adhere to a specific constraint. For example, the component developer may require that the IT administrator install a Java Virtual Machine with a version number greater than or equal to 1.5.

3. *Local Addressing Configuration*: A component used by one role needs to know the address or unique identifier of another component in the application. For example, the component developer needs to know the bean names (unique identifiers in a Spring XML configuration file) of the DAOs created by another component developer.

4. *Remote Addressing Configuration*: A component used by one role needs to know the

address or unique identifier of a remotely accessible component provided by another role. For example, the Data Access Objects (DAOs) used by the component developer need to know the table names created in the database by the database developer.

5. *Application Configuration*: A component used by one role needs another component in the application instantiated. For example, the DAOs need an instance of the database driver instantiated.

6. *Infrastructure Configuration*: A component used by one role needs another process outside the application installed, configured, and launched or a specific type of hardware setup. A MessageDriven bean (a bean that receives Java Messaging Service (JMS) messages) created by the component developer will require that a specific messaging queue be installed, configured, and started by the IT administrator. The DAOs used by the component developer require that the database developer install certain tables into the database. The component developer's Java Transaction API (JTA) DAOs need the JTA libraries loaded into memory by the IT Admin.

Challenges Produced by Competing Roles and Forces

Enterprise Java applications are prone to a number of common configuration problems. In ideal situations, these errors are easily identified by an application that fails to load into its container. In more serious situations, these errors reflect subtle inconsistencies, such as incorrect file permissions, that may be overlooked and could lead to failures, such as security breaches.

There are four major types of configuration errors produced by the complexity of configuring an enterprise Java product:

Problem 1 - Feature Selection Complexity Functional composition rules are not adhered to when a feature set is selected because the large number of rules and features involved makes it too combinatorially complex to manage manually. A further challenge of the

feature selection process is that the decisions made by one role may spill over into the decisions that need to be made by a second role and it is difficult to both foresee these ripple effects and to enforce them.

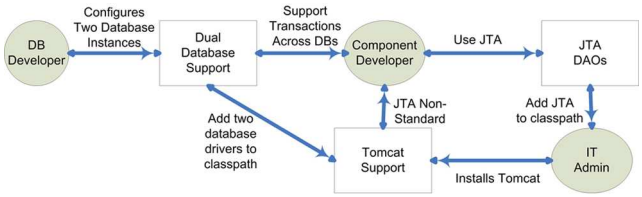


Figure IV.2: Data Tier Feature Selection Forces and Their Effect on Various Roles

The Spring Pet Store, for example, offers the ability to use a single or dual database setup and either plain DAOs or JTA-enabled DAOs. As can be seen in Figure IV.2, if the database developer chooses to use the dual database setup the component developer must support transactions across multiple databases. This decision requires the use of JTA enabled DAOs. A side-effect of enabling the JTA DAOs is that the Pet Store can no longer run in a standard J2EE web container, such as Tomcat [28]. This requirement means that the IT administrator must either use a full-blown J2EE Application Server, such as JBoss [55], or configure the web-container with additional components to support JTA. In this case, a decision made by the database developer ripples through the functional composition decisions that must be made by multiple other roles. The numerous dependencies between roles and features makes the feature selection process complex.

If the constraints are not adhered to across roles, these ripple effects can lead to the selection of an invalid feature set. The more components that are in the application and the more dependencies exist between developers, the harder it is to account for the side effects of a feature selection.

Problem 2 - Incorrect Feature Selection Implementation Feature selections may not be implemented properly. After a feature set is selected, multiple configuration files must be edited and various actions (*e.g.*, starting processes, etc.) taken by the roles to enable the

features. If the IT administrator, for example, does not edit the application server XML configuration files properly to load the correct libraries or does not completely understand the requirements or implications of the feature selection decisions, a non-functional variant can be produced. The non-functional variant may fail to load properly into its container or load correctly but function incorrectly.

As can be seen in Figure IV.3, to enable transaction support across databases with JTA, the component developer must edit the application XML deployment descriptor to link in an XML configuration file containing the JTA enabled DAOs. These DAOs must have a reference to the DB Drivers provided by the database developer. Furthermore, the DB Drivers need the correct port and URIs of the database instances. The IT administrator must not only edit the web.xml descriptor of the application to load the DB Driver libraries into the classpath but must also ensure that the descriptor references the appropriate XML configuration files for the Pet Store. Finally, the IT administrator must install the extra JTA Libraries into Tomcat. If any of these steps are performed improperly or are not consistent with each other, the Pet Store will not function.

This example shows how feature selection involves the coordination of multiple roles in the configuration process. Mistakes due to human error and mis-communication between roles are common in a configuration process. In some studies, misconfiguration from manual processes has been shown to cause over 50% of all application failures [50]. For complex enterprise Java configuration tasks, manual processes are extremely tedious and error-prone.

Problem 3 - Incorrect Information Flows Across Roles Often, roles misunderstand decisions made by another role. The most costly and generally difficult to identify misunderstandings involve environmental properties (*e.g.*, application server vendor, file permissions, etc.). For example, the Pet Store provides both generic DAOs that use only standard SQL mechanisms and DAOs for Oracle and MSSQL that use vendor-specific interfaces. The standard SQL DAOs will load properly into the Pet Store without errors regardless

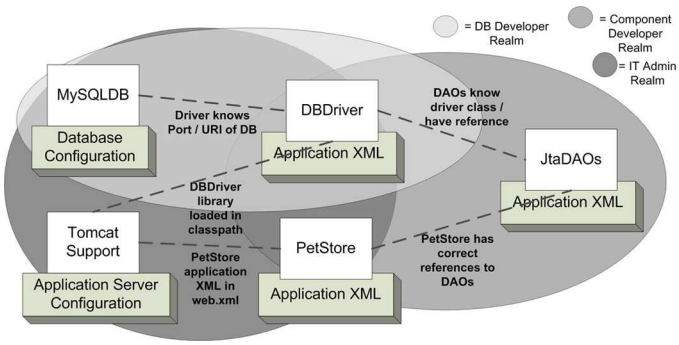


Figure IV.3: Configuration Dependencies between Features and Roles for Data Tier Configuration

of the database vendor. The Oracle SequenceDAO uses an Oracle-specific thread-safe sequence. Failing to use the Oracle SequenceDAO with an Oracle database would not prevent the application from launching but could potentially cause thread-safety problems, which are notoriously difficult to diagnose [111]. A component developer can incorrectly believe that the application is going to use a MySQL database instead of an Oracle database and cause a configuration problem that is both dangerous and hard to identify.

If the SQL SequenceDAO is selected by the component developer when an Oracle database is present, which is a violation of the feature model composition rules, the mistake will not be clearly visible until a runtime error occurs. Furthermore, the runtime error that it will produce, a synchronization error, could be extremely difficult to diagnose and trace back to the feature selection mistake. Finally, the mistake will be identified only after any damage, such as data corruption, is done.

Security is another type of decision where a misunderstanding will produce a flawed but functional variant. Moreover, unlike misunderstandings that affect the visible functional properties of an application, a missing security requirement may be detected only after it has been exploited by an attacker and costly damage done. Thus, it is critical that these types of misunderstandings that do not lead to discernibly flawed variant selections be prevented.

Problem 4 - Important Information Fails to Flow Across Roles The involvement of multiple participants leads to situations where the decisions made by one or more participants are not synchronized. In most development processes, each role operates independently of other roles for significant periods of time. Synchronization of the decisions between roles is performed during weekly project meetings, testing, or application installation. Thus, a significant amount of time exists between synchronization points of the roles. If the decisions of the multiple parties are not in sync, the participants can select incompatible feature sets. If the incompatibility is discovered, one or more roles may need to roll back one or more potentially complex or costly decisions. If the synchronization mistake is not discovered, the application will not function properly.

Information may also fail to flow across roles because participants do not understand what decisions impact other roles. In Figure IV.3, each role needs to understand where its Venn Diagram's realm of responsibility overlaps another role's realm of responsibility. In the Pet Store example, the IT administrator enacts decisions on the target infrastructure, such as selecting the component container that will be used. The component developer may not have access to the target infrastructure and thus may not be aware that the IT administrator has selected a specific container. If the IT administrator selects and installs Tomcat without JTA support as the application container for the Pet Store and the component developer selects the JTA DAOs by adding them to the XML configuration file, a mismatch can occur that leads to a non-functional variant.

Open Problems in Applying Existing Configuration Approaches

Although various approaches have been presented for dynamically configuring component applications using feature models and other mechanisms, these approaches do not address the configuration challenges inherent in the enterprise Java applications for some combination of the following four reasons:

Tightly-coupled Top-down Approaches: Many existing approaches advocate the use of a tightly-coupled monolithic modeling approach where all configuration decisions are made in a single large model at design-time. Enterprise Java development involves multiple participants and thus makes synchronizing a single large model hard. The tight-coupling between roles also limits developer concurrency and does not integrate well with common development practices, such as extreme programming that focus on source code.

A further complication of tightly-coupled top-down modeling approaches are that they require all of the relevant information for each role's viewpoint be captured in a single model. Capturing all of the information required for each viewpoint in both an intuitive and usable manner is difficult. Additionally, a monolithic model potentially exposes participants from each role to irrelevant details from other roles. Even though different types of filtering mechanisms can be applied to limit what each viewpoint sees, these mechanisms are complex to develop since the complexity of the model may make it very difficult to predict which details are relevant and which are not.

Explicit Communication between Roles is Required: Current approaches require that all decisions that a role makes that affect another role must explicitly be communicated to the other role. Most approaches do not detail how this communication is accomplished. First, explicitly communicating decisions across roles is problematic because it is very difficult for each role to anticipate which of its decisions will affect another role and what role it will affect. These dependencies between the decisions of different roles can only be enforced if they are explicitly stated, which is challenging. Even if each role can identify which decisions affect other roles, the effect of these decisions must be evaluated from each other roles' viewpoint. Relating the affects of a role's decisions to the features of another role means that roles must relate features and decisions across viewpoints that they are not familiar with, which is tedious and error-prone.

Not all Variabilities/Decisions are Captured in a Model: Existing approaches assume that all decisions that are relevant to the configuration of the application are captured in the

model. Approaches do not detail how this is accomplished. Documenting all decisions and variabilities is not straight-forward. In some cases, a role may not deem that a variability is important enough to its viewpoint to be included in the model. However, another viewpoint may be affected by this undocumented variability. The complexity of the model and the distinct separation of the roles' viewpoints makes it hard for each role to understand if a variability needs to be documented for another role's sake.

Many development approaches, such as extreme programming, are focused on source code. Documentation, such as a model, is updated to reflect the state of the source artifacts. If a developer fails to document every source-level decision in the model, either because they forget or do not understand how the changes map to the model, a dangerous disconnect can occur that is not addressed by current approaches. Additionally, a development process may need to interact with legacy or third-party software for which there is no clear model nor way to produce a model. In this case, important decisions/variabilities are left out of the model.

No Runtime Feedback: In enterprise Java applications, it is not desirable to determine all application related decisions at design-time. For example, the concept of *cloning* (determining the number of instances of a feature), is a design-time decision in most approaches. In enterprise Java applications, the application container normally manages an object pool and dynamically changes the number of instances (clones) of the objects at runtime. Many other types of decisions, such as load-balancing policies, are also better determined dynamically at runtime.

Existing approaches do not account for how dynamic changes to the application that affect the feature model can be identified and understood. If the container changes a runtime policy, it is changing feature selections. If there is no way to relate runtime changes back to the feature model, the model becomes a design-time only artifact and none of the feature decisions made by the application container or other runtime decisions can be constrained or understood.

No Configuration Injection: Existing tools do not provide a mechanism to inject their configuration decisions from the model directly into the application. Instead, the tool derives a correct configuration and a manual process must be used to acutally implement the configuration (which is tedious and error-prone).

Solution Approach

The key to correctly configuring a Spring application's components is to (1) construct a coherent model of the feature decisions that have been made, (2) determine what variabilities have been constrained, and (3) set values for the remaining component variabilities that are consistent with the constrained variabilities. We propose that by executing a series of Java probes at application launch to identify frozen variabilities, formalizing and solving a CSP of the configuration problem, and dynamically rewriting the application's XML configuration files, we can eliminate the problems we have outlined.

The following list sketches our proposed solution to each of these problems:

- **Use Probes to Identify Constrained Variabilities:** Probes can be used to automate the discovery of the decisions made by each role. We show that probes can be constructed to cover the wide dimensions of configuration previously listed. Automatically identifying configuration decisions allows the feature selection process to ensure that the selected feature set conforms to any points of variability that have already been fixed. Probing of the environment also eliminates manual characterization errors. Just as unit tests can be written to test functionality, probes can be created for each feature to validate dependent features and properties.

- **Formalize Configuration as a CSP and Use a Constraint Solver to Derive Values for the Final Un-constrained Variabilities:** A constraint solver can handle the combinatorial complexity and interdependencies of feature selection that a manual process cannot address Problem 1. Furthermore, a constraint solver is guaranteed to

produce a correct selection with regard to the constraints (if a correct configuration exists). Although we do not provide a formal proof, we assert that for any configuration that could be deduced manually, a constraint solver can derive faster.

- **Generate Configuration Files from a Feature Selection:** A generative software development process can be used to automatically generate correct configuration files from the solution produced by the constraint solver. Our solution allows developers to annotate their configuration files to show how features are bound to actual configuration decisions. This allows the configuration engine to regenerate the configuration files for the selected feature set.

The Fresh Prototype

To demonstrate our approach for automating the collection of feature modeling decisions, phrasing a feature selection problem as a CSP, and using a constraint solver, we developed a prototype automated feature selection engine for enterprise Java applications. Our prototype is called *Fresh* and is based on the Spring framework [79]. Fresh allows the application configuration participants to describe the functional requirements, non-functional requirements, and a fitness function for choosing a configuration when multiple solutions exist. Fresh leverages this information and the Choco CLP solver [2] to derive a complete feature selection for a partially configured application. Finally, Fresh provides an XML annotation language that can inject the feature selection decisions into XML configuration files.

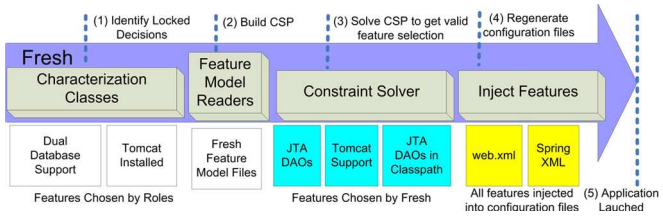


Figure IV.4: Fresh Application Configuration Process

Spring uses the factory [61] pattern to instantiate and wire enterprise Java components. Spring makes extensive use of Java reflection and allows the application components to be oblivious to the configuration process. At launch, a factory is created and initialized using one or more XML configuration files. The factory then uses the XML configuration files to determine what objects it constructs and how they are wired together. In the process of constructing objects, the factory may associate crosscutting aspect advice with them, generate dynamic proxies to perform remote invocations, load objects into a naming service, or perform numerous other complex application configuration tasks.

The Fresh prototype is implemented as an extension to the standard factories provided by Spring. When a Spring application factory attempts to load the application configuration files, Fresh probes the environment, runs the constraint solver, and rewrites the configuration files before they are returned to the Spring factory. Spring and the application components are not aware of the process. Furthermore, the Fresh extension can be swapped in and out of the application without affecting components or Spring.

As can be seen in Figure IV.4, in the first step of the Fresh configuration process, automated probes are run to aggregate the feature selection decisions of the roles into the feature model. Second, the decisions and feature model roles are transformed into a CSP. In the third step, Fresh uses the Java Choco solver to solve the CSP for a valid feature set. In step four, the configuration files for the application are regenerated and in step five control is passed to the Spring factory to initialize the application.

Fresh's configuration file annotation language is based on XML comments and does not interfere with the configuration directives. The annotations can be added to existing files or removed from the application entirely without affecting it. Both the container extension and the XML annotations allow Spring and the application components to be oblivious to Fresh.

Using the Target Environment as a Common Language

As we outlined earlier, there are multiple limitations of existing techniques that prevent them from being applied to enterprise Java application configuration. A key limitation is that a configuration process must provide a way of relating how the actions of different roles affect each other. Current approaches either attempt to use a single manually-produced large model to formally capture these interactions or rely on manually creating complex mappings across different models. The first approach suffers from the problems of a complex top-down approach, while the second approach forces the roles to explicitly specify complex cause and effect relationships across unfamiliar viewpoints.

Probing uses the target environment as a *lingua franca*. Each role expresses how changes in the target environment affect its model of the system. A probe checks a property of the environment and maps the property to a change in a role's model. For example, a probe can be used to automatically detect if JTA is installed and update the JTA feature in the component developer's model accordingly.

The first benefit of this approach is that it avoids a monolithic top-down modeling approach (Problem 1). Each role can use a model that is intuitive to the role's viewpoint. The models of each role are synchronized when the probes are run. The probes determine the changes that the roles have made to the target environment and update each role's model to reflect the configuration state. With this approach, each role maintains a model reflecting its viewpoint and is not tightly-coupled to the models of other roles.

The second advantage is that the roles do not have to explicitly detail how changes in their models map to changes in the models of another viewpoint (Problem 2). Instead, each role specifies how changes to the target environment affect it. Since the mappings are based on actual executable code, they provide much more well understood semantics. The mappings also do not require a participant in a role to understand another role's viewpoint. Each viewpoint maps its feature selections to changes in the target environment and each

role's probes translate the environment modifications into changes in the role's model. The environment serves as the common language, as seen in Figure IV.5.

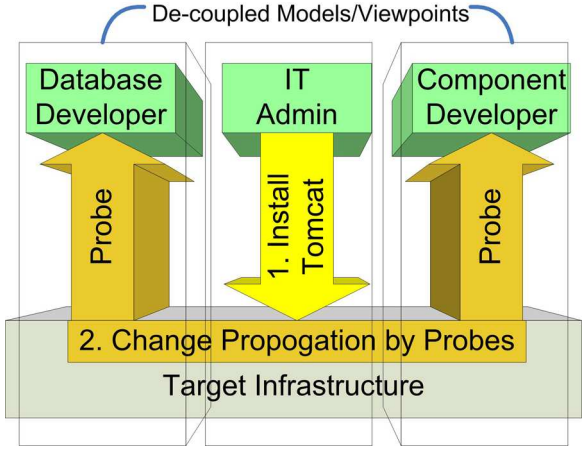


Figure IV.5: Synchronizing Role/Viewpoint Models through Probes

The third key attribute of the approach is that the probes do not differentiate between human induced environmental changes and dynamic changes to the environment from the container or other runtime actors. The container becomes another participant that may enact changes to the application at runtime. Since the probes are automated, they can be reused at runtime to detect changes to each role's feature model produced by the container. Runtime processes can become roles that provide feedback to the application eliminating Problem 3.

Since the dissemination of information across roles is automated by the probes, the approach can eliminate Problems 3 & 4. Automated probes are more reliable than human inspection of the configuration and environment. Rather than pushing information to the roles that are affected by environment changes, the probes pull the required information to each role avoiding communication failures and misunderstanding.

Probes are similar to Unit Tests, such as JUnit tests. Each probe checks a specific set of conditions and notifies the framework of the results of the tests. In JUnit, the tests report error messages indicating that the code failed to perform as expected. In Fresh, the probes

report the state of the application configuration and environment. Both Unit Testing and probing rely on developers writing correct tests of the conditions but can greatly improve both the reliability of correctness testing (configuration correctness for probes) and the efficiency of correctness testing. We assert that just as Unit Testing has been shown to be an integral part of application correctness testing, probing should be a part of application configuration.

Probing the Target Environment

The probes run by Fresh identify which features or components are present (*e.g.*, is JTA installed), what the values of different properties of the target infrastructure (*e.g.*, application server vendor, OS, RAM, etc.) are, and what configuration steps have been performed (*e.g.*, does a specific JMS queue exist). The probes produce a series of values for the variabilities in the model. For example, if JTA is installed, a probe may set the JTA feature to enabled or the JTAVersion attribute.

Fresh uses a plug-in architecture to allow product developers to create *characterization classes* that can be packaged with an application and run by Fresh to automate environment characterization. Each characterization class is a probe that is used to determine the value of one or more of the variabilities in the model used for the configuration process. Before Fresh performs its constraint-based feature selection, each characterization class is invoked. A characterization class performs a test on the target environment and returns a list of variable/value pairs representing characteristics of the target.

The values of the variables produced by characterization determine what points of product variability have already been fixed by each role. Fresh then derives values for the other variabilities to be correct with respect to these fixed points and the feature model constraints. The following list gives examples how configuration decisions from four common dimensions of configuration listed earlier can be discovered through characterization classes:

- *Local/Remote Addressing Configuration*: Local addressing configuration within Spring XML configuration files is handled by Spring. For external addressing, such as JNDI names or service URIs characterization classes can be created that attempt to resolve the object and if it cannot be resolved, set the corresponding feature to disabled.

- *Library Configuration*: A characterization classes can attempt to resolve Classes on which features depend using the Java Reflection API. For example, to test for JTA, a characterization class can perform a `Class.forName("javax.transaction.Transaction")`, which will throw an exception if JTA is not present. If the characterization class catches a `java.lang.ClassNotFoundException` exception, it indicates that JTA is not enabled.

- *Attribute Configuration*: A characterization class can obtain values for various attributes from environmental context classes, such as java.lang.Runtime, ServletContext, or ApplicationContext. These context classes can provide critical infrastrural attributes such as JVM version, OS, RAM, etc. for the CSP variables. A characterization class may also determine attribute values by instantiating one or more application components and using getter methods or the Java Reflection API to obtain member variable values.

- *Infrastructure Configuration*: Characterization classes can be used to test that specific infrastructural features are running. For example a class can be created that attempts to connect and post a message to a required JMS queue or run a query against a database table. If the queue does not exist or an exception is thrown the feature variable for the queue can be set to disabled. Similarly, the database configuration can be checked by creating a class that obtains an instance of the DB driver and attempts to perform queries to check that the tables are configured properly.

The above list is by no means exhaustive. Numerous other types of characterization classes, such as running a CPU benchmark, can be used to obtain complex properties.

In most cases, if the application is affected by a configuration decision, it can probe its environment to determine the value of that point of configuration variability.

Class characterization allows the Fresh feature selection engine to determine what variabilities have been fixed in the product. After correctly determining what variable parts are fixed, the constraint solver can select features to ensure the application functions properly with respect to these fixed parts and the application requirements.

Feature Selection as Constraint Satisfaction

The first problem is that the configuration process is complex due to the large number of constraints and role viewpoints involved. Significant work has been done in applying different algorithmic techniques to handling this complexity. The probing techniques that we have described could potentially be used with any of these algorithmic approaches. For Fresh, we chose to apply the extensive research and tools for Constraint Logic Programming (CLP) [77] to manage this complexity.

Fresh transforms a feature model and set of non-functional requirements into a CSP. The feature model and the non-functional requirements are specified through Fresh configuration files which reside in the classpath of the Spring application. We extend the reduction of feature selection presented by Benavides et al. [22] to include cardinality-based constraints, feature references, and resource consumption constraints. By building a formal model of feature selection as a CSP [134], Fresh can use a constraint solver to 1) check the correctness of a configuration and 2) derive valid values for unconstrained variabilities in a partially configured application. Using a constraint solver to perform both configuration validation and completion eliminates problem 1. In this section, we show how Fresh reduces feature selection to a constraint satisfaction problem.

A CSP is a problem that involves finding a labeling (a set of values) for a set of variables that adheres to a set of labeling rules (constraints). For example, with the constraint "$X < Y$", $X = 3, Y = 4$ is a correct labeling of the values for X and Y. Typically, the more

variables and constraints that are involved in a CSP, the more complex finding a correct labeling of the variables is.

Selecting a feature set for a product can be reduced to a CSP. Fresh constructs a set of variables $P_0 \ldots P_n$, with domain $[0,1]$, to indicate whether or not the ith feature is present in a feature set. Thus, a feature set becomes a binary string where the ith position represents if the ith feature is present. Satisfying the CSP for feature selection is devising a labeling of $P_0 \ldots P_n$ such that the composition rules of the feature model are adhered to.

The functional requirement rules for a feature model ensure that only a coherent set of features is selected. For example, in the Pet Store, if the *JtaDAOs* feature is chosen, the *JTA* feature must also be selected. To phrase this rule using our CSP model of feature selection, we can say that if the JtaDAOs feature is represented by the variable P_1 and the JTA feature is represented by the variable P_2, then $P_1 = 1 \rightarrow P_2 = 1$.

CSPs may incorporate constraints based on the conjuction or disjunction of several constraints on other features. One example of this is the extension to cardinality constraints on features proposed by Czarnecki et al. [49]. Their approach extends cardinality constraints to include a sequence of intervals. For example, assume that the Pet Store can use [1..2], or [4..4] different remoting mechanisms from the remoting feature group. If the variable P_0 represents the Pet Store, and the variables $P_{15} \ldots P_{18}$ represent the remoting features, we can transform this interval sequence into the constraint: $P_0 = 1 \rightarrow (\sum Pt_{15} \ldots P_{18} > 0) \wedge (\sum Pt_{15} \ldots P_{18} \leq 2) \vee (\sum Pt_{15} \ldots P_{18} = 4)$.

The CSP model of feature selection can be extended with new requirement types by translating these constraints into a CSP model. We define a resource consumption constraint that prevents a resource from being overconsumed by a chosen feature set. For example, assume that the ith feature consumes an amount of RAM denoted by the variable $Pram_i$. If the total amount of RAM available in the system is denoted by the variable *Ram*, we can create the constraint: $\sum (Pram_0 * P_0) + (Pram_1 * P_1) + \ldots (Pram_n * P_n) \leq Ram$. This constraint limits the total memory consumed by the selected feature set to be less than or

equal to the RAM available in the system. Thus, the CSP model is extensible and can incorporate new requirement types between features as they emerge.

One of the benefits of reducing feature selection to a CSP is that we can unify the non-functional and functional requirements into a single logical model based on constraint logic. Let's assume that the DualDatabaseSupport feature and the JTADAOs are represented by the variables P_{10} and P_{11} respectively. We can encode the rule that the DualDatabaseFeature requires JTADAOs as $P_{10} = 1 \rightarrow P_{11} = 1$. Assume that the developer requires at least JTA version 1.01 for functional reasons. The IT Administrator, however, requires a version number less than 1.03 because only versions up to that point have been through the organization's security and stability certification process for production environments (a non-functional requirement). This non-functional constraint can be encoded as $P_{11} = 1 \rightarrow (JTAVersion \geq 1.01) \wedge (JTAVersion < 1.03)$.

The *JTAVersion* variable is a new variable introduced to store the version number of the JTA version installed on the target host. The value of this variable can be populated from a configuration file. For each infrastructural property that a non-functional requirement depends on, a developer can introduce a corresponding variable into the CSP. If a requirement depends on the response time of a component X, a $ComponentXResponseTime$ variable can be created. Any number of variables can be introduced to represent the target host and component properties. By formalizing the feature selection problem as a CSP, there is now a clear relationship between the selection of the DualDatabaseSupport feature, $P_{10} = 1$, and its implications.

Aggregating Feature Models and Feature Requirements

At startup, one or more directories are provided to Fresh that contain the feature models for each role, non-functional requirements, and configuration mechanisms for the products. Fresh constructs its CSP by composing the feature models of each viewpoint and the non-functional requirements it discovers. Adapters are used to load the feature model

Steps for Initial Deployment	Lines of XML Changed	Roles Involved	Location of Change
1. Change Datasource Driver Class/URI	1	Database Dev/IT Admin	dataAccessContext.xml
2. Remove Standard Sequence DAO	3	Database Dev/IT Admin	dataAccessContext.xml
3. Add Oracle Sequence DAO	3	Database Dev/IT Admin	dataAccessContext.xml
4. Add Mail Sender Bean to application.xml	3	IT Admin	application.xml
5. Add Insert Order Pointcut	1	Component Dev	application.xml
6. Add Email Advice	3	Component Dev	application.xml
7. Add RMI Remoting Service Export	6	Component Dev/IT Admin	application.xml
Total Steps: 7	Total Lines of XML: 20	Roles Involved: 3	Files Involved: 2
Steps for Second Deployment			
1. Change Datasource Driver Class/URI	1	Database Dev/IT Admin	dataAccessContext.xml
2. Remove Standard Order DAO	3	Database Dev/IT Admin	dataAccessContext.xml
3. Add MSSQL Order DAO	3	Database Dev/IT Admin	dataAccessContext.xml
4. Remove Oracle Sequence DAO	3	Database Dev/IT Admin	dataAccessContext.xml
5. Add Standard Sequence DAO	3	Database Dev/IT Admin	dataAccessContext.xml
6. Remove RMI Service Export	6	Component Dev/IT Admin	application.xml
7. Remove Mail Sender Bean	3	IT Admin	application.xml
8. Remove Insert Order Pointcut	1	Component Dev	application.xml
9. Remove Email Advice	3	Component Dev	application.xml
Total Steps: 9	Total Lines of XML: 26	Roles Involved: 3	Files Involved: 2

Figure IV.6: Cost of a Manual Approach to Configuration for the Scenario

and non-functional requirements. By default, Fresh provides adapters for reading feature models and non-functional requirements that use a syntax similar to cascading style-sheets. Adapters can be plugged-in to read other formats, such as XMI models produced by the Eclipse Modeling Framework (EMF) [30].

Since specifying feature dependencies and constraints using CSP syntax is not ideal for most development processes, we developed a Domain-Specific Language (DSL) for specifying feature models and constraints. The feature modeling language, called *Feature Styles*, allows a product developer to specify the features in the model, the dependencies between features, and the non-functional requirements associated with each feature. The language uses a simple textual notation and is not difficult to grasp.

Fresh supports the following dependency rule types:

- *Required* features that must be present for a feature to function properly. JTADAOs *requires* JTAEnabled.
- *Excluded* features that cannot be present at the same time as a feature. OracleSupport *excludes* SQLSequenceDAO.
- *Cardinality* constraints on required features. OrderRemoting requires a user to *select* [1..*] of the features HessianRemoting, RMIRemoting, and BurlapRemoting.

Product developers use these dependency rule types to build complex feature models for a product. Previously, we detailed how these rules are translated into a Constraint Satisfaction Problem (CSP) [134] for a Java Constraint Logic Programming (CLP(X)) solver [77]. The solver uses these rules to guarantee that only compatible and coherent sets of features are selected for a variant.

The non-functional requirement specification language of Feature Styles allows product developers to leverage the characterization variables produced from the automated environment characterization. Each feature can be annotated with constraints based on the variable names which must hold for the values assigned to the attributes of the target environment. Fresh provides constraints based on conjunctions or disjunctions of $>, <, =, != , =<, >=$.

A feature can be annotated with any number of constraints on the attribute values. Developers use these constraints to encode the non-functional requirements of the features. As with the feature dependency rules, the constraints are encoded into the CSP provided to the feature selection engine.

The full feature specification for the JtaDAOs is shown below:

```
JtaDAOs {
  Requires: JTA, DatabaseDriver;
  Excludes: NonJtaDAOs;
  JTAVersion > 1.01;
  JTAVersion < 1.03;
}
```

Results from Experiments with Fresh

To demonstrate the reduction in manual configuration complexity provided by Fresh, we devised a realistic configuration scenario for the Pet Store example. In this scenario, Pet Store has a base deployment descriptor (the out-of-the-box descriptor included with the Spring Pet Store) that must be modified to install the Pet Store on Tomcat with an Oracle

Database, Email Notification, and RMI Remoting. Pet Store is then migrated to a new target where it is hosted on JBoss with an MSSQL database, no RMI Remoting (to avoid conflicts with the application server), and no Email Notification (email order notification is handled by a new payment processing application when the customer's credit card has been charged). The results in this section show that Fresh's automated configuration approach can reduce the total number of steps required to configure an enterprise Java application by 72% and the total lines of XML code by 92%.

Testing Configuration Complexity

In the test scenario, we compute the configuration cost in lines of XML code that must be changed. We assume that optional components, such as Email Notification's Email Advice, are not initially present in the deployment descriptor. When a role selects a feature requiring a component, the component is added to the configuration files. Table IV.6 shows the steps involved in configuring the Pet Store for the first deployment configuration.

As shown in Table IV.6 there are many steps, roles, and files involved. To migrate to the second target environment, the roles must remove some of the initially chosen components (*e.g.*, Oracle Sequence DAO, Email Advice, Order Pointcut, etc.) and add other new components (*e.g.*, MSSQL Order DAO). The steps involved in the migration are shown in Table IV.6.

Table IV.6 also shows that there are a significant number of steps and changes required to migrate to the new setup. Each change in the target environment or desired feature set will necessitate similar reconfiguration costs. Moreover, if the application is widely used, the support team for each application *instance* must pay this configuration cost.

We then performed the same migration experiment using Fresh. Fresh required an extra initial investment of building a basic feature model for the features from the migration experiment. It also required the addition of comments to the Pet Store's XML configuration

files that mapped features to XML configuration directives (so that the configuration files could be regenerated). The initial Fresh configuration overhead is shown in Table IV.7.

Steps to Enable Fresh	Lines of XML Changed	Roles Involved	Location of Change
1. Build Fresh Feature Model	6	Component Dev/IT Admin/Database Dev	petStoreFeatureModel.xml
2. Add Application Server Detection Probe	1	Component Dev	probes.xml
3. Add Database Detection Probe	1	Database Dev	probes.xml
4. Make Sequence DAO Switchable	4	Component Dev	dataAccessContext.xml
5. Make Order DAO Switchable	4	Component Dev	dataAccessContext.xml
6. Make Mail Sender Switchable	4	Component Dev	application.xml
7. Make Insert Order Pointcut Switchable	2	Component Dev	application.xml
8. Make Email Advice Switchable	4	Component Dev	application.xml
9. Make RMI Remoting Service Switchable	7	Component Dev	application.xml
Total Steps: 9	Total Lines of XML: 33	Roles Involved: 3	Files Involved: 4
Steps for Initial Deployment			
1. Change Datasource Driver Class/URI	1	Database Dev/IT Admin	dataAccessContext.xml
2. Change Desired Features	1	IT Admin	dataAccessContext.xml
Total Steps: 2	Total Lines of XML: 2	Roles Involved: 2	Files Involved: 1
Steps for Second Deployment			
1. Change Datasource Driver Class/URI	1	Database Dev/IT Admin	dataAccessContext.xml
2. Change Desired Features	1	IT Admin	dataAccessContext.xml
Total Steps: 2	Total Lines of XML: 2	Roles Involved: 2	Files Involved: 1

Figure IV.7: Fresh Configuration Cost for the Scenario

Fresh requires an initial overhead of 33 lines of XML/Feature Model configuration. This extra configuration code allows Fresh to (1) detect the database type used (inferred from the data source driver class), (2) detect if a web container or application server is the container (by checking for EJB-specific classes), and (3) add/remove XML configuration directives for the components of enabled/disabled features, respectively. Although the initial cost of enabling Fresh is higher than a traditional manual approach, this price is paid only once, rather than each time the application is deployed.

Table II shows the steps required for installing the Pet Store on the initial target with Oracle and Tomcat. Only two configuration steps are required. First, the correct database driver class is added to the configuration and then the desired feature set is specified as Tomcat, Oracle, etc. Fresh performs all other XML configuration tasks, including deriving a valid feature selection with respect to the desired features.

Table II also summarizes the steps required to perform the second migration to the JBoss/MSSQL environment. Again, only two steps are required: setting the database driver and updating the desired features. These two steps provide a significant improvement over the manual approach, where 26 lines of XML were changed for the same migration.

Table IV.8 compares the totals for the manual vs. Fresh configuration approaches. Fresh initially incurs a marginal configuration cost for building a feature model and annotating the XML configuration files for the Pet Store. After the migration to the second target environment, however, Fresh reduced the complexity of configuring the Pet Store by 9 lines of XML configuration. Moreover, for each configuration, Fresh derived a valid feature set based on the desired features specified by the roles. With a manual approach, this derivation is not automated and can produce numerous types of errors. In contrast, Fresh assures that each configuration is correct by using a constraint solver to derive a configuration based on the feature model constraints and constrained variabilities.

When the cost of configuring the Pet Store over 100 separate deployments is analyzed, the benefits of the Fresh approach are amplified. At the minimum (assuming that each deployment uses the default configuration), the manual approach requires 200 configuration steps and 600 lines of XML changes. The total cost of the manual approach can be over 900 configuration steps and 2,600 lines of XML code, however, if the default configuration is not used on each deployment, which we assume is common.

With Fresh, conversely, the total configuration steps are fixed at 209 and the total lines of XML configuration at 233. At a minimum Fresh requires 62% less lines of XML configuration changes and a maximum of 92% less. Step-wise, Fresh uses at most 4.5% more steps but can also use 72% less total steps. As the number of deployments of the Pet Store increases, Fresh's development savings also increase. With increased numbers of deployments, the initial investment cost of Fresh becomes insignificant compared to the savings.

The intial cost paid to enable Fresh is incurred by the original application developers. Applications are often developed by one group, yet have hundreds or thousands of instances installed and maintained by other groups, *e.g.*, testers and users. Moreover, the users often perform the final configuration, such as choosing the database, OS/middleware version, network configuration, etc. These users rarely possess the same intimate knowledge of the application, so they are more likely to make errors or produce poor configurations.

With Fresh, conversely, the initial developers can package their intimate feature model, non-functional requirement, and configuration knowledge with the application.

Since this expert configuration information is packaged with the application, users focus on declaratively informing Fresh what they want, rather imperatively programming new configurations to provide what they want. Application users can therefore benefit from the expert configuration knowledge of the original developers, which is much harder with conventional manual approaches. Moreover, Fresh greatly reduces the configuration cost for users since they do not pay the initial Fresh integration cost, which is borne by the original application developers.

	Total Steps	Lines of XML Changed	Total Roles Involved	Files Changed
Initial Overhead				
Manual	0	0	0	0
Fresh	9	33	3	4
Configuring for Tomcat/Oracle				
Manual	7	20	3	2
Fresh	2	2	2	1
Migrating to JBoss/MSSQL				
Manual	9	26	3	2
Fresh	2	2	2	1
Scenario Totals				
Manual	16	46	3	2
Fresh	13	37	3	4
Configuration Cost Per Deployment				
Manual	2min to 9+max	6min to 26+max	3	2
Fresh (not counting initial overhead)	2	2	2	1min to 2max
Total Configuration Cost Over 100 Deployments				
Manual	200min to 900+max	600min to 2600+max	3	2
Fresh (including initial overhead)	209	233	3	4

Figure IV.8: Manual vs. Fresh Configuration Cost Totals

Fresh Performance Overhead

To determine the performance penalty for deriving a configuration with a constraint solver and rewriting an application's configuration files, we built a set of experiments to test the startup time of Pet Store. We first devised several new feature models of increasingly finer granularity to see how long application startup took with varying feature model sizes. Feature models of 60, 80, and 100 features were created. The 60, 80, and 100 feature models were actual feature models of the Pet Store. The 60 feature model did not account

for features related to the web-tier of the Pet Store. The 80 feature model added features for the web-tier and Spring's Web Flow front end. The 100 feature model added features for the alternate Apache Struts front-end of the Pet Store's web-tier.

Each test was built so that the feature set derived from Fresh would lead to an identical application configuration, *i.e.*, produce the same set of XML configuration directives. We also reproduced this configuration statically in XML to launch without Fresh and derive the overhead incurred by using Fresh. We launched Pet Store in Tomcat 6.0.9 using JDK 1.5.0_11 on an IBM Think Pad T-43 with a 1.86GHZ Pentium M processor, 1.5GB of RAM, and Windows XP. We then tested the time needed to launch Pet Store within Tomcat and configured it using Fresh with each feature model. The results were compared to the static configuration launched in Tomcat without Fresh and are shown in Figure IV.9.

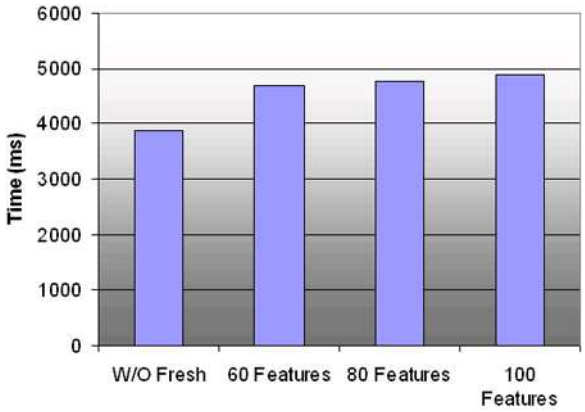


Figure IV.9: Pet Store Initialization Time in Tomcat

Figure IV.9 shows that using Fresh with a 60 feature model required an extra ∼800ms to launch vs. a static configuration. For 100 features, the total penalty was ∼1,000ms. This overhead should be acceptable for many enterprise Java application deployment scenarios because it is only incurred once at application startup.

CHAPTER V

AUTOMATED CONFIGURATION INTEGRATION FOR CORBA COMPONENT MODEL APPLICATIONS

Introduction

Distributed real-time and embedded (DRE) systems are increasingly being built using component-based technologies. Component technologies facilitate software reuse across applications by allowing the dynamic assembly of applications at deployment time via configuration scripts. The late-binding properties of component technologies allow application developers to reuse existing software and reduce costs by leveraging commercial-off-the-shelf (COTS) components.

Application developers have traditionally used tightly-coupled proprietary solutions to handle the tight requirements and resource restrictions of DRE systems. Composing a component-based application from components that are not specifically designed for the individual application poses a number of challenges. For example, highly specialized components can make assumptions, such as the what type of underlying operation system will be used, that reusable components cannot make. These assumptions can help improve performance (*e.g.* using specialized APIs) at the cost of reusability. Because DRE systems often operate in environments with little resource slack, being unable to make these key assumptions makes it difficult to find a configuration that meets the required timeliness, safety, and other non-functional properties.

A further challenge of configuring DRE systems is that the configuration process must integrate the concerns of numerous participants divided into multiple roles, such as component developers and hardware developers. Each role has a unique viewpoint on what it considers the ideal solution. Thus, each role attempts to pull the solution in the direction

that best meets the requirements it is responsible for, such as power consumption or security functionality. These multiple opposing viewpoints make it hard to find a configuration that satisfies the requirements of each role simultaneously.

For example, in applications developed using the Lightweight CORBA Component Model (CCM) [19, 138], component developers often prefer to host the applications on the most powerful processing hardware available and be allocated as much network bandwidth as possible to make their realtime scheduling deadlines easier to meet. Hardware developers, in contrast, will attempt to use the least powerful processors that are adequate for the job to minimize power consumption, weight, and cost to make the system more efficient. Component assemblers (the role that creates instances of components and wires them together) will want to have the widest array of component types and implementations available to compose a solution. Testers and certification engineers, conversely, will want to limit the number of possible application parts to reduce testing and verification complexity.

Even after a configuration is found that satisfies the numerous/competing concerns of the roles, implementing the configuration can be tedious and error-prone. In particular, multiple roles must coordinate and correctly edit configuration scripts required to assemble the application. Component developers instruct component assemblers on the port functions and requirements. Component assemblers wire the components together and dictate the CPU and memory requirements to application deployers (the role responsible for placing components on nodes). Deployers obtain the correct binaries from application packagers and place them onto the appropriate nodes. Miscommunication between roles, subtle mistakes in configuration scripts, and other hard-to-diagnose errors can allow configuration errors to creep into applications and are thus a major contributor to application failure [50].

This chapter extends our previous work [144] on simplifying the configuration of enterprise Java applications. We include new contributions that show how our original Java-based approach can be generalized to other types of component-based systems. In particular, the chapter shows the complexity of configuring DRE component-based systems through a Lightweight CCM avionics application. We demonstrate how the same challenges that plague enterprise Java configuration extend into DRE component-based systems (and are possibly even more challenging). Moreover, the chapter presents results showing that the same reductions in manual configuration effort we achieved applying Fresh to enterprise Java can be obtained by applying Fresh to Lightweight CCM.

At the heart of our approach is a model-driven engineering (MDE) tool called *Fresh* that is designed to reduce the complexity of deriving a correct application configuration and implementing the configuration in configuration scripts. Fresh simplifies and improves the correctness of configuring DRE component-based applications by:

1. Capturing configuration rules through feature models, which describe application variability in terms of differences in functionality.

2. Translating an application's feature models into a constraint satisfaction problem (CSP) and using a constraint solver to automatically derive a correct application configuration for a requirements set,

3. Facilitating configuration optimization for a requirements set by providing a configurable cost function to the constraint solver to select optimal configurations, and

4. Providing an XML configuration file annotation language that allows it to inject configuration decisions into configuration scripts directly and reduce configuration implementation errors.

Fresh uses feature models [81] to describe the rules for configuring an application. Feature modeling can be used to describe an application's configuration rules in terms of

variations in functionality. For example, an avionics mission computing application that could be built using different satellite positioning systems could be described by feature models in terms of its:

1. Variations in functional capabilities (*e.g.*, GPS vs. Galileo satellite positioning sensors),

2. Variations in non-functional properties (*e.g.*, processor power consumption, weight, etc.), and

3. Constraints between features (*e.g.*, ARM binaries for the Galileo positioning sensor require an ARM processor)

Feature modeling provides an intuitive model for describing application variability and has been applied to a number of domains ranging from automobiles [108] to applications for mobile phones [149]. Deriving a valid configuration from a feature model involves:

1. Selecting required features (*e.g.*, Galileo),

2. Selecting features corresponding to the capabilities of the target platform (*e.g.*, ARM), and

3. Deriving any remaining features needed to create a complete and valid configuration (*e.g.*, ARM Galileo binaries)

Avionics Application Example of a DRE System

As a representative example of a component-based DRE system, we use the BasicSP scenario, which is based on the Boeing Bold Stroke avionics mission computing platform [126] shown in Figure V.1. The BasicSP application includes several Lightweight CCM components. One component is an avionics navigational display that receives updated airframe position coordinates from a positioning sensor. The rate generator component sends out a periodic pulse that causes the positioning sensor to update its current

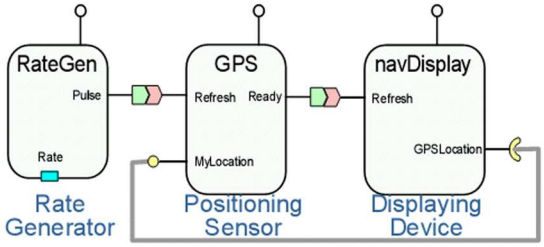


Figure V.1: Architecture of the BasicSP Avionics Example

coordinates. Once the coordinates are updated, the positioning sensor sends a ready signal to the display component to update its coordinates.

Lightweight CCM supports the deployment and configuration of components based on XML configuration files. An emerging trend in the development of avionics systems is to use component-based middleware along with a product-line architecture (PLA) [38]. A PLA consists of a group of core assets, such as reusable software components and test cases, and a set of rules for composing the assets into a product variant. When an application for a new set of requirements is needed, an application variant is configured from the reusable assets to meet the new requirement set. A PLA helps reduce development costs by reusing existing core assets and codifying the process of correctly configuring assets into an application variant.

The BasicSP product-line. To demonstrate the complexity of declaratively configuring a set of assets into a variant, we created a product-line from the BasicSP example. The modified BasicSP example includes multiple satellite-based positioning systems that can be leveraged as the positioning sensor to provide the coordinates of the airframe. Moreover, the product-line includes different variations in the processors that can be leveraged to run the rate generator, positioning sensor, and display.

Configuring a variant from the BasicSP product-line involves several participants divided into different roles [139]. For example, component developers are responsible for producing software components, application assemblers composes software components

into applications, application deployers determine which processing units host which components, and infrastructure developers determine what processing units are available in the airframe. Each role has its own viewpoint and concerns regarding the properties of the configuration. For example, component developers are focused on the functional aspects of the components and their real-time scheduling, whereas infrastructure developers are geared towards the weight, power consumption, and cost of the available processing units.

A valid BasicSP variant must integrate the concerns of each viewpoint into a functioning application. To codify the rules for configuring a proper variant, we produced feature models that relate how the different points of application variability (such as the number and types of processing units) affect each other (*e.g.*, the available processing power will restrict the components that can be used). Feature modeling describes an application's points of variability in terms of variations in functional and non-functional capabilities. Moreover, feature modeling provides a method of codifying the rules that restrict how selecting one feature affects how other features can be selected.

An overview of the BasicSP feature modeling notation. Figure V.2 shows the feature model for BasicSP. BasicSP requires the *Rate Gen*, *Position Sensor*, and *Display* features, which is denoted by the filled oval above each of these features. Moreover, BasicSP requires one to three processors, which is denoted by the "[1..3]" cardinality label applied to the Processor feature. Figure V.3 contains additional feature modeling notations. The *Rate*

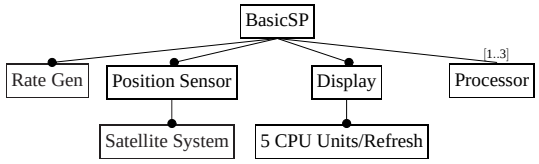


Figure V.2: Feature Model of BasicSP

feature requires exactly one (an XOR relationship) of the features *20hz*, *25hz*, and *30hz*. Finally, Figure V.6 contains the notation for optional features. The *x86* feature can (but is not required to) include the *GPS* feature, which is denoted by the unfilled oval.

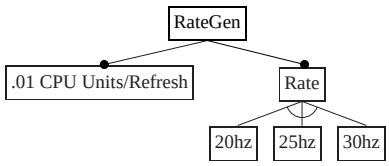


Figure V.3: Feature Model of the RateGen

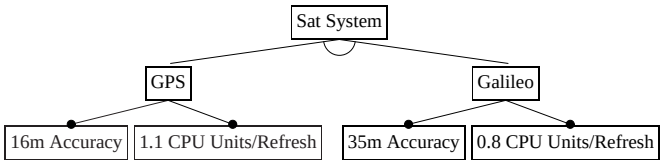


Figure V.4: Feature Model of the Available Satellite Systems

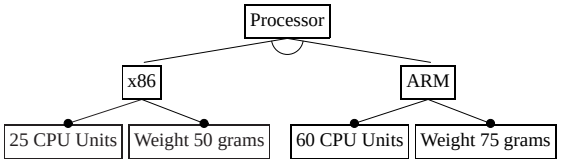


Figure V.5: Feature Model of the Processor Options for BasicSP

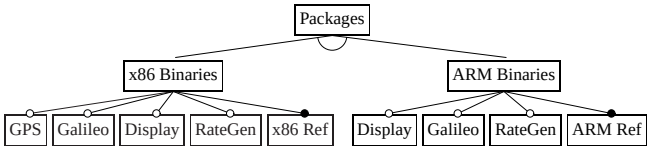


Figure V.6: Feature Model of the Packaging Options for BasicSP

Challenges of Configuring Component-based Applications for DRE Systems

This section outlines the key challenges of configuring a component-based application (such as BasicSP) for DRE systems (such as avionics mission computing). In general, it is hard to configure component-based applications for DRE systems due to the numerous competing concerns, such as balancing processor power consumption against required processing power. This problem is exacerbated by the multiple roles and viewpoints in the configuration process.

Challenge 1: Configuration Complexity

Each configuration choice in a component-based application may affect numerous other decisions that can be made by other roles. In many cases, no formal documentation of these cause/effect relationships exists. Even when semi-formal documentation, (*e.g.*, feature models) exists, the large number of components, numerous cause/effect relationships, and complex global constraints (*e.g.*, limitations on available memory), make it hard to derive a valid configuration manually.

In the BasicSP application, for example, selecting the GPS component has numerous side effects on further configuration decisions. The total number of CPU Units consumed per second cannot exceed the rated CPU Units per second of the processors. If the GPS component is selected along with a RateGen at 25hz, the GPS component will consume 27.5 CPU Units on its host. This combination of a GPS at 25hz precludes using the x86 based processor.

The problem with the feature combination outlined above, however, is that there are no binaries to run the GPS component on the ARM processor. Although the configuration appears correct, a subtle combination of a resource constraint and a packaging limitation (that may not be realized until deployment time) makes the combination invalid. These long chains of cause/effect relationships are hard to predict and handle manually.

Challenge 2: Incorrect configuration implementation

Configuring a component-based application involves correctly editing numerous configuration files (*e.g.*, CCM XML deployment descriptors), preparing the target infrastructure (*e.g.*, installing required libraries and starting supporting processes), and installing the application's own binaries on its target hosts. These configuration tasks are spread across multiple roles participating in the application's configuration. For example, the application deployer will install the application's binaries on the correct hosts and the application

assembler will create the XML configuration files specifying how to connect components together.

The BasicSP example uses multiple XML deployment descriptors, which provide standardized Lightweight CCM mechanisms to specify configuration directives. Numerous changes must be made to BasicSP's XML deployment descriptor, however, to change the satellite system used as a position sensor. First, the specification of the component used to implement the position sensor must be changed (performed by component assemblers). The new implementation specification of the position sensor must also include the IDs of its associated implementation artifacts (*e.g.*, dynamic link libraries). The IDs for these artifacts are produced by component packagers. If the new position sensor uses a different interface than the previous position sensor, the component assembler must also update the wiring of the components by changing the ports and facets involved in the position sensor's refresh signal, the display's coordinates input, and the display's refresh signal.

The numerous configuration activities that must be coordinated across the various participating roles makes manual configuration of a component-based application tedious and error-prone. Simple mistakes, such as packaging the application with binaries for the wrong processor architecture, can cause the application to crash at launch. More subtle mistakes, such as accidentally using the identifier for the 30hz RateGen instead of the 20hz RateGen, will produce an application that launches correctly but fails under load. Figure V.7 shows the multiple dependencies between roles responsible for configuring BasicSP. As shown in this figure, coordinating multiple roles and executing a complex configuration is tricky.

Solution Approach: An Automated Configuration Engine for Lightweight CCM Applications

This section describes the *Fresh* configuration engine and how it addresses the challenges of configuring component-based applications for DRE systems.

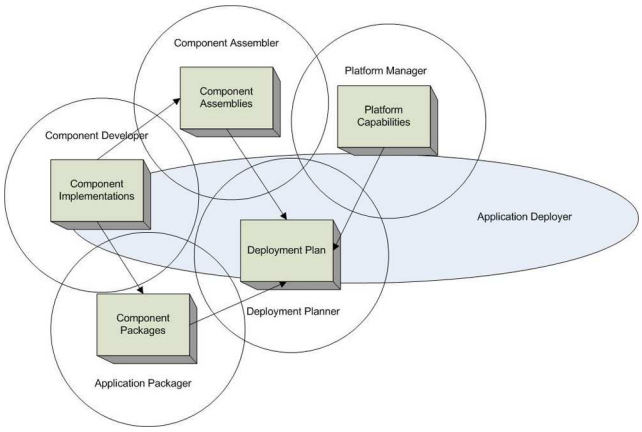


Figure V.7: Configuration Dependencies between Roles for BasicSP

Capturing Configuration Rules in Feature Models

One of the key steps towards correctly configuring a component-based application is to capture the rules for configuring the application. Fresh uses feature models [81] to describe the rules for configuring an application.

Fresh's feature modeling language is implemented as both a textual Domain-Specific Language (DSL) and a graphical modeling tool in Eclipse. The graphical modeling tool is based on top of the Generic Eclipse Modeling System (GEMS) [107], which is an MDE tool for rapidly creating diagram-based modeling tools from a metamodel.

Automating Configuration Derivation

In addition to providing an intuitive interface for documenting configuration rules, previous research [22] has demonstrated reductions from feature models to constraint satisfaction problems (CSPs). Once a CSP formulation of a feature model has been obtained, a constraint solver can be used to derive a correct application configuration. Using a constraint solver to derive an application solver addresses Challenge 1 by eliminating manual derivation. Moreover, using a constraint solver to derive an application configuration has the following benefits over a manual configuration process:

- The correctness of derived configurations is guaranteed with respect to application constraints,
- The solver can identify if no valid solution exists that meets the requirements,
- A cost function can be used to select a configuration that optimizes key properties of the solution,
- No manual effort is required to reconcile the complex cause/effect relationships described, and
- The solver can find a solution that reconciles opposing viewpoints and concerns involved in configuration (if such a solution exists).

A missing element of existing mechanisms for translating feature models into CSPs and satisfiability problems [93], is that these approaches do not take into account resource constraints, which are important in DRE systems. In previous work [144], we have extended the work in [22] to incorporate resource constraints and show that it is feasible to consider them for certain size problems. The exact upper bound on a feasible resource problem varies from problem instance to problem instance but is typically not a limitation of automated configuration from CSPs.

Configuration Injection

Along with the difficulty of deriving a valid configuration, we described the complex coordination needed to implement a valid configuration in an application's configuration scripts. To help decrease the complexity of implementing a configuration, Fresh includes an XML configuration file annotation language that can be used to inject a derived configuration directly into an application's configuration files.

Fresh's configuration annotation language includes a number of annotations that can be used to match an XML configuration file to a derived solution, including mechanisms for:

1. Inserting different attribute values based on the selected feature set,

2. Removing configuration sections,

3. Conditionally inserting configuration sections based on the selection of specific feature combination, and

4. Performing template-based duplication of configuration directives for specific feature types.

Fresh's annotation language is based on XML comments and does not change the structure or semantics of the original configuration language, as can be seen in Figure V.8. If the application must be configured without Fresh in certain circumstances, therefore, the Fresh annotations need not be removed to configure the application normally. By automatically injecting configuration decisions directly into XML configuration scripts, Fresh significantly reduces manual configuration effort, and configuration errors.

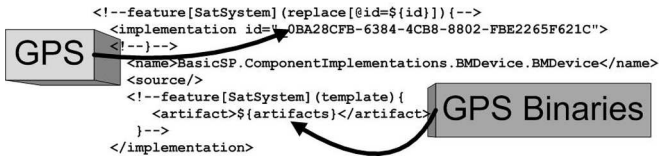


Figure V.8: Fresh XML Annotations

A final benefit of directly injecting configuration decisions into application configuration files is that the bindings for each configuration decision can be unit tested. For example, a unit test can be built to ensure that when the GPS component in BasicSP is selected, the correct XML configuration directives in the component deployment descriptor are produced. After validating the injection of each feature into the configuration files, application developers can be certain that future configurations involving the tested features will be implemented correctly.

With a manual configuration process, conversely, each time a new configuration is produced the configuration files must be checked to ensure that no mistakes are made. In

some cases, an application may be delivered to customer who are responsible for properly implementing a configuration, which they may not do correctly. Using Fresh's automated approach, in contrast, enables customers that receive an application to ensure it is configured correctly to meet its requirements.

Empirical Results

To demonstrate the reduction in manual configuration complexity provided by Fresh, this section evaluates a scenario in which the BasicSP example has the position sensor changed from GPS to Galileo. In this scenario, BasicSP has a base deployment descriptor (the out-of-the-box descriptor included with the CIAO Lightweight CCM container implementation) that must be modified to:

1. Add the required implementation of Galileo,
2. Create an instance of the Galileo component,
3. Connect the Galileo component to the RateGen and Display, and
4. Add Galileo to the deployment plan by specifying its servant, executor, and stub along with their associated implementation artifacts.

The Galileo and GPS position sensors possess the same basic functionality but name their ports/facets slightly differently. Thus, although the two can be swapped, their connections and various deployment descriptor configuration lines must also be swapped. We evaluate the reduction in manual configuration complexity in terms of the total lines of configuration directives, total steps, possible points of where mistakes can be made, and total roles that must be coordinated to acheive the swap. Although we assert that using a constraint solver to derive configurations adds a mental complexity reduction, this cannot be quantified readily and is thus not included in our results.

A key characteristic that we evaluate is the number of possible steps at which a configuration error can occur. With a manual approach, each time a new configuration is

produced, it must be tested to ensure that the configuration file producer has not made any errors, which adds significant overhead. With the Fresh approach, conversely, the injection of each feature into the configuration file can be unit tested. Once it is certified that Fresh correctly injects each feature into the configuration files, therefore, Fresh is guaranteed to produce a correct configuration.

As seen in the inital implementation section of Figure V.9, the base configuration file for BasicSP contains 650 lines of configuration directives. Adding Fresh XML annotation

Initial Implementation	**Lines of Configuration**	**Certifiable**
BasicSP.cdp XML configuration script	650	Y
Fresh XML Annotations	22	Y
Fresh Feature Model Data	8	Y
Fresh Binding Values	28	Y
Total Added Fresh Configuration Lines	58	
% Increase in Lines of Configuration	8.923076923	
Manual Configuration Steps to Use Galileo	**Lines of Configuration**	**Role**
Remove GPS Implementation	7	Component Developer
Remove GPS Alternate OS Implementation	7	Component Developer
Remove GPS Instance	17	Component Assembler
Disconnect GPS from RateGen Pulse	13	Component Assembler
Disconnect GPS from Display Refresh	13	Component Assembler
Disconnect Display from GPS Coordinates	13	Component Assembler
Remove GPS Executor	17	Deployment Planner
Remove GPS Servant	17	Deployment Planner
Remove GPS Stub	6	Deployment Planner
Remove GPS Executor - Alternate OS	17	Deployment Planner
Remove GPS Servant - Alternate OS	17	Deployment Planner
Remove GPS Stub - Alternate OS	6	Deployment Planner
Add Galileo Implementation	7	Component Developer
Add Galileo Alternate OS Implementation	7	Component Developer
Add Galileo Instance	17	Component Assembler
Connect Galileo to RateGen Pulse	13	Component Assembler
Connect Galileo to Display Refresh	13	Component Assembler
Connect Display to Galileo Coordinates	13	Component Assembler
Add Galileo Executor	17	Deployment Planner
Add Galileo Servant	17	Deployment Planner
Add Galileo Galileo Stub	6	Deployment Planner
Add Galileo Executor - Alternate OS	17	Deployment Planner
Add Galileo Servant - Alternate OS	17	Deployment Planner
Add Galileo Stub - Alternate OS	6	Deployment Planner
Total	300	
Fresh Configuration Steps to Use Galileo	**Lines of Configuration**	**Role**
Change Required Features	1	Component Assembler
Invoke Fresh	1	Component Assembler
Total	2	
Reconfiguration Cost of Manual Approach		
Lines of Configuration	300	
Steps	24	
Possible Points of Errors	24	
Roles Involved	3	
Reconfiguration Cost of Fresh Approach		
Lines of Configuration	2	
Steps	2	
Possible Points of Errors	2	
Roles Involved	1	
Fresh Complexity Reduction Summary		
Fresh % Reduction in Configuration Lines	99.33333333	
Fresh % Reduction in Configuration Lines w/ Overhead	80	
Fresh % Reduction in Configuration Steps	91.66666667	
Fresh % Reduction in Possible Error Points	91.66666667	
Fresh % Reduction in Involved Roles	66.66666667	

Figure V.9: Results of Configuring BasicSP with Fresh vs. a Manual Approach

directives, building a simple feature model of BasicSP, and creating values to be injected

into the configuration file by Fresh adds a total of 58 configuration directives. Fresh thus adds ∼8% to the total lines of configuration directives required for BasicSP.

Modifying the BasicSP configuration file to use Galileo requires removing the old GPS implementation, connections, etc. As seen in the "Manual Configuration Steps to Use Galileo" section in Figure V.9, a significant number of steps and lines of configuration directives are involved. At each step in the process, the role modifying the configuration directives can make mistakes and introduce errors.

The "Fresh Configuration Steps to Use Galileo" section in Figure V.9 shows the total lines of configuration directives to reconfigure the BasicSP configuration file with Fresh. Fresh requires the addition of one configuration directive to enable the Galileo feature and the execution of Fresh from the command line to regenerate the BasicSP deployment descriptor.

The "Fresh Complexity Reduction Summary" section in Figure V.9, compares the total manual configuration effort of the manual approach versus the Fresh approach. If the initial overhead of setting up Fresh is included in the calculations, Fresh yields an 80% reduction in the total lines of configuration directives. If the intial overhead is not considered (for cases where the application is configured by a customer), Fresh creates a 99.3% reduction in total lines of configuration directives.

In the manual approach, if component assemblers decide to change to the Galileo component, the component developers and deployment planners must be involved in updating the deployment descriptor. With the Fresh approach, component developers and deployment planners initially encode their expertise into the configuration file as Fresh XML annotations. Thus, each time application assemblers need to swap a component, Fresh uses the XML annotations produced by the other two roles and does not require their involvement. As can be seen in the "Fresh Complexity Reduction Summary" section in Figure V.9, Fresh reduces the total roles involved in the change by two-thirds. Limiting the number of

roles required to implement a change reduces the cost of coordinating the participants and the chances of miscommunication.

Finally, as shown in the “Fresh Complexity Reduction Summary” section in Figure V.9, Fresh reduces the total number of configuration steps that must be performed by 91.67%. Moreover, each eliminated manual configuration step was a potential source of errors in the process, so the overall number of steps where errors can be made are also reduced by 91.67%. Although an intial cost is incurred by adding Fresh configuration directives, it allows for the configuration process to be unit-tested and certified. After the Fresh configuration process is certified correct, there is a large reduction in the potential sources of configuration errors, which are a major contributor to system downtime and failure [50].

CHAPTER VI

AUTOMATED ASPECT CONFIGURATION

Introduction

Developers of complex enterprise applications are faced with the daunting task of managing not only numerous functional concerns, such as ensuring that the application properly executes key business logic, but also meeting challenging non-functional requirements, such as end-to-end response time and security. Enterprise domain solutions have traditionally been developed using large monolithic models that either provide a single view of the system or a limited set of views [63]. The result of using a limited set of views to build the system is that certain concerns are not cleanly separated by the dominant lines of decomposition and are scattered throughout the system's models.

Aspect-Oriented Modeling (AOM) [15, 53, 117] has emerged as a powerful method of untangling and managing scattered concerns in large enterprise application models [59, 65]. With AOM, any scattered concern can be extracted into its own view. For example, caching considerations of an application can be extracted into an aspect. Once caching is separated into its own aspect, the cache sizes and types can be adjusted independently of the application components where the caches are applied. When a final composite solution model for the application is produced, the various aspects are woven back into the solution model and the numerous affected modeling elements are updated to reflect the independently modeled concerns.

Although concerns can often be separated easily into their own aspects or views, it is hard to correctly or optimally merge these concerns back into the solution model. Merging the models is hard because there are typically numerous competing non-functional and functional constraints, such as balancing encryption levels for security against end-to-end performance, that must be balanced against each other without violating domain constraints

(such as maximum available bandwidth). Manual approaches for deriving solutions to these types of constraints do not scale well.

Most current model weavers [25, 51, 65, 117, 136] rely on techniques, such as specifying queries or patterns to match against model elements, that are ideal for matching advice against methods and constructors in application code, but are not necessarily ideal for static weaving problems. Many enterprise applications require developers to incorporate global constraints into the weaving process that can only be solved in a static weaving problem. The techniques used to match against dynamic joinpoints, such as pattern matching, cannot capture global constraints, such as resource constraints (*e.g.*, total RAM consumed < available RAM), that are common in enterprise applications. Because global constraints are not honored by the model weaver, developers are forced to expend significant effort manually deriving weaving solutions that honor them.

When weavers cannot handle global constraints, optimization, or dependency-based constraints, traditional model weaving becomes a manual four stage process, as shown in Figure VI.1. The left-hand column shows the steps involved in model weaving problems

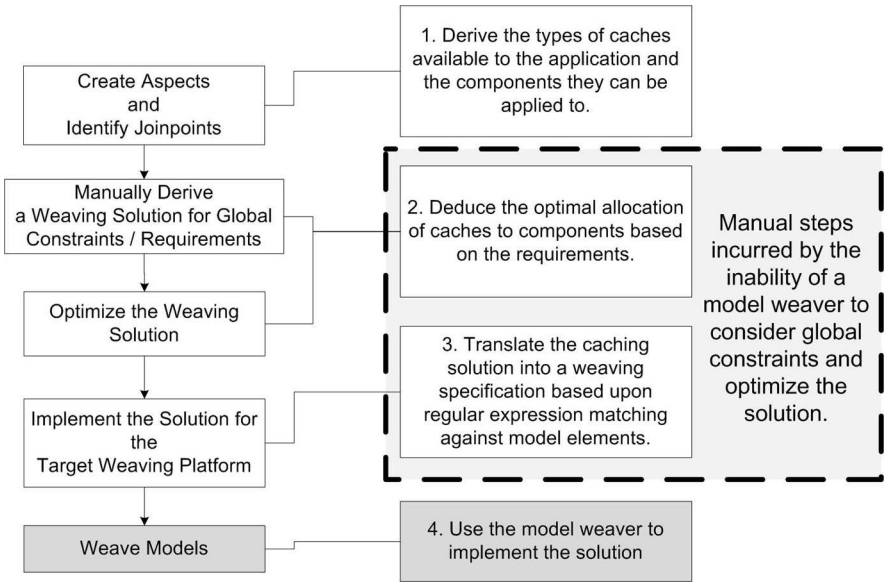


Figure VI.1: The Model Weaving Process

with global constraints in general. The right-hand column shows how these steps manifest themselves in the cache weaving example. First, the advice and joinpoint elements (*e.g.*, caches and components) available in the solution model are identified in step 1. Second, as shown in steps 2 and 3, because a weaver cannot handle global constraints or optimization, developers manually determine which advice elements should be matched to which model elements (*e.g.*, the cache types, cache sizes, and the components to apply the caches to). This second step requires substantial effort because it involves deriving a solution to a complex set of global constraints.

In terms of deriving cache placements in an enterprise application, the second step involves determining cache architectures that fit within the required memory budget and respect the numerous dependency and exclusion constraints between caches. After viable cache architectures are identified, a developer must use the expected request distribution patterns and queueing theory to predict the optimal cache architecture. As the examples show, even for a small set of caches and potential cache locations, the cache placement process requires significant work.

In the third step, developers take this manually-derived solution and translate it into pointcut definitions that match against model elements using regular expressions or queries (*e.g.*, a specification of how to insert the caching model elements into the models to implement the caching architecture). In some cases, the manually derived solution needs to be translated into the pointcut specification languages of multiple model weavers so that the architecture can be implemented in a set of heterogeneous models spanning multiple modeling tools. The model weavers then take these final specifications and merge the models. Each time the underlying solution models change (*e.g.*, the available memory for caching changes), the global constraints can cause the entire solution to change (*e.g.*, the previously used caches no longer fit in the budgeted memory) and the entire three steps must be repeated from scratch.

This chapter shows that the manual steps of deriving a weaving solution that meets the

global application requirements (steps 2 and 3) can be automated in many cases by creating a weaver capable of handling global constraints and optimization. Creating a weaver that can honor these constraints and optimize weaving allows developers to translate the high-level application requirements into pointcut specifications and optimization goals that can be used by the weaver when producing a weaving solution. Finally, because the weaver is responsible for deducing a weaving solution that meets the overall application requirements, as the individual solution models change, the weaver can automatically update the global weaving solution and re-implement it on behalf of the developer for multiple model weaving platforms.

This chapter shows how model weaving can be mapped to a constraint satisfaction problem (CSP) [40, 99, 134]. With a CSP formulation of a model weaving problem, a constraint solver can be used to derive a correct—and in some cases optimal—weaving solution. Using a constraint solver to derive a correct weaving solution provides the following key benefits to model weaving:

- It ensures that the solution is correct with respect to the various modeled functional and non-functional weaving constraints.
- A constraint solver can honor global constraints when producing a solution and not just local regular expression or query-based constraints.
- A constraint solver automates the deduction of the correct weaving and saves considerable manual solution derivation effort.
- The weaving solution can automatically be updated by the solver when the core solution models (and hence joinpoints) change.
- The solver can produce a platform-independent weaving solution (a symbolic weaving solution that is not coupled to any specific pointcut language) where model transformations [24,47] are applied to create a weaving solution for each required weaving platform and

- The solver can derive an optimal weaving solution (with respect to a cost function) in many cases.

Case Study: The Java Pet Store

This chapter uses a case study based on Sun's Java Pet Store [100] multi-tiered e-commerce application. The Pet Store is a canonical e-commerce application for selling pets. Customers can create accounts, browse the Pet Store's product categories, products, and individual product items (*e.g.*, male adult Bulldog vs. female adult Bulldog).

The Pet Store application was implemented by Sun to showcase the capabilities of the various Java 2 Enterprise Edition frameworks [132]. The Pet Store has since been re-implemented or modified by multiple parties, including Microsoft (the .NET Pet Store) [8] and the Java Spring Framework [10]. The Spring Framework's version of the Pet Store includes support for aspects via AspectJ [1] and Spring Interceptors and is hence the implementation that we base our study on.

Middle-tier Caching in the Pet Store

Our case study focuses on implementing caching in the middle-tier (*i.e.*, the persistent data access layer) of the Pet Store through caching aspects. The business logic and views in the Pet Store are relatively simple and thus the retrieval and storage of persistent data is the major performance bottleneck. In performance tests that we ran on the Pet Store using Apache JMeter [56], the average response time across 3,000 requests for viewing the product categories was 3 times greater for a remotely hosted database versus a remotely hosted database with a local data cache (25% hit rate). The same tests also showed that caching reduced the worst case response time for viewing product categories by a factor of two.

Our experiments tested only a single middle-tier and back-end configuration of the Pet Store. Many different configurations are possible. The Spring Pet Store can use a single

database for product and order data or separate databases. Data access objects (DAOs) are provided for four different database vendors. Choosing the correct way of weaving caches into the middle-tier of the Pet Store requires considering the following factors:

- The workload characteristics or distributions of request types, which determine what data is most beneficial to cache [92]. For example, keeping the product information in the cache that is most frequently requested will be most beneficial.

- The architecture of the back-end database servers providing product, account, and order data to the application determines the cost of a query [90]. For example, in a simple Pet Store deployment where the back-end database is co-located with the Pet Store's application server, queries will be less expensive than in an arrangement where queries must be sent across a network to the database server.

- The hardware hosting the cache and the applications co-located with it will determine the amount of memory available for caching product data. If the Pet Store is deployed on small commodity servers with limited memory, large caches may be undesirable.

- The number of possible cache keys and sizes of the data associated with each cache item will influence the expected cache hit rate and the penalty for having to transfer a data set across the network from the database to the application server [102]. For example, product categories with large numbers of products will be more expensive to serialize and transfer from the database than the information on a single product item.

- The frequency that the data associated with the various middle-tier DAOs is updated and the importance of up-to-date information will affect which items can be cached and any required cache coherence schemes [102]. For example, product item availability is likely to change frequently, making product items less suitable to cache than product categories that are unlikely to change.

Modeling and Integrating Caches into the Pet Store

Aspect modeling can be used effectively to weave caches into the Pet Store to adapt it for changing request distribution patterns and back-end database configurations. We used this scenario for our case study to show that although caches can be woven into code and models to adapt the Pet Store for a new environment, creating and maintaining a cache weaving solution that satisfies the Pet Store's global application requirements takes significant manual effort due to the inability of model weavers to encode and automate weaving with the global application constraints. Each time the global application requirements change, the manually deduced global cache weaving solution must be updated. Updating the global cache weaving solution involves a number of models and tools. Figure VI.2 shows the various models, code artifacts, and tools involved in implementing caching in the Pet Store.

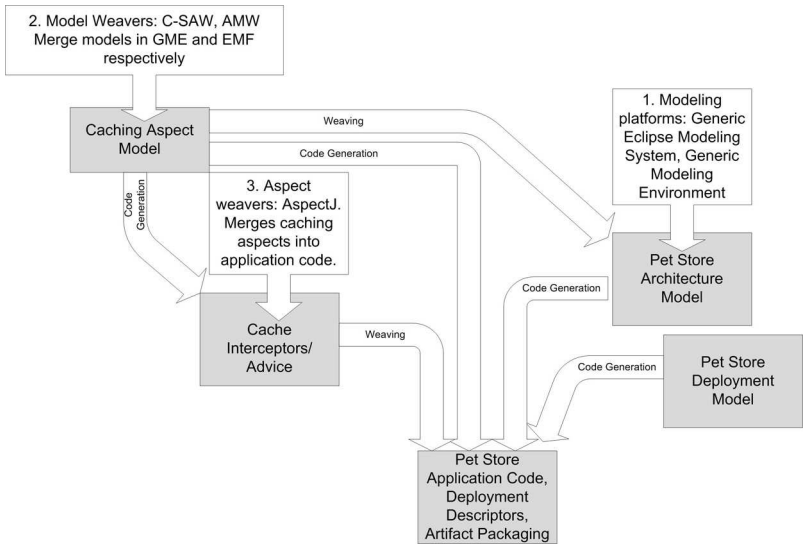


Figure VI.2: Models and Tools Involved in the Pet Store

1. Modeling platforms. We have implemented models of different parts of the Pet Store in two different modeling tools: the Generic Eclipse Modeling System (GEMS) [160] and the Generic Modeling Environment (GME) [87]. GME was chosen due to its extensive support for different views, while GEMS was selected for its strengths in *model intelligence*,

which was used for automating parts of the deployment modeling process. Using different tools simplifies the derivation of the deployment plan and the understanding of the system architecture but also requires some level of integration between the tools.

GEMS is a graphical modeling tool built on top of Eclipse [127] and the Eclipse Modeling Framework (EMF) [29]. GEMS allows developers to use a Visio-like graphical interface to specify metamodels and generate domain-specific modeling language (DSML) tools for Eclipse. In GEMS, a deployment modeling tool has been implemented to capture the various deployment artifacts, such as required Java Archive Resources (JAR) files, and their placement on application servers. Another Neat Tool (ANT) [72] build, configuration, and deployment scripts can be generated from the GEMS deployment model.

GME [87] is another graphical modeling tool similar to GEMS that allows developers to graphically specify a metamodel and generate a DSML editor. A modeling tool for specifying the overall component architecture of the Pet Store has been implemented in GME. The GME architecture model is used to capture the component types, the various client types, back-end database architecture, and expected distribution of client requests to the Pet Store. The GME architecture model is shown in Figure VI.3.

2. Model weaving tools. The caching aspect of the Pet Store is modeled separately from the GEMS deployment model and GME architecture model. Each time the caching model is updated, model weaving tools must be used to apply the new caching architecture to the GEMS and GME models. For the GME models, the C-SAW [130] model weaver is used to merge the caching architecture into the architecture model. C-SAW relies on a series of weaving definition files to perform the merger. Each manually derived global cache weaving solution is implemented in C-SAW's weaving definition files to apply to the GME architecture models. Again, because we need two separate modeling tools to produce the best possible deployment and architecture models, we must also utilize and integrate two separate model weavers into the development process.

The deployment models in GEMS need to be updated via a model weaver, such as the

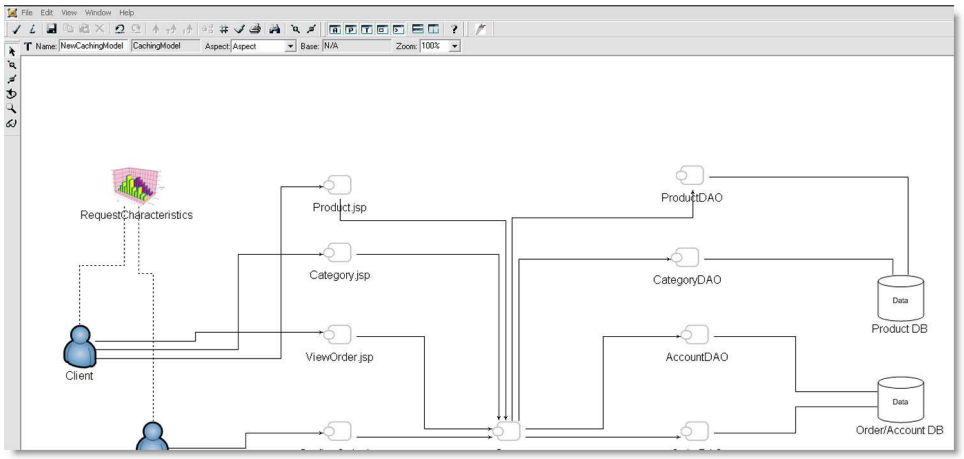


Figure VI.3: GME Pet Store Architecture Model

Atlas Model Weaver (AMW) [51], which can interoperate with models based on EMF. With AMW, developers specify two EMF models and a series of merger directives (*i.e.*, a weaving specification). AMW produces a third merged EMF model from the two source models. Each global cache weaving solution must also be implemented as a weaving specification for AMW. Once the AMW specification is implemented, the cache weaving solution can be merged into the GEMS EMF-based deployment model to include any required JAR files and cache configuration steps.

3. Code weaving tools. Finally, to apply the cache weaving solution to the legacy Pet Store code, the Java cache advice implementations must be woven into the Pet Store's middle-tier objects using AspectJ [1], which is a framework for weaving advice into Java applications. Although the Spring framework allows the application of AspectJ advice definitions to the Pet Store, it requires that the Spring bean definition files for the Pet Store be updated to include the new AspectJ pointcuts and advice specifications. A final third implementation of the global cache weaving solution must be created and specified in terms of Spring bean definitions and AspectJ pointcuts.

Overall, there are three separate tool chains that the Pet Store cache weaving solution must be implemented in. First, C-SAW weaving specifications must be created to update the GME architectural models. Second, AMW weaving specifications must be produced to

update the GEMS deployment models. Finally, the weaving solution must be turned into AspectJ advice/pointcut definitions for weaving the caches into the Pet Store at runtime.

Model Weaving Challenges

One of the primary limitations of applying existing model weavers to the Pet Store case study is that existing model weaver pointcut specifications cannot encode global application constraints, such as memory consumption constraints, and also cannot leverage global constraints or dependency-based weaving rules to produce an overall global weaving solution. Developers must instead document and derive a solution for the overall global application constraints and implement the solution for each of the numerous modeling and weaving platforms for the Pet Store. Moreover, each time the underlying global application constraints change (*e.g.*, the memory available for caches is adjusted) the overall global weaving solution must be recalculated and implemented in the numerous modeling tools and platforms.

Differences Between Aspect Weavers and Model Weavers

To understand why model weavers do not currently support global constraints and how this can be rectified, we first must evaluate aspect weavers at the coding level, which have influenced model weavers. Aspect weavers, such as AspectJ and HyperJ [7], face an indeterminate number of potential joinpoints (also referred to as *joinpoint shadows* [71]) that will be passed through during application execution. For example, late-binding can be used in a Java application to dynamically load and link in multiple libraries for different parts of the application.

Each library may have hundreds or thousands of classes and numerous methods per class (each a potential joinpoint). An aspect weaver cannot know which classes and methods the execution path of the application will pass through before the process exits. The weaver can therefore never ascertain the exact set of potential joinpoints that will be used

ahead of time. Although the weaver may have knowledge of every joinpoint shadow, it will not have knowledge of which are actually used at runtime. Model weaving, however, faces a different situation than a runtime aspect weaver. The key differences are:

- Model weaving merges two models of finite and known size.
- Because models have no thread of execution, the weaver can ascertain exactly what joinpoints are used by each model.
- Model weaving speed is less critical than aspect weaving speed at runtime and adding additional seconds to the total weaving time is not unreasonable.

Because a model weaver has knowledge of the entire set of joinpoints used by the models at its disposal it can perform a number of activities that are not possible with runtime weaving where the entire used set of target joinpoints is not known. For example, a model weaver can incorporate global constraints into the weaving process. A runtime weaver cannot honor global constraints because it cannot see the entire used joinpoint set at once. To honor a global constraint, the weaver must be able to see the entire target joinpoint set to avoid violating a global constraint.

Runtime aspect weaving involves a large number of potential joinpoints or joinpoint shadows and is not well-suited for capturing and solving global application constraints as part of the weaving process. When weaving must be performed on an extremely large set of target joinpoints, the weaver must use a high-efficiency technique for matching advice to joinpoints (every millisecond counts). The most common technique is to use a query or regular expression that can be used to determine if a pointcut matches a joinpoint. The queries and regular expressions are independent of each other, which allows the weaver to quickly compare each pointcut to the potential joinpoints and determine matches.

If dependencies were introduced between the queries or expressions (*e.g.*, only match pointcut A if pointcut B or C do not match), the weaver would be forced to perform far less efficient matching algorithms. Moreover, since the weaver could not know the entire

joinpoint set passed through by the application's execution thread ahead of time, it could not honor a dependency, such as match pointcut A only if pointcuts B and C are *never* matched, because it cannot predict whether or not B and C will match in the future. Finally, when dependencies are introduced, there is no longer necessarily a single correct solution. Situations can arise where the weaver must either choose to apply A or to apply B and C.

Challenge 1: Existing Model Weaving Poinctut Specifications Cannot Encode Global Application Constraints

Most model weavers, such as C-SAW, AMW, and the Motorola WEAVR [43], have adopted the approach of runtime weavers and do not allow dependencies between pointcuts or global constraints. Because the model weaver does not incorporate these types of constraints, developers cannot encode the global application constraints into the weaving specification. Figure VI.4 presents the manual refactoring steps (the first six steps) that

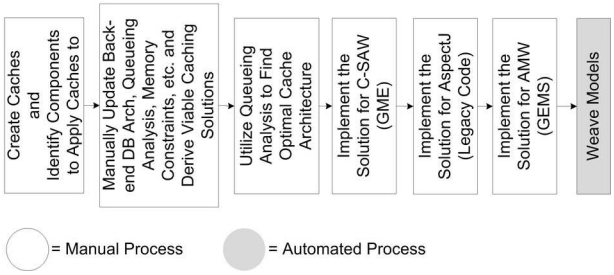


Figure VI.4: Solution Model Changes Cause Weaving Solution Updates

must be performed when the modeled distribution of request types to the Pet Store changes.

In the Pet Store case study, there are a number of dependencies and global constraints that must be honored to find a correct weaving. We created caching advice implementations that capture all product queries and implementations that are biased towards specific data items, such as the `FishCache`. The biased cache is used when the majority of requests are for a particular product type. The `FishCache` and the generic product cache should be mutually exclusive. The use of the `FishCache` is excluded if the percentage of requests

for fish drops below 50%. Moreover, the generic product cache will then become applicable and must be applied.

A small change in the solution model can cause numerous significant ripple effects in the global application constraints and hence weaving solution. This problem of changes to the solution models of an applicaiton causing substantial refactoring of the weaving solution is well-known [66]. The problem becomes even more complex, however, with the global weaving solution where significant refactoring causes multiple implementations of the weaving specification to change.

The problem with managing this ripple effect with existing model weavers is that both the `FishCache` and the generic product cache have a pointcut that matches the same model element, the `ProductDAO`. With existing pointcut languages based on regular expressions or queries, there is no way to specify that only one of the two pointcut definitions should be matched to the `ProductDAO`. The pointcut definitions only allow the developer to specify matching conditions based on joinpoint properties and not on the matching success of other pointcuts.

Developers often need to restrict the overall cache selection to use less than a specified amount of memory. For example, rather than having the `FishCache` and `GenericCache` be mutually exclusive, the two caches could be allowed to be applied if there is sufficient memory available to support both. Requiring that the woven caches fit within a memory budget is a resource constraint on the total memory consumed by the weaving solution and relies on specifying a property over the entire weaving solution. Existing regular expression and query-based pointcut languages usually do not capture these types of rules.

Another challenge of producing this weaving constraint on the memory consumed by the caches is that it relies on properties of both the advice objects (*e.g.*, the memory consumed by the cache) and the joinpoint objects (*e.g.*, the memory available to the hosting object's application server). Most model weaving pointcut languages allow specifying conditions only against the properties of the target joinpoints and not over the advice elements

associated with the pointcut. To circumvent this limitation, developers must manually add up the memory consumed by the advice associated with the pointcut and encode it into the pointcut specification's query (*e.g.,* find all elements hosted by an application server with at least 30 MB of memory).

Challenge 2: Changes to the Solution Model Can Require Significant Refactoring of the Weaving Solution

As the solution models of the application that determine the set of joinpoints change, each manual step in Figure VI.4 may need to be repeated. The caching solution relies on multiple solution models, such as the server request distribution model, the cache hit ratio and service times model, and the PetStore software architecture model. A change in any of these models can trigger a recalculation of the global weaving solution. Each recalculation of the global weaving solution involves multiple complex caculations to determine the new targets for caches. After the new cache targets are identified, the implementation of the solution for each weaving platform, such as the C-SAW weaving definition files, must be updated to reflect the new caching architecture.

For example, the correct weaving of caches into the Pet Store requires considering the back-end organization of the product database. If the database is hosted on a separate server from the Pet Store's application server, caching product information can significantly improve performance. The cache weaving solution is no longer correct, however, if biased caches are applied to various product types that are being retrieved from a remote database and the database is co-hosted with the Pet Store's application server. A developer must then update the weaving solution to produce a new and correct solution for the updated solution model.

As seen in Figure VI.5, not only are numerous manual steps required to update the weaving solution when solution model changes occur, but each manual step can be complex. For example, re-caculating the optimal placement of caches using a queueing model

is non-trivial. Moreover, each manual step in the process is a potential source of errors that can produce incorrect solutions and require repeating the process. The large numbers of solution model changes that occur in enterprise development and the complexity of updating the weaving solution to respect global constraints, make manually updating a global weaving solution hard.

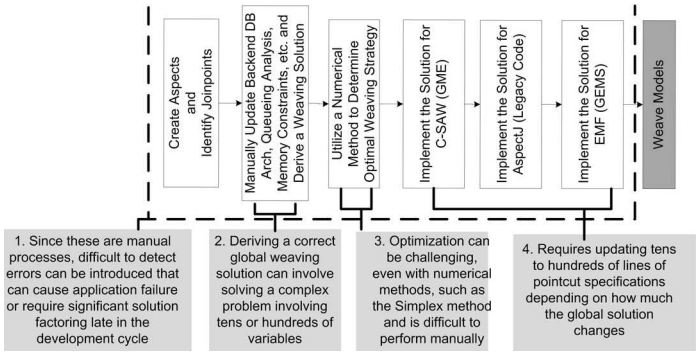


Figure VI.5: Challenges of Updating a Weaving Solution

Challenge 3: Existing Model Weavers Cannot Leverage a Weaving Goal to Find an Optimal Concern Merging Solution

Another challenge of encoding global application constraints into a weaving specification is that global constraints create situations where there are multiple correct solutions. Existing model weavers do not allow situations where there are multiple possible weaving solutions. Because the weaver cannot choose between weaving solutions, developers must manually deduce the correct and optimal solution to use.

Optimizing a solution bound by a set of global constraints is a computationally intensive search process. Searching for an optimal solution involves exploring the solution space (the set of solutions that adhere to the global constraints) to determine the optimal solution. This type of optimization search can sometimes be performed manually with numerical methods, such as the Simplex [109] method, but is typically hard. In particular, each time

the solution models change, developers must manually derive a new optimal solution from scratch.

For example, to optimize the allocation of caches to DAOs in the Pet Store, developers must:

- Evaluate the back-end database configuration to determine if product, account, or other data must be cached to reduce query latency.
- Derive from the cache deployment constraints what caches can be applied to the system and in what combinations.
- Determine how much memory is available to the caches and how memory constraints restrict potential cache configurations.
- Exhaustively compare feasible caching architectures using queuing analysis to derive the optimal allocation of caches to DAOs based on DAO service rates with and without caching and with various cache hit rates.

It is hard to manually perform these complex calculations each time the solution models change or caching constraints are modified.

CSP-based Model Weaving

To address the challenges described earlier, we have developed *AspectScatter*, which is a static model weaver that can:

1. Transform a model weaving problem into a CSP and incorporate global constraints and dependencies between pointcuts to address Challenge 1.
2. Using a constraint solver, automatically derive a weaving solution that is correct with respect to a set of global constraints, eliminating the need to manually update the weaving solution when solution models change, as described in Challenge 2.

3. Select an optimal weaving solution (when multiple solutions exist) with regard to a function over the properties of the advice and joinpoints, allowing the weaver rather than the developer to perform optimization, thereby addressing Challenge 3 from Section VI.

4. Produce a platform-independent weaving model and transform it into multiple platform-specific weaving solutions for AspectJ, C-SAW, and AMW through model transformations, thus addressing the problems associated with maintaining the weaving specification in multiple weaving platforms.

Figure VI.6 shows an overview of AspectScatter's weaving approach. In Step 1, de-

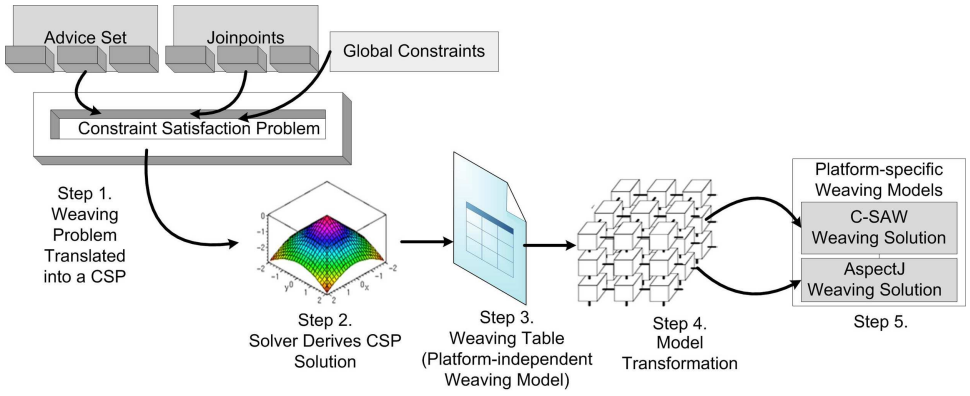


Figure VI.6: Constraint-based Weaving Overview

velopers describe the advice, joinpoints, and weaving constraints to AspectScatter using its domain-specific language (DSL) for specifying aspect weaving problems with global constraints. In Step 2, AspectScatter transforms the DSL instance into a CSP and uses a constraint solver to derive a guaranteed correct and, if needed, optimal weaving solution. In Step 3, AspectScatter transforms the solution into a platform-independent weaving model. Finally, in Step 4, model transformations are used to transform the platform-independent weaving model into specific implementations, such as C-SAW weaving definition files, for each target weaving platform.

The remainder of this section presents a mapping from model weaving to a CSP. By producing a CSP for model weaving, a constraint solver can be used to deduce a correct and in many cases optimal solution to a weaving problem.

CSP Background

A CSP is a set of variables and a set of constraints over those variables. For example, $A < B < 100$ is a CSP over the integer variables A and B. A solution to a CSP is a set of values for the variables (called a labeling) that adheres to the set of constraints. For example, $A = 10, B = 50$ is a valid labeling (solution) of the example CSP.

Solutions to CSPs are obtained by using *constraint solvers*, which are automated tools for finding CSP solutions. Constraint solvers build a graph of the variables and constraints and apply techniques, such as arc-consistency, to find the ranges that variable values can be set to. Search algorithms then traverse the constraint network to hone in on a valid or optimal solution.

A constraint solver can also be used to derive a labeling of a CSP that maximizes or minimizes a specific goal function (*i.e.*, a function over the variables). For example, the solver could be asked to maximize the goal function $A + B$ in our example CSP. A maximal labeling of the variables with respect to this goal function would be $A = 98, B = 99$.

Mapping Cache Weaving to a CSP

Cache weaving can be used as a simple example of how a CSP can be used to solve a weaving problem. In the following example, we make several assumptions, such as the hit ratio for the caches being the same for both joinpoints, to simplify the problem for clarity. Real weaving examples involving optimal caching or other types of global constraints are substantially more difficult to solve manually and hence motivate our constraint solver weaving solution.

Assume that there are two caches that can be woven into an application, denoted $C1$

and $C2$. Furthermore, assume that there are two joinpoints that the caches can be applied to, denoted $J1$ and $J2$. Let there be a total of 200K of memory available to the caches. Furthermore, the two caches are mutually exclusive and cannot be applied to the same joinpoint. Let the time required to service a request at $J1$ be 10ms and the time at $J2$ be 12ms.

Each cache hit on $C1$ requires 2ms to service and each cache hit on $C2$ requires 3ms. All requests pass through both $J1$ and $J2$ and the goal is to optimally match the caches to joinpoints and set their sizes to minimize the total service time per request. The size of each cache, $C1_{size}$ and $C2_{size}$, determines the cache's hit ratio. For $C1$ the hit ratio is $C1_{size}/500$ and for $C2$ the hit ratio is $C2_{size}/700$. Let's assume that cache $C1$ is woven into joinpoint $J1$ and $C2$ is woven into joinpoint $J2$, the service time per request can be calculated as

$$SvcTime = 2(C1_{size}/500) + 10(1 - C1_{size}/500) + 3(C1_{size}/700) + 12(1 - C1_{size}/700)$$

With this formulation, we can derive the optimal sizes for the caches subject to the global weaving constraint:

$$C1_{size} + C2_{size} < 200$$

The problem, however, is that we want to know not only the optimal cache size but also where to weave the caches and the above formulation assumes that cache $C1$ is assigned to $J1$ and $C2$ to $J2$. Thus, instead we need to introduce variables into the service time calculation to represent the joinpoint that each cache is actually applied to so that we do not assume an architecture of how caches are applied to joinpoints. That is, we want to deduce not only the cache sizes but also the best allocation of caches to joinpoints (the caching architecture). Let the variable M_{jk} have value 1 if the j_{th} cache C_j is matched to

joinpoint J_k and 0 otherwise. We can update our service time formula so that it does not include a fixed assignment of caches to joinpoints:

$$SvcTime = 2(M_{11} * C1_{size}/500) + 3(M_{21} * C2_{size}/700) +$$

$$10(1 - ((M_{11} * C1_{size}/500) + (M_{21} * C2_{size}/700))) +$$

$$2(M_{12} * C1_{size}/500) + 3(M_{22} * C2_{size}/700) +$$

$$12(1 - ((M_{12} * C1_{size}/500) + (M_{22} * C2_{size}/700)))$$

The new formulation of the response time takes into account the different caches that could be deployed at each joinpoint. For example, the service time at joinpoint $J1$ is defined as:

$$J1SvcTime = 2(M_{11} * C1_{size}/500)$$

$$+3(M_{21} * C2_{size}/700) +$$

$$+10(1 - ((M_{11} * C1_{size}/500) + (M_{21} * C2_{size}/500)))$$

In this formulation the variables M_{11} and M_{21} are influencing the service time calcuation by determining if a specific cache's servicing information is included in the calculation. If the cache $C1$ is applied to $J1$, then $M_{11} = 1$ and the cache's service time is included in the calculation. If the cache is not woven into $J1$, then $M_{11} = 0$, which zeros out the effect of the cache at $J1$ since:

$$J1SvcTime = 2(0) \ldots 10(1 - (0 + (M_{21} * C2_{size}/500)))$$

Thus, by calculating the optimal values of the M_{ij} variables, we are also calculating the optimal way of assigning the caches (advice) to the joinpoints.

To optimally weave the caches into the application, we need to derive a set of values for the variables in the service time equation that minimizes its value. Furthermore, we must derive a solution that not only minimizes the above equation's value but respects the constraints:

$$C1_{size} + C2_{size} < 200$$

$$(M_{11} = 1) \Rightarrow (M_{21} = 0)$$

$$(M_{21} = 1) \Rightarrow (M_{22} = 0)$$

because the cache sizes must add up to less than the alloted memory (200K) and both caches cannot be applied to the same joinpoint.

When the constraint solver is invoked on the CSP, the output will be the values for the M_{ij} variables. That is, for each Advice, i, and joinpoint, j, the solver will output the value of the variable M_{ij}, which specifies if Advice, A_i, should be mapped to joinpoint, B_j. The M_{ij} variables can be viewed as a table where the rows represent the advice elements, the columns represent the joinpoints, and the values (0 or 1) at each cell are the solver's solution as to whether or not a particular advice should be applied to a specific joinpoint. Furthermore, any variables that do not have values set, such as the cache sizes ($C1_{size}$ and $C2_{size}$), will have optimal values set by the constraint solver.

Even for this seemingly simple weaving problem, deriving what joinpoints the caches should be applied to and how big each cache should be is not easy to do manually. However, by creating this formulation of the weaving problems as a CSP, we can use a constraint solver to derive the optimal solution on our behalf. The solution that the solver creates will include not only the optimal cache sizes, but also which joinpoints each cache should be applied to, which would be very difficult to derive manually.

A General Mapping of Weaving to a CSP

Previously, we showed how a CSP could be used to solve a weaving problem involving optimization and global constraints. This section presents a generalized mapping from a weaving problem to a CSP so that the technique can be applied to arbitrary model weaving problems with global constraints.

We define a solution to a model weaving problem as a mapping of elements from an advice set α to a joinpoint set β that adheres to a set of constraints γ. To represent this mapping as a CSP, we create a table—called the *weaving table*—where for each advice A_i in α and joinpoint B_j in β, we define a cell (*i.e.*, a variable in the CSP) M_{ij}. If the advice A_i should be applied to the joinpoint B_j, then $M_{ij} = 1$ (meaning the table cell <i,j> has value 1). If A_i should not be applied to B_j, then $M_{ij} = 0$. The rules for building a weaving solution are described to the constraint solver as constraints over the M_{ij} variables. An example weaving table where the `ProductsCache` is applied to the `ProductDAO` is shown in Table 1.

	ProductDAO	ItemDAO
ProductsCache	$M_{00} = 1$	$M_{01} = 0$
FishCache	$M_{10} = 0$	$M_{11} = 0$

Table VI.1: An Example Weaving Table

Some weaving constraints are described purely in terms of the weaving table. For example, Challenge 1 introduced the constraint that the `FishCache` should only be used if the `ProductsCache` is not applied to any component. This constraint can be defined in terms of a constraint over the weaving table. If the `FishCache` is A_0 and the `ProductsCache` is A_1, then we can encode this constraint as for all joinpoints, j:

$$(\sum_{j=0}^{n} M_{0j} > 0) \rightarrow (\sum_{j=0}^{n} M_{1j} = 0)$$

Some examples of dependency constraints between advice elements that can be implemented as CSP constraints on the weaving table are:

$Advice_0$ requires $Advice_1$ to always be applied to the same joinpoint:

$$\forall B_j \subset \beta, (M_{0j} = 1) \rightarrow (M_{1j} = 1)$$

$Advice_0$ excludes $Advice_1$ from being applied to the same joinpoint:

$$\forall B_j \subset \beta, (M_{0j} = 1) \rightarrow (M_{1j} = 0)$$

$Advice_0$ requires between $MIN \ldots MAX$ of $Advice_1 \ldots Advice_k$ at the same joinpoint:

$$\forall B_j \subset \beta, (M_{0j} = 1) \rightarrow (\sum_{i=1}^{k} M_{ij} \geq MIN) \wedge (\sum_{i=1}^{k} M_{ij} \leq MAX)$$

Advice and Joinpoint Properties Tables

Other weaving constraints must take into account the properties of the advice and joinpoint elements and cannot be defined purely in terms of the weaving table. To incorporate constraints involving the properties of the advice and joinpoints, we create two additional tables: the *advice properties table* and *joinpoint properties table*. Each row P_i in the advice properties table represents the properties of the advice element A_i. The columns of the advice table represent the different property types. Thus, the cell <i,j>, represented by the variable PA_{ij}, contains A_i's value for the property associated with the j_{th} column. The joinpoint properties table is constructed in the same fashion with the rows being the joinpoints (*i.e.*, each cell is denoted by the variable PT_{ij}). An example joinpoint properties table is shown in Table 2.

Challenge 1 introduced the constraint that the `FishCache` should only be applied to the `ProductDAO` if more than 50% (the majority) of the requests to the `ProductDAO` are for fish. We can use the advice and joinpoint properties tables to encode this request

	%Fish Requests	%Bird Requests
ProductDAO	65% ($PT_{00} = 0.65$)	20% ($PT_{01} = 0.2$)
ItemDAO	17% ($PT_{10} = 0.17$)	47% ($PT_{11} = 0.47$)

Table VI.2: An Example Joinpoint Properties Table

distribution constraint. Let the joinpoint properties table column at index 0 be associated with the property for the percentage of requests that are for Fish, as shown in the the joinpoint properties table shown in Table 2. Moreover, let A_1 be the `FishCache` and B_0 be the `ProductDAO`. The example request distribution constraint can be encoded as $M_{10} \rightarrow (PT_{00} > 50)$.

Global Constraints

In enterprise systems, global constraints are often needed to limit the amount of memory, bandwidth, or CPU consumed by a weaving solution. Global constraints can naturally be incorporated into the CSP model as constraints involving the entire set of variables in the weaving table. For example, the memory constraint on the total amount of RAM consumed by the caches, described in Challenge 1, can be specified as a constraint on the weaving and properties tables.

	...	RAM on Application Server
ProductDAO	...	1024K ($PT_{05} = 1024$)
...	...	...

Table VI.3: An Example Joinpoint Properties Table with Available Memory

	...	RAM Consumed
ProductCache	...	400K ($PA_{04} = 400$)
FishCache	...	700K ($PA_{14} = 700$)

Table VI.4: An Example Advice Properties Table with RAM Consumption

Let the joinpoint property table column at index 5, as shown in Table 3, represent the amount of free memory available on the hosting application server of each joinpoint. Moreover, let the advice property table column at index 4, as shown in Table 4, contain the amount of memory consumed by each cache. The memory consumption constraint can be specified as:

$$\forall B_j \subset \beta, (\sum_{i=0}^{n} PA_{i4} * M_{ij}) < PT_{j5}$$

If an advice element is matched against a joinpoint, the corresponding M_{ij} variable is set to 1 and the advice element's memory consumption value, PA_{i4}, is added to the total consumed memory on the target application server. The constraint that the consumed memory be less than the available memory is captured by the stipulation that this sum be $< PT_{j5}$, which is the total amount of free memory available on the joinpoint's application server.

Joinpoint Feasibility Filtering with Regular Expressions and Queries

Some types of constraints, such as constraints that require matching strings against regular expressions, are more naturally represented using existing query and regular expression techniques. The CSP approach to model weaving can also incorporate these types of constraint expressions. Regular expressions, queries, and other pointcut expressions that do not have dependenices can be used as an initial filtering step to explicitly set zero values for some M_{ij} variables. The filtering step reduces the set of feasible joinpoints that the solver must consider when producing a weaving solution.

For example, the `FishCache` should only be applied to DAOs with the naming convention "Product*". This rule can be captured with an existing pointcut language and then checked against all possible joinpoints, as shown in Figure VI.7. For each joinpoint, j, that the pointcut does not match, the CSP variable, M_{ij}, for each advice element, i, associated with the pointcut is set to 0. Layering existing dependency-free pointcut languages as filters on top of the CSP based weaver can help to increase the number of labeled variables provided to the solver and thus reduce solving time.

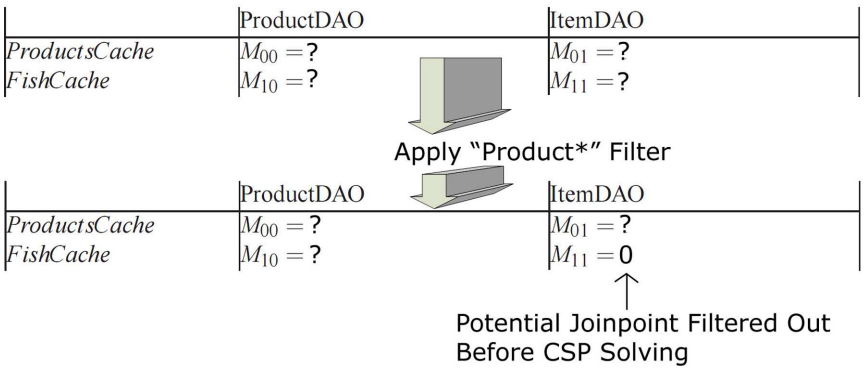


Figure VI.7: Joinpoint Feasibility Filtering

CSP-Weaving Benefits

Challenge 3 showed the need for the ability to incorporate a weaving goal to produce an optimal weaving. Using a CSP model of a weaving problem, a weaving goal can be specified as a function over the M_{ij}, PA_{ij}, and PT_{ij} variables. Once the goal is defined in terms of these variables, the solver can be used to derive a weaving solution that maximizes the weaving goal. Moreover, the solver can set optimal values for attributes of the advice elements, such as cache size.

Allowing developers to specify optimization goals for the weaver enables different weaving solutions to be obtained that prioritize application concerns differently. For example, the same Pet Store solution models can be used to derive caching solutions that minimize response time at the expense of memory, balance response time and memory consumption, or minimize the response time of particular user actions, such as adding items to the shopping cart. To explore these various solution possibilities, developers update the optimization function provided to AspectScatter and not the entire weaving solution calculation process. With the manual optimization approaches required by existing model weavers, it is typically too time-consuming to evaluate multiple solution alternatives.

Mapping aspect weaving to a CSP and using a constraint solver to derive a weaving solution addresses Challenge 1. CSPs can naturally accomodate both dependency constraints and complex global constraints, such as resource or scheduling constraints. With existing

model weaving approaches developers manually identify and document solutions to the global weaving constraints. With a CSP formulation of weaving, conversely, a constraint solver can perform this task automatically as part of the weaving process.

Manual approaches to create a weaving solution for a set of constraints have numerous points where errors can be introduced. When AspectScatter is used to derive a weaving solution, the correctness of the resulting solution is assured with respect to the weaving constraints. Moreover, in cases where there is no viable solution, AspectScatter will indicate that weaving is not possible.

A further benefit of mapping an aspect weaving problem to a CSP is that extensive prior research on CSPs can be applied to deriving aspect weaving solutions. Existing research includes different approaches to finding solutions [84], incorporating soft constraints [122], selecting optimal solutions or approximations in polynomial time [27, 54, 118], and handling conflicting constraints. Conflict resolution has been singled out in model weaving research as a major challenge [164]. Numerous existing techniques for over-constrainted systems [26,78,80,137] (*i.e.*, CSPs with conflicting constraints), such as using higher-order constraints, can be applied by mapping model weaving to a CSP.

The AspectScatter DSL

Manually translating an aspect weaving problem into a CSP using the mapping presented earlier is not ideal. A CSP model can accomodate global constraints and dependencies but requires a complex mapping that must be performed correctly to produce a valid solution. Working directly with the CSP variables to specify a weaving problem is akin to writing assembly code as opposed to Java or C++ code.

AspectScatter provides a textual DSL for specifying weaving problems and can automatically transform instances of the DSL into the equivalent CSP model for a constraint solver. AspectScatter's DSL allows developers to work at the advice/joinpoint level of abstraction and still leverage a CSP and constraint solver for deriving a weaving solution.

The CSP formulation of an aspect weaving problem is not specific to any one particular type of joinpoint or advice. The construction and solving of the CSP is a mathematical manipulation of symbols representing a set of joinpoints and advice. As such, the joinpoints could potentially be Java method invocations or model elements. Later, we discuss how these symbols are translated into platform-specific joinpoints and advice. For this section, however, it is important to remember that we are only declaring and stating the symbols and constraints that are used to build the mathematical CSP weaving problem.

For example, in the context of the cache weaving example, there are two different types of platform-specific joinpoints. First, there are the joinpoints used by C-SAW, which are types of model elements in a GME model. Second, there are AspectJ type joinpoints, which are the invocation of various methods on the Java implementations of the `ProductDAO`, `OrderDAO`, etc. In the platform-independent model used by the CSP, the joinpoint definition `OrderDAO` is merely a symbolic definition of a joinpoint. When the platform-specific solution is translated into a platform-specific weaving solution, `OrderDAO` is mapped to a model element in the GME model used by C-SAW and an invocation of a query method on the Java implementation of the `OrderDAO`.

The basic format for an AspectScatter DSL instance is shown below:

```
ADVICE_1_ID
{
  (DIRECTIVE;)*
}
...
ADVICE_N_ID
{
  (DIRECTIVE;)*
}
JOINPOINT_1_ID
{
  (VARIABLENAME=EXPRESSION;)*
}
...
JOINPOINT_N_ID
```

```
  {
    (VARIABLENAME=EXPRESSION;)*
  }
```

The JOINPOINT declaration specifies a joinpoint, an element $B_j \subset \beta$, that ADVICE elements can be matched against. The JOINPOINT_ID is the identifier, such as "OrderDAO," that is given as a symbolic name for the joinpoint. Each JOINPOINT element contains one or more property declarations in the form of VARIABLENAME=EXPRESSION. The columns for the joinpoint properties table are created by traversing all of the JOINPOINT declarations and creating columns for the set of VARIABLENAMEs. The EXPRESSION that a JOINPOINT specifies for a VARIABLENAME becomes the entry for that JOINPOINT's row in the specified VARIABLENAME column, PT_{ij}.

Each ADVICE declaration specifies an advice element that can be matched against the set of JOINPOINT elements, an element $A_i \subset \alpha$. The DIRECTIVES within the advice element specify the constraints that must be upheld by the weaving solution produced by AspectScatter and the properties of the ADVICE element (values for the PA_{ij} variables). The directives available in AspectScatter are shown in Table 5.

As an example, the AspectScatter ADVICE definitions:

```
GenericCache
{
  Excludes:FishCache;
  DefineVar:CacheSize;
}
FishCache
{
}
```

defines two advice elements called GenericCache and FishCAche. The DIRECTIVEs within the GenericCache declaration (between "{..}") specify the constraints that must be upheld by the joinpoint it is associated with and the properties the advice element defines. The GenericCache excludes the advice element FishCache from being applied

DIRECTIVE	Applied To	Description
Requires : *ADVICE*+	one or more other ADVICE elements	Ensures that all of the specified ADVICE elements are applied to a JOINPOINT if the enclosing ADVICE element is
Required : (*true*|*false*)	an ADVICE element	The enclosing ADVICE element must be applied to at least one JOINPOINT (if true).
Excludes : *ADVICE*+	one or more other ADVICE elements	Ensures that none of the specified ADVICE are applied to the same JOINPOINT as the enclosing ADVICE
Select : [*MIN*..*MAX*],*ADVICE*+	a cardinality expression and one or more other ADVICE	Ensures that at least MIN and at most MAX of the specified ADVICE are mapped to the same JOINPOINT as the enclosing ADVICE
Target : *CONSTRAINT*	an ADVICE element	Requires that CONSTRAINT hold true for the ADVICE and JOINPOINT's properties if the ADVICE is mapped to the JOINPOINT
Evaluate : (*ocl*|*groovy*), *FILTER_EXPRESSION*	an ADVICE element	Requires that FILTER_EXPRESSION defined in OCL or Groovy hold true for the ADVICE and JOINPOINT's properties if the ADVICE is mapped to the JOINPOINT
DefineVar : *VARIABLENAME* (= *EXPRESSION*)?	a weaving problem	Defines a variable. The final value for the variable is bound by the weaver and must cause the optional EXPRESSION to evaluate to true
Define : *VARIABLENAME* = *EXPRESSION*	a weaving problem	Defines a variable and sets a constant value for it
Goal : (*maximize*|*minimize*), *VARIABLE_EXPRESSION*	a weaving problem	Defines an expression over the properties of ADVICE and JOINPOINTS that should be maximized or minimized by the weaving

Table VI.5: AspectScatter DSL Directives

EXPRESSION	(*CONSTANT*|*VARIABLE_EXPRESSION*) (+|−|×) (*CONSTANT*|*VARIABLE_EXPRESSION*)	An expression
CONSTRAINT	(*VARIABLE_EXPRESSION*|*CONSTANT*) (<|>|=|!=|=<|>=) (*VARIABLE_EXPRESSION*|*CONSTANT*)	Defines a constraint that must hold true in the final weaving solution.
VARIABLE_EXPRESSION	(*VARIABLE_V_EXPRESSION*|*CONSTANT*) (+|−|×) (*VARIABLE_V_EXPRESSION*|*CONSTANT*)	An expression over a set of variables
VARIABLE_V_EXPRESSION	(Target|Source).VARIABLENAME	The value of the specified defined variable (VARIABLENAME) on a ADVICE or JOINPOINT element. *Target* specifies that the variable should be resolved against the JOINPOINT matched by the enclosing ADVICE. *Source* specifies that the variable should be resolved against the enclosing ADVICE element.

Table VI.6: AspectScatter DSL Expressions

to the same joinpoint as the `GenericCache`. The `GenericCache` declaration also specifies a property variable, called `CacheSize`, that the weaver must determine a value for.

Assume that the `GenericCache` is A_2 and the `FishCache` is A_1. The AspectScatter specification would be transformed into: the mapping variables $M_{20} \ldots M_{2n}$, the advice property variables $PA_{20} \ldots PA_{2k}$, an advice property table column for `CacheSize`, and the CSP constraint $\forall B_j \subset \beta, (M_2 j = 1) \rightarrow (M_1 j = 0)$.

The final part of an AspectScatter DSL instance is an optional set of global variable definitions and an optimization goal. The global variable definitions are defined in an element named `Globals`. Within the `Globals` element, properties can be defined that are not specific to a single `ADVICE` or `JOINPOINT`. Furthermore, the `Goal` directive key word can be used within the `Globals` element to define the function that the constraint solver should attempt to maximize or minimize.

The values for variables provided by the weaver are determined by labeling the CSP for the weaving problem. For example, the global constraints for the Pet Store weaving problem define the goal as the minimization of the response time of the `ItemDAO` and `ProductDAO`, as can be seen below:

```
Globals {
  Define:TotalFish = 100;
  Define:TotalBirds = 75;
  Define:TotalOtherAnimals = 19;
  Constraint:Sources.CacheSize.Sum < 1024;
  Goal:minimize, ProductDAO.RequestPercentage * ProductDAO.ResponseTime +
                          ItemDAO.RequestPercentage * ItemDAO.ResponseTime;
}
```

Each `Define` creates a variable in the CSP and sets its value. The variable created by the `Define` can then have a constraint bound to it. For example, the variable `TotalBirds` is used in the constraint $(\sum_{j=0}^{n} M_{0j} > 0) \rightarrow (TotalBirds < 80)$. This simple constraint states that the 0th advice element can only be applied to a joinpoint if there are less than 80 birds.

The `Constraint` directive adds a constraint to the CSP. In the example above, the specification adds a constraint that the sum of the cache sizes must be less than 1024. The statement "`Sources.CacheSize.Sum`" is a special AspectScatter language expression for obtaining a value from a properties table (the advice properties table), a column (`CacheSize`), and an operation (summation). Assuming `CacheSize` is the 0th column in the advice properties table, the statement adds the following constraint to the CSP:

$$\forall B_j \subset \beta, (\sum_{i=0}^{n} (M_{ij} * PA_{i0}) < 1024)$$

Since no explicit values for each advice element's `CacheSize` is set, these will be variables that the solver will need to find values for as part of the CSP solving process. Because the response times of the DAOs are dependent on the size of each cache, the `CacheSize` variables will be set by the weaver to minimize response time. Developers

can use the AspectScatter DSL to produce complex aspect weaving problems with both global constraints and goals.

AspectScatter's DSL also includes support for the filtering operations described previously. Filters to restrict the potential joinpoints that an advice element can be mapped to can be defined using an Object Constraint Language (OCL) [140] or Groovy [83] language expression that must hold true for the advice/joinpoint mapping (*i.e.*, the choice of expression language is up to the user). Filters are defined via the `Evaluate` directive. For example, a Groovy constraint can be used to restrict the FishCache from being applied to any order related DAOs via a regular expression constraint:

```
FishCache {
 ...
 Evaluate:groovy,{!target.name.contains("Order")};
}
```

An OCL constraint could be used to further restrict the FishCache to only be applied to DAOs that receive requests from a category listing page:

```
FishCache {
 ...
 Evaluate:ocl,{target.requestsFrom->collect(x | x.name = 'ViewCategories.jsp')->size() > 0};
}
```

The filter expressions defined via `Evaluate` are used to preprocess the weaving CSP and eliminate unwanted advice/joinpoint combinations.

AspectScatter Model Transformation Language

The result of solving the CSP is a platform-independent weaving solution that symbolically defines which advice elements should be mapped to which joinpoints. This symbolic weaving solution still needs to be translated into a platform-specific weaving model, such as an AspectJ weaving specification. The platform-specific weaving specification can then be executed to perform the actual code or model weaving.

Each platform-independent weaving representation of the weaving solution can be transformed into multiple platform-specific weaving solutions, such as AspectJ, C-SAW, or AMW specific weaving specifications. Producing a platform-independent weaving model of the solution and transforming it into implementations for specific tools allows AspectScatter to eliminate much of the significant manual effort required to synchronize multiple weaving specifications across a diverse set of models, modeling languages, and modeling tools. For example, when the modeled request distribution changes for the Pet Store, the C-SAW, AspectJ, and GEMS weaving specifications can automatically be re-generated by AspectScatter, as shown in Step 4 of Figure VI.6.

AspectScatter's platform-independent weaving model can be transformed into a platform-specific model with a number of Java-based model transformation tools, such as ATL [85]. AspectScatter also includes a simple model transformation tool based on pointcut generation templates that can be used to create the platform-specific weaving model. In this section, we show the use of the built-in transformation language in the context of the C-SAW weaving definition files needed for the GME model.

C-SAW weaves the caching specification into the GME architecture according to a set of weaving directives specified in a weaving definition file. The implementation of the C-SAW weaving definition file that is used to merge caches into the architecture model is produced from the platform-independent weaving solution model. To transform the platform-independent solution into a C-SAW weaving definition file, an AspectScatter model transformation is applied to the solution to create C-SAW *strategies* to update model elements with caches and C-SAW *aspects* to deduce the elements to which the strategies should be applied. For each cache inserted into the GME architecture model, two components must be added to the C-SAW weaving definition file. First, the *Strategy* for updating the GME model to include the cache and connect it to the correct component must be created, as shown below:

```
strategy ProductDAOAddGenericCache( ) {
    declare parentModel : model;
```

```
    declare component, cache : atom;
    parentModel := parent();
    component := self;
    cache := parentModel.addAtom("Cache", "GenericCacheForProductDAO");
    parentModel.addConnection("CacheInstallation",cache,component);
}
```

A root `Aspect` and `Strategy` must also be created that matches the root element of the GME model and invokes the weaving of the individual DAO caches. The root definitions are shown below:

```
aspect RootAspect()
{
  rootFolder().models()->AddCaches();
}
strategy AddCaches()
{
    declare parentModel : model;
    parentModel := self;
    parentModel.atoms("Component")->select(m|m.name() == "ProductDAO")->ProductDAOAddGenericCache ( );
    ....
}
```

For each advice/joinpoint combination, the `Strategy` to weave in the cache must be created. Moreover, for each advice/joinpoint combination, a weaving instruction must be added to the root `AddCaches` strategy to invoke the advice/joinpoint specific weaving strategy.

To create the advice/joinpoint specific cache weaving strategy, an AspectScatter template can be created, as follows:

```
#advice[*](for-each[list=targets]){#
strategy ${value}Add${advice}Cache( ) {
    declare parentModel : model;
    declare component, cache : atom;
    parentModel := parent();
    component := self;
    cache := parentModel.addAtom("Cache", "${advice}CacheFor${value}");
    parentModel.addConnection("CacheInstallation",cache,component);
```

```
}
#}#
```

The template defines that for all advice elements matched against joinpoints "*advice*[∗]", iterate over the joinpoints that each advice element is applied to "`for-each[list=targets]`", and create a copy of the template code between "{#" and "#}" for each target joinpoint. Moreover, each copy of the template has the name of the advice element and target element inserted into the placeholders "${advice}" and "${value}", respectively. The "${advice}" placeholder is filled with the symbolic name of the advice element from its `ADVICE` declaration in the AspectScatter DSL instance.

The "${value}" placeholder is the symbolic name of the joinpoint, also obtained from its definition in the AspectScatter DSL instance, that the advice element has been mapped to. The properties of an advice element can also be referred to using the placeholder "`${PROPERTYNAME}`." For example, the property `CacheSize` of the advice element could be referred to and inserted into the template by using the placeholder "${CacheSize}".

After deriving a weaving solution, AspectScatter uses the templates defined for C-SAW to produce the final weaving solution for the GME model. Invoking the generated C-SAW file inserts the caches into the appropriate points in the architecture diagram. A final woven Pet Store architecture diagram in GME can be seen in Figure VI.8.

With existing weaving approaches, each time the global properties, such as request distributions change, developers must manually derive a new weaving solution. When the properties of the solution models change, however, AspectScatter can automatically solve for new weaving solutions, and then use model transformation to generate the platform-specific weaving implementations, thereby addressing Challenge 2. The CSP formulation of a weaving problem is based on the weaving constraints and not specific solution model properties. As long as the constraint relationships do not change, AspectScatter can automatically re-calculate the weaving solution and regenerate the weaving implementations.

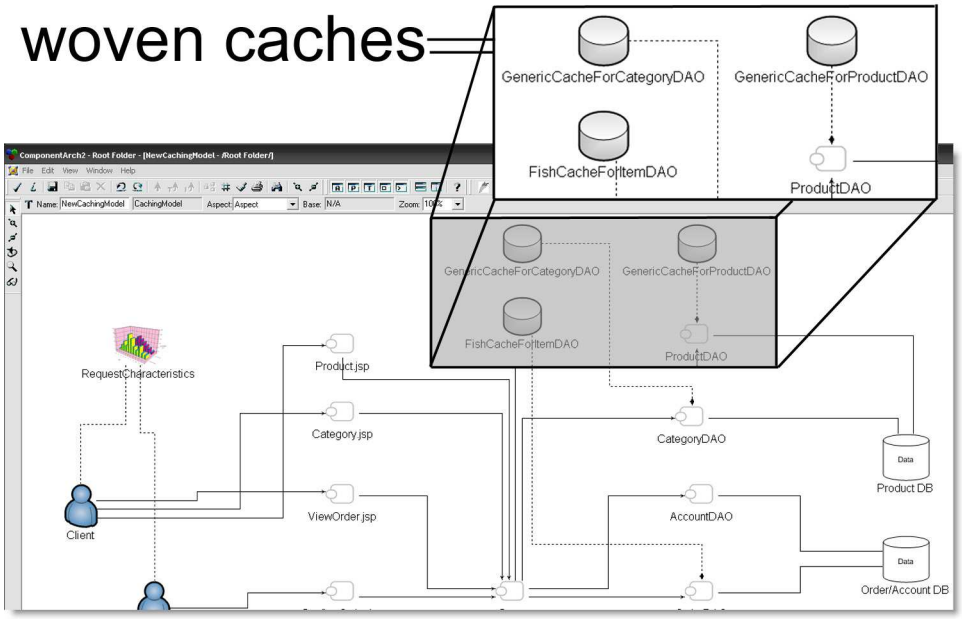


Figure VI.8: The GME Architecture Model with Caches Woven in by C-SAW

For example, if new request distributions are obtained, AspectScatter can re-calculate the weaving solution to accomodate the new information. Automatically updating the weaving solution as the solution model properties change can save substantial development effort across multiple solution model refactorings.

Applying Constraint-based Weaving to the Java Pet Store

This section demonstrates the reduction in manual effort and complexity achieved by applying AspectScatter to the Spring Java Pet Store to handle global constraints and generate platform-specific weaving implementations. For comparison, we also applied the existing weaving platforms C-SAW and AspectJ to the same code base using a manual weaving solution derivation process. The results document the manual effort required to derive and implement a caching solution for the Pet Store's `ItemDAO` and `ProductDAO`.

Manual Complexity Overview

It is difficult to directly compare the manual effort required to execute two different aspect weaving processes. The problem is that there is no way of correlating the relative difficulty of the individual tasks of each process. Furthermore, the relative difficulty of tasks may change depending on the developer.

Although it is difficult to quantify the relative difficulty of the individual steps, we can define functions $M(WP)$ and $M'(WP)$ to calculate the total number of manual steps required for each process as a function of the size of the weaving problem (WP) input. That is, as more advice elements, joinpoints, and constraints are added to the weaving problem, how is the number of manual steps of each process impacted? What we can show is that one process exhibits a better algorithmic O bound for the number of manual steps as a function of the input size.

Let's assume that each step in one process is E times harder than the steps in the second process. This gives the formula:

$$E * M_{step} = M'_{step}$$

Even if there is some unknown coefficient E, representing the extra effort of each step in the process yielding $M'(WP)$, if $M'(WP)$ posseses a better O bound, then there must exist an input, $wp_i \subset WP$ (WP is sorted in ascending order based on size), for which:

$$E * M'(wp_i) \leq M(wp_i)$$

and for all $wp_x \subset (wp_{i+1} \ldots wp_n)$:

$$E * M'(wp_x) < M(wp_x)$$

Once the size of the weaving problem reaches size wp_{i+1}, even though the steps in M' are E times more complicated than the steps in $M(WP)$, the faster rate of growth of the function $M(WP)$ makes it less efficient. If we can calculate O bounds for the number of manual steps required by each process as a function of the size of the weaving problem, then we can definitively show that for large enough problems, the process with the better O bound will be better.

In order to compare the AspectScatter based approach to our original C-SAW and AspectJ approach, we provide an example weaving problem involving global constraints and optimization. We apply each process to the problem to show the manual steps involved in the two processes. Next, we calculate functions $M(WP)$ and $M'(WP)$, for the traditional and AspectScatter processes respectively, and show that $M'(WP)$ exhibits a superior O bound.

Experimental Setup

We evaluated both the manual effort required to use the existing weaving solutions to implement a potentially non-optimal caching solution and the effort required to derive and implement a guaranteed optimal caching solution. By comparing the two different processes using existing weavers, we determined how much of the manual effort results from supporting multiple weaving platforms and how much results from the solution derivation process. Both processes with existing tools were then compared to a process using AspectScatter to evaluate the reduction in solution derivation complexity and solution implementation effort provided by AspectScatter.

Deriving and Implementing a Non-Optimal Caching Solution with Existing Weaving Techniques

The results for applying existing weavers to derive and implement a non-optimal caching solution are shown in Figure VI.9. Each individual manual set of steps is associated with

Existing Model Weaving Approach w/o Optimization

Initial Implementation

Activity	Step	Min Lines of Code	Max Lines of Code	Min Steps	Max Steps
Create Aspects				1	1
Identify/Define Joinpoints				1	1
Derive Caching Strategy				1	1
Implement Weaving Specification for C-SAW	Create AddCache Strategies	8	48	1	6
Implement Weaving Specification for C-SAW	Create Root AddCaches Strategy	1	6	1	1
Implement Weaving Specification for AspectJ	Add ProductDAO / ItemDAO Proxy	11	22	1	2
Implement Weaving Specification for AspectJ	Add Cache Beans	3	18	1	6
Implement Weaving Specification for AspectJ	Apply Cache Beans to ProductDAO/ItemDAO Methods	1	6	1	6
Totals		24	100	8	24

Refactoring for Request Distribution Change

Derive New Caching Strategy				1	1
Implement Weaving Specification for C-SAW	Remove Unused AddCache Strategies	0	48	1	6
Implement Weaving Specification for C-SAW	Remove Unused AddCaches Strategy	0	6	1	1
Implement Weaving Specification for C-SAW	Create AddCache Strategies	8	48	1	6
Implement Weaving Specification for C-SAW	Create Root AddCaches Strategy	1	6	1	1
Implement Weaving Specification for AspectJ	Remove Previous Proxies	0	22	1	2
Implement Weaving Specification for AspectJ	Remove Previous Cache Beans	0	18	1	6
Implement Weaving Specification for AspectJ	Remove Unused Cache Beans from ProductDAO/ItemDAO Methods	0	6	1	6
Implement Weaving Specification for AspectJ	Add ProductDAO / ItemDAO Proxy	11	22	1	2
Implement Weaving Specification for AspectJ	Add Cache Beans	3	18	1	6
Implement Weaving Specification for AspectJ	Apply Cache Beans to ProductDAO/ItemDAO Methods	1	6	1	6
Totals		24	200	11	43

Figure VI.9: Manual Effort Required for Using Existing Model Weaving Techniques Without Caching Optimization

an activity that corresponds to the process diagram shown in Figure VI.4. The results tables contain minimum and maximum values for the number of steps and lines of code. The implementation of each step is dependent on the solution chosen. The minimum value assumes that only a single cache is woven into the Pet Store, whereas the maximum value assumes every possible cache is used.

The top table in Figure VI.9 shows the effort required to produce the initial caching solution and implementation for the Pet Store. In the first two steps, developers identify and catalog the advice and joinpoint elements. Developers then pick a caching architecture (which may or may not be good or optimal) that will be used to produce a weaving solution. In the next three steps, developers must implement the weaving solution as a C-SAW

weaving definition file. Finally, developers must update the Spring bean definition file with various directives to use AspectJ to weave the caches into the legacy Pet Store code base.

The bottom table in Figure VI.9 documents the steps required to update the caching architecture and weaving implementation to incorporate a change in the distribution of request types to the Pet Store. In the first step, the developer derives a new caching architecture. In the next 12 steps, developers remove any caches from the original C-SAW and AspectJ implementations that are no longer used by the new solution and implement the new caching solution using C-SAW and AspectJ.

Deriving and Implementing an Optimal Caching Solution with Existing Weaving Techniques

Figure VI.10 presents the manual effort to derive and implement an optimal caching solution for the Pet Store using existing weavers. The change in this experiment is that it

Existing Model Weaving Approach w/ Optimization

Initial Implementation

Activity	Step	Min Lines of Code	Max Lines of Code	Min Steps	Max Steps
Create Aspects				1	1
Identify/Define Joinpoints				1	1
Derive Optimal Caching Strategy	Arch			19	115
Implement Weaving Specification for C-SAW	Create AddCache Strategies	8	48	1	6
Implement Weaving Specification for C-SAW	Create Root AddCaches Strategy	1	6	1	1
Implement Weaving Specification for AspectJ	Add ProductDAO / ItemDAO Proxy	11	22	1	2
Implement Weaving Specification for AspectJ	Add Cache Beans	3	18	1	6
Implement Weaving Specification for AspectJ	Apply Cache Beans to ProductDAO/ItemDAO Methods	1	6	1	6
Totals		24	100	26	138

Figure VI.10: Manual Effort Required for Using Existing Model Weaving Techniques With Caching Optimization

measures the manual effort required to derive an optimal solution for the Pet Store by calculating the Pet Store's response time using each potential caching architecture and choosing the optimal one. The steps for implementing the weaving solution are identical to those from the results presented in Figure VI.9.

The steps labeled *Derive Optimal Caching Strategy* in Figure VI.10 presents the manual optimal solution derivation effort incorporated into this result set. First, developers must

enumerate and check the correctness according to the domain constraints, or each potential caching architecture for both the `ProductDAO` and `ItemDAO`. Developers must then enumerate and check the correctness of the overall caching architectures produced from each unique combination of `ProductDAO` and `ItemDAO` caching architectures. After determining the set of valid caching architectures, developers must use the Pet Store's modeled request distribution, memory constraints, and response time goals to derive the optimal cache sizes and best possible response time of each caching architecture. Finally, developers select the optimal overall architecture and implement it using C-SAW and AspectJ.

As shown in Figure VI.11, refactoring the weaving solution to accomodate the solution model change in request type distributions forces developers to repeat the entire process. First, they must go back and perform the optimal solution derivation process again. After a new result is obtained, the existing solution implementations in C-SAW and AspectJ must be refactored to mirror the new caching structure.

Existing Model Weaving Approach w/ Optimization

Refactoring for Request Distribution Change

Activity	Step	Min Lines of Code	Max Lines of Code	Min Steps	Max Steps
Derive Optimal Caching Strategy				19	115
Implement Weaving Specification for C-SAW	Remove Unused AddCache Strategies	0	48	1	6
Implement Weaving Specification for C-SAW	Remove Unused AddCaches Strategy	0	6	1	1
Implement Weaving Specification for C-SAW	Create AddCache Strategies	8	48	1	6
Implement Weaving Specification for C-SAW	Create Root AddCaches Strategy	1	6	1	1
Implement Weaving Specification for AspectJ	Remove Previous Proxies	0	22	1	2
Implement Weaving Specification for AspectJ	Remove Previous Cache Beans	0	18	1	6
Implement Weaving Specification for AspectJ	Remove Unused Cache Beans from ProductDAO/ItemDAO Methods	0	6	1	6
Implement Weaving Specification for AspectJ	Add ProductDAO / ItemDAO Proxy	11	22	1	2
Implement Weaving Specification for AspectJ	Add Cache Beans	3	18	1	6
Implement Weaving Specification for AspectJ	Apply Cache Beans to ProductDAO/ItemDAO Methods	1	6	1	6
Totals		24	200	29	157

Figure VI.11: Manual Effort Required for Using Existing Model Weaving Techniques to Refactor Optimal Caching Architecture

Deriving and Implementing an Optimal Caching Solution using AspectScatter

Figure VI.12 contains the steps required to accomplish both the initial implementation of the Pet Store caching solution and the refactoring cost when the request distribution changes. In steps 1 and 2, developers use AspectScatter's DSL to specify the caches,

Aspect Scatter					
Initial Implementation					
Activity	**Step**	**Min Lines of Code**	**Max Lines of Code**	**Min Steps**	**Max Steps**
Create Aspects		12	12	6	6
Identify/Define Joinpoints		12	12	2	2
Derive Optimal Caching Strategy	Define Weaving Goal	1	1	1	1
Implement Weaving Specification for C-SAW	Create AddCache Model Transformation	8	8	1	1
Implement Weaving Specification for C-SAW	Create Root AddCaches Model Transformation	6	6	1	1
Implement Weaving Specification for AspectJ	Create ProductDAO / ItemDAO Proxy Model Transformation	22	22	2	2
Implement Weaving Specification for AspectJ	Create Cache Beans Model Transformation	18	18	6	6
Implement Weaving Specification for AspectJ	Create Cache Beans to ProductDAO/ItemDAO Methods Model Transformation	1	1	1	1
Implement Weaving Specification	Invoke AspectScatter	1	1	1	1
Totals		81	81	21	21
Refactoring for Request Distribution Change					
Identify/Define Joinpoints	Update Request Distribution Properties	1	2	1	2
Implement Weaving Specification	Invoke AspectScatter	1	1	1	1
Totals		2	3	2	3

Figure VI.12: Manual Effort Required for Using AspectScatter With Caching Optimization

joinpoints, and constraints for the weaving problem. Developers then define the weaving goal, the response time of the application in terms of the properties of the joinpoints and advice elements woven into a solution. The goal is later used by AspectScatter to ensure that the derived weaving solution is optimal.

The next two steps (3 and 4) require the developer to create a model transformation, using AspectScatter's transformation templates to specify how to transform the platform-independent weaving solution into a C-SAW implementation. The approach thus represents a higher-order transformation where C-SAW transformations are generated from more abstract transformation rules. The subsequent three steps define a model transformation to produce the AspectJ implementation. Finally, AspectScatter is invoked to deduce the optimal solution and generate the C-SAW and AspectJ implementations.

The bottom of Table VI.12 presents the steps required to refactor the solution to accomodate the change in request distributions. Once the aspect weaving problem is defined using AspectScatter's DSL, the change in request distributions requires updating one or both of the request distribution properties of the two joinpoints (*i.e.*, the `ProductDAO`

and `ItemDAO`) in the AspectScatter DSL instance. After the properties are updated, AspectScatter is invoked to recalculate the optimal caching architecture and regenerate the C-SAW and AspectJ implementations using the previously defined model transformations.

Results Analysis and Comparison of Techniques

By comparing the initial number of lines of code (shown in Figures VI.9-VI.12) required to implement the caching solution using each of the three techniques, the initial cost of defining an AspectScatter problem and solution model transformations can be derived. AspectScatter initially requires 81 lines of code versus between 24 and 100 for the approach based on existing techniques. The number of lines of code required to implement the initial weaving specification grows at a rate of $O(n)$, where n is the number of advice and joinpoint specifications, for both AspectScatter and existing approaches. The more advice and joinpoint specifications, the larger each weaving specification needs to be.

The benefit of AspectScatter's use of model transformations becomes most apparent by comparing the refactoring results. AspectScatter only requires the developer to change between 1-2 lines of code before invoking AspectScatter to regenerate the C-SAW and AspectJ implementations. Using the existing weaving approaches, the developer must change between 24-200 lines of code. Moreover, this manual effort required by the existing approaches is incurred *per solution model change*. Thus, AspectScatter requires a constant or $O(1)$ number of changes per refactoring while existing approaches require $O(n)$ changes per refactoring.

For a single aspect weaving problem without optimization that is implemented and solved exactly once, both AspectScatter and the manual weaving approach exhibit roughly $O(n)$ growth in lines of code with respect to the size of the weaving problem. The more caches that need to be woven, the larger the weaving specifications have to be for both processes. For a single weaving in this scenario, we cannot directly show that AspectScatter provides an improvement since it has an equivalent big O bound.

If we calculate the weaving cost over K refactorings, however, we see that AspectScatter exhibits a bound of $O(2K+n) = O(K+n)$ lines of code. ApsectScatter requires an initial setup cost of $O(n)$ lines of code and then each of the K refactorings requires manually changing 1-2 lines of code. The manual approach requires $O(n)$ lines of code changes for each of the K refactorings because the developer may have to completely rewrite all of the joinpoint specifications. Over K refactorings, the manual process requires $O(Kn+n) = O(Kn)$ lines of code changes. Thus, AspectScatter provides a better bound, $O(K+n) < O(Kn)$ on the rate of growth of the lines of code changed over multiple refactorings.

When optimization is added to the scenarios, AspectScatter's reduction in manual complexity becomes much more pronounced. With existing approaches, each time the weaving solution is implemented, the developer must calculate the optimal cache weaving architecture. Let γ be the number of manual steps required to calculate the optimal cache weaving architecture, then the cost of implementing the initial weaving solution with an existing approach is $O(n+\gamma)$. The developer must implement the $O(n)$ lines of code for the weaving specification and derive the optimal architecture.

Since we are doing a big O analysis, we will ignore any coefficients or differences in difficulty between a step to implement a line of code and a step in the derivation of the optimal caching architecture. We will say that n lines of code require n manual steps to implement. The next question is how the number of steps γ grow as a function of the size of the weaving problem. The caching optimization problem with constraints is an instance of a mixed integer optimization problem, which is in NP, and thus has roughly exponential complexity. Thus, $\gamma = \theta^n$, where θ is a constant

The overall complexity of the existing approach for the optimization scenario is $O(n+\theta^n)$. Note, this complexity bound is for solving a single instance of the weaving problem. Over K refactorings, the complexity bound is even worse at $O(n+K(n+\theta^n))$. With AspectScatter, the solver performs the optimization step on the developer's behalf and the θ^n manual steps are eliminated. When optimization is included and K refactorings are

performed, AspectScatter shows a significantly better bound on manual complexity than existing approaches:

$$O(n+K) < O(n+K(n+\theta^n))$$

One might argue that a developer wouldn't manually derive the optimal caching architecture by hand but would instead use some automated tool. We note, however, that this is essentially arguing for our approach, since we are using an external tool to derive the caching architecture and then using code generation to automatically implement the solution. Thus, even using an external tool would still require a developer to rewrite the weaving specification after each refactoring and would also add setup cost for specifying the weaving problem for the external tool and translating the results back into a weaving solution. Our approach automates all of these steps on behalf of the developer.

A final analysis worth looking at is the effect of the number of weaving platforms on the complexity of the weaving process. For both processes, the overhead of the initial setup of the weaving solution is linearly dependent on the number of weaving platforms used. In the experiments, AspectJ and C-SAW are used as the weaving platforms. Given P weaving platforms, both processes exhibit an initial setup complexity of $O(Pn)$.

With existing processes, when K refactorings are performed, the number of weaving platforms impacts the complexity of each refactoring. Rather than simply incurring $O(n)$ complexity for each refactoring, developers incur $O(Pn)$ per refactoring. This leads to an overall complexity bound of $O(Pn+KPn)$ for existing processes versus a bound of $O(Pn+K)$ for AspectScatter. As we showed in the previous analyses, even for a single weaving platform, such as just AspectJ, AspectScatter reduces complexity. However, when numerous weaving platforms are used AspectScatter shows an even further reduction in complexity.

Weaving Performance

There is no definitive rule to predict the time required to solve an arbitrary CSP. The solution time is dependent on the types of constraints, the number of variables, the degree of optimality required, and the initial variable values provided to the solver. Furthermore, internally, the algorithms used by the solver and solver's implementation language can also significantly affect performance.

Our experience with AspectScatter indicated that the weaving process usually takes 10ms to a few seconds. For example, to solve a weaving problem involving the optimal weaving of 6 caches that can be woven into any of 10 different components with fairly tight memory constraints requires approximately 120ms on an Intel Core 2 Duo processor with 2 gigabytes of memory. If a correct—but not necessarily optimal solution is needed—the solving time is roughly 22ms. Doubling the available cache memory budget essentially halves the optimal solution derivation time to 64ms. The same problem expanded to 12 caches and 10 components requires a range from 94ms to 2,302ms depending on the tightness (i.e., amount of slack memory) of the resource constraints.

In practice, we found that AspectScatter quickly solves most weaving problems. It is easy to produce synthetic modeling problems with poor performance, but realistic model weaving examples usually have relatively limited variability in the weaving process. For example, although a caching aspect could theoretically be applied to any component in an application, this behavior is rarely desired. Instead, developers normally have numerous functional and other constraints that bound the solution space significantly. In the Pet Store, for example, we restrict caching to the four key DAOs that form the core of the middle-tier.

In cases where developers do encounter a poorly performing problem instance, there are a number of potential courses of action to remedy the situation. One approach is to relax the constraints, *e.g.*, allow the caches to use more memory. Developers can also improve solving speed by accepting less optimal solutions, *e.g.*, solving for a cache architecture

that produces an average response time below a certain threshold rather than an optimal response time. Finally, developers can try algorithmic changes, such as using different solution space search algorithms, *e.g.*, simulated annealing [118], greedy randomized adaptive search [118], and genetic algorithms [118].

CHAPTER VII

MANUAL CONFIGURATION OPTIMIZATION

Challenge Overview

This chapter illuminates the challenges of modeling the configuration of software intensive systems, motivates why manual approaches are not sufficient for these domains, and shows how automated modeling guidance mechanisms are needed to help guide manual modeling. The chapter evaluates the limitations of related work in the area of modeling guidance and demonstrates the current limitations. The chapter then presents an approach to providing modelers with modeling guidance from a constraint solver. Specific emphasis is placed on how modeling guidance can be used to reduce the complexity of modeling software intensive systems. Finally, the chapter illustrates how a constraint solver can be integrated into a graphical modeling tool.

Introduction

The complexity of modeling an arbitrary domain can be measured along the following three axes:

1. Typical Model Size in Elements: Large Models are harder to work with using a manual approach. Clearly, modeler are more apt to make mistakes managingŮand much more likely to have trouble visualizing - a domain with hundreds of model elements than one with dozens of model elements.

2. Degree of Global Constraint: Global constraints, such as resource constraints, that are dependent on multiple modeling steps or the order of modeling steps make a domain much harder to work with. For example, a constraint requiring the deployment of an ABS component to a single ECU at a certain distance from the perimeter of

the car is relatively easy to solve. It is much harder to solve constraints of an ABS component requiring its deployment to two ECUs, both a minimum distance from the outside of the car and a minimum distance from each other (for fault tolerance guarantees).

3. Degree of Optimality Required: Optimality is hard to achieve with a manual modeling approach. In many domains, such as manufacturing, a small increase in the cost of a solution can lead to a dramatic increase in the overall cost of manufacturing when the millions of units affected by the change are considered. Many solutions must therefore be tried to find the best one. Domains that require optimal or good answers are much more challenging to model.

The key reasons that manual modeling approaches do not scale as modeling domains become more complex are:

- When there are thousands, millions, billions, or more possible ways that a model can be constructed and few correct ones, finding a valid solution is hard.
- A valid solution may not be a good solution in these domains. Often, a modeler may find a solution that is valid but is far from the optimal solution. Automation and numerical methods, such as the Simplex method [109], are needed to efficiently search the solution space and find good candidates. A human modeler cannot effectively search a solution space manually once it grows past a certain magnitude.
- For large models, manual construction methods, such as pointing and clicking to intricately connect hundreds or more components, are tedious and error prone.
- Often, global constraints rely on so much information that not all of the relevant bits of information can be seen at once. When not all of the information can be seen, modelers cannot make an informed decision.

Another difficulty of highly combinatorial domains is that although modelers may create a model that satisfies the domain constraints, the model may be considered poor in quality. For example, a modeler creating a deployment of components to ECUs could easily select a scheme that utilized far more ECUs than the true minimum number required to host the set of components. For domains, such as automotive manufacturing, each modeling decision can have significant cost consequences for the final solution. For example, if a model can be constructed that uses three fewer control units to host the car's components and consequently saves $100 in manufacturing costs, millions of dollars in overall cost reduction for all cars of this make that are manufactured can be achieved. In these cases, it is crucial to not only find a correct solution but to find a cost effective one.

The difficulty of finding a good solution is that with large models and complex global constraints, modelers are lucky to find any valid solution. Since finding a single solution is incredibly challenging, it becomes infeasible or cost prohibitive to produce scores of valid solutions and search for an optimal one. Even if the set of valid solutions is large, there are numerous numerical methods to search for a solution with a given percentage of optimality. These methods, however, all rely on the ability to generate large numbers of valid solutions and are not possible without automation.

In domains with large models and intricate constraints, modelers must be able to see hundreds of modeling moves into the future to satisfy a global constraint or optimize a cost. The more localized a modelers decisions are and the less distant they peer into the future, the less chance there is that a correct or good solution will be found. Good local decisions, also known as "greedy decisions," do not necessarily produce a globally good decision.

For example, consider a simple model that determines the minimum number of ECUs needed to host a set of components. Assume that there are two types of ECUs, one that costs $10 and can host 2 components and another that costs $100 and can host 42 components. If modelers are deploying using a myopic view and not peering into the future, they will select many $10 ECUs and create a solution that costs $210, rather than looking ahead and

choosing two $100 controllers for a final cost of $200. Making a series of locally good decisions may not produce the overall best decision.

Solution Approach

An MDD tool provides a visual language for a developer to build a solution specification. An instance of a visual model contains modeling entities or elements, similar to OO classes, and different visual queues (e.g. connections, containment) specifying relationships between the elements. For example, a connection between a component and an ECU specifies deployment in the automotive modeling example.

The key objective of a modeler is to add the right model entities and relationships between the entities so that they create a solution that meets the application requirements. Modelers express relationships between entities by drawing connections between them, placing entities within each other for containment, or other visual means. For each relationship that a modeler creates between entities, such as deployment, the modeler must find the right source and target for the relationship so that the relationship satisfies any constraints placed on it. In the example of deploying components to ECUs, the modeler must only draw a connection from a component to an ECU that has the OS and resource capabilities to support the component.

As has been shown, the large size of DRE models and their complex constraints can make manually finding the right endpoints for these relationships, such as deployment, infeasible. To address the scalability challenges of manual modeling approaches presented, this section outlines how a constraint solver can be integrated with an MDD tool to help automate the selection of endpoints for relationships between model entities.

In the context of modeling, a constraint solver is a tool that takes as input one or more model elements, a goal that the user is attempting to achieve, and a set of constraints that must be adhered to while modifying the elements to reach the goal. As output, the constraint solver produces a new set of states for the model elements that achieves the desired

goal while adhering to the specified constraints. For example, a set of components can be provided to a constraint solver along with the deployment requirements (constraints) of the components. The goal can then be set to "all components connected to an ECU." The constraint solver will in turn produce a mapping of components to ECUs that satisfies the deployment constraints.

The remainder of this section first outlines the different type of modeling assistance that an MDD tool and integrated constraint solver can provide to a user. Next, the section discusses how a user's actions in an MDD tool can be translated into constraint satisfaction problems (CSPs) so that a constraint solver can be used to automatically derive the correct endpoints for the relationships the user wishes to create. Finally, the section illustrates an architecture for integrating Prolog as a constraint solver into an MDD tool.

Modeling Assistance

There are two types of constraint solver guidance that can be used to help modelers produce solutions in challenging domains: local guidance and batch processes. Local guidance is a mechanism whereby the constraint solver is given a relationship and one endpoint of the relationship and provides a list of valid model entities that could serve as the other endpoint for the relationship. One example is that a constraint solver could be provided a deployment relationship and a component and return the valid ECUs that could be attached to the other end of the connection. This type of local guidance for deploying components is shown in Figure VII.1.

The second type of modeling guidance is for deriving endpoints for a group of relationships so that the group as a whole satisfies a global constraint. An example of a batch process would be to connect each component to an ECU in a manner such that the no ECU hosts more components than its resources can support. A batch process takes an overall goal that the modeler is trying to achieve, such as all components connected to an ECU,

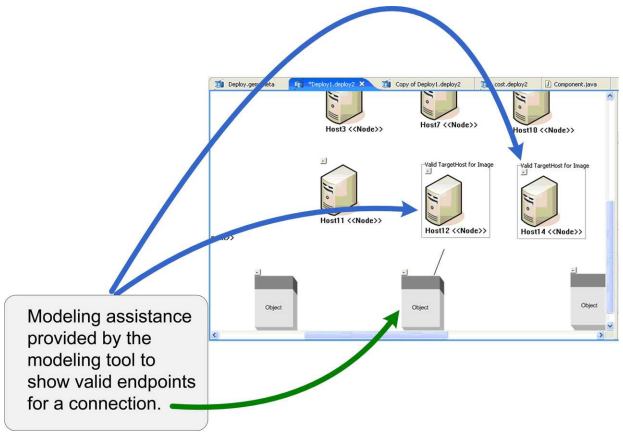


Figure VII.1: Local Modeling Guidance

and creates a series of relationships on behalf of the user to accomplish that goal. By offering both local guidance and batch processes, a MDD tool can help users to accomplish both small incremental refinements to a model and large goals covering multiple modeling steps.

Local Guidance

Local guidance helps modelers correctly complete a single modeling step. A single modeling step is defined as the creation of one relationship between two modeling elements. Local guidance can be implemented as a visual queue that shows the modeler the valid endpoints for a relationship that he or she is creating. For example, when a modeler creates a connection from a component to an ECU to specify where a component is deployed, the modeler must first click on the component modeling element to initiate the connection. When the connection is initiated, the constraint solver can be used to solve for the valid deployment locations for the component and the model elements corresponding to these deployment locations can be highlighted in the model.

Challenges 3 & 4 can be addressed with local guidance. By identifying the model elements that are valid target endpoints of the modeling action a user is performing, a modeling tool can use visual queues (e.g. highlighting, filtering, etc.) to show the user only

the information relevant to the action. Furthermore, the modeling tool can use the list of valid targets to both help the modeler identify valid solutions (helping address challenge 1 of Key Challenges of Complex Domains) and to prevent the user from applying an action to an invalid target endpoint (addressing challenge 3 of Key Challenges of Complex Domains). With a traditional MDD approach, the correctness of a user's action is checked after completion and thus the user may have to do and undo an action multiple times before the correct target endpoint is found. By finding valid solutions before a modeler completes a modeling action, the tool can preemptively constrain (e.g. veto modeling actions) what modeling elements the action can be applied to and prevent tedious and error-prone manual solution searching.

Local guidance can not only provide suggestions of correct endpoints of a relationship but can provide rankings of the local optimality of each of the endpoints. For example, deployment locations could be ranked by the resource slack available on them so that modelers are led to choose deployment targets with sufficient free resources. This manner of local guidance provides a greedy strategy to modeling guidance. At each step, modelers are led towards a solution that provides the greatest immediate benefit to the model's correctness.

Correct solutions to modeling transactions of a single modeling step can be found using local guidance. In some cases, only considering single step transactions will not produce a solution that satisfies global constraints. For example, if modelers can add ECUs as needed to deploy components to, local guidance can produce a solution that is correct with respect to the constraints, although not necessarily optimal. If, however, ECUs cannot be added to the model and the local strategy guides the modeler to a solution where no ECU has free resources and several components are undeployed, the global constraints cannot be met.

Although a greedy strategy may not produce optimal results for certain types of CSPs, such as bin-packing, in many cases these localized strategies can provide a lower bound on the optimality of the final solution. With bin-packing, a First Fit Decreasing (FFD) [39]

packing strategy that sorts items to be placed into bins by their size and non-deterministically selects the first bin that can hold the item will guarantee that the solution never uses more than 1.87 times as many bins as the optimal solution. Providing a lower bound on the quality of the solution that a modeler can produce can be extremely important in some domains, such as automotive manufacturing, where you want to minimize risk or cost. Although not guaranteed, a localized strategy may in fact arrive at an optimal or nearly optimal solution. Moreover, local guidance is substantially less computationally complex than providing a global maximum and can be implemented easily with a number of the approaches discussed later in this section.

Batch Processes

Global constraints require the correct completion of numerous modeling steps and are typically not amenable to user intervention. For global strategies, therefore, batch processes guided by constraint solvers can be used to create multiple relationships to bring the model into a correct state. The key differentiator between local guidance and a batch process is that local guidance deals with modeling transactions involving a single relationship while batch processes operate on modeling transactions containing two or more relationships. The larger the number of relationships in the transaction, generally the more complicated it is to complete.

One possible batch process for the component-to-ECU deployment tool could take each component in the model and create a connection to an ECU in the model to specify a deployment location. Local guidance would produce a single deployment connection for a single component. By increasing the size of the modeling transaction to consider the deployment locations of multiple components, the batch process can use the constraint solver to guarantee that if a possible solution is found, it utilizes only the ECUs currently in the model. By expanding the transaction size that the solver operates on, the batch

process allows it to make model modifications that are not locally optimal, but lead to a globally optimal or globally correct solution.

Batch processes help address challenges 1, 2, & 3. First, a batch process can correctly complete large numbers of modeling actions on behalf of the user, eliminating tedious and error-prone manual modeling (addressing challenge 3). Second, a constraint solver can create both a correct and an optimal solution that can be enacted by a batch process on behalf of the modeler (addressing challenge 1). By tuning the parameters used by the constraint solver, the modeler can guarantee both optimality and correctness (addressing challenge 2).

Transforming Non-functional Requirements into Constraint Satisfaction Problems

To integrate local and batch process guidance from a constraint solver, a model and the actions that modelers can perform on the model must be transformed into a series of Constraint Satisfaction Problems (CSPs). This transformation allows the MDD tool to translate the actions of users into queries for a constraint solver. Valid satisfactions of the CSPs correspond to correct ways of completing a modeling action, such as creating a connection.

A CSP is a set of variables and constraints over the values assigned to the variables. For example, $X < Y < 6$ is a CSP with integer variables X and Y. Solving a CSP is finding a set of values (a labeling) for the variables such that the constraints hold true. The labeling $X = 3$, $Y=4$, is a correct labeling of $X < Y < 6$. A constraint solver takes a CSP as input and produces a labeling (if one exists) of the variables. Solvers may also produce labelings that attempt to maximize or minimize variables. For example, $X = 4$, $Y =5$, is a labeling that maximizes the value of X.

For the deployment example, a deployment of a set of components to a set of ECUs can be viewed as a binary matrix where the cell at row i and column j is 1 if and only if the ith component is deployed to the jth ECU (and 0 otherwise). Each cell can be represented as an

independent variable in a CSP. Thus, each variable Dij determines if the ith component is deployed to the jth ECU. Finding a correct labeling of the values for the D variables creates a deployment matrix that can be used to determine where components should be placed.

Assume that the ABS (anti-lock braking system) component and the WheelRPMs components must be deployed to the same ECU. Also assume that the ABS component must be placed on an ECU at least 3 feet from the perimeter of the car. This series of deployment constraints can be translated in a CSP model. Let the ABS component be the 0th component and the WheelRPMs component be the 1st component. First, the constraint that the ABS component be deployed to the same ECU as the WheelRPMs component is encoded as (D0j = 1) $\rightarrow$ (D1j = 1). Next, for each ECU, a constant Distj can be created to store the distance of the jth ECU from the perimeter of the car. Using these constants, the constraint on the placement of the ABS component relative to the perimeter of the car can be encoded as (D0j = 1) $\rightarrow$ (Distj $\geq$ 3). If this CSP is input into a constraint solver, the solver will label the variables and produce a deployment matrix that is guaranteed to be correct with respect to the deployment constraints.

A constraint solver can also be used to derive a solution with a certain degree of optimality. Assume that N components need to be deployed to one or more of M ECUs using as few ECUs as possible. A new variable UsedECUs can be introduced to store the total number of ECUs used by a solution. The constraint UsedECUs = $\sum$Dij for all i from 0..N and all j from 0..M. The solver can then be asked to produce a labeling of the variables Dij that minimizes the variable UsedECUs. The solver will in turn produce a valid deployment of the components to ECUs that also minimizes the total number of ECUs used.

Constraint solvers typically offer a number of solution optimization options. The options range from maximizing or minimizing a function to using a fast approximation algorithm that guarantees a specific worst-case percentage of optimality. Depending on the constraint solver settings used, a modeler can guarantee the optimality of a model or trade

a certain percentage of model optimality for significantly reduced solving time. In contrast, a manual modeling approach provides no way to guarantee correctness, optimality, a percentage of optimality, or a tradeoff between optimality and solution time. For software intensive systems where optimality is important, allowing modelers to tune these parameters is a key advantage of using a constraint solver-integrated modeling approach.

One goal of using a constraint solver is to produce better solutions than a human modeler can create manually and to produce good solutions more reliably. When a solver uses either optimal or approximation algorithms, the solver's solution has a known and guaranteed worst case solution quality. In contrast, there is no guarantee on the solution quality with a manual approach.

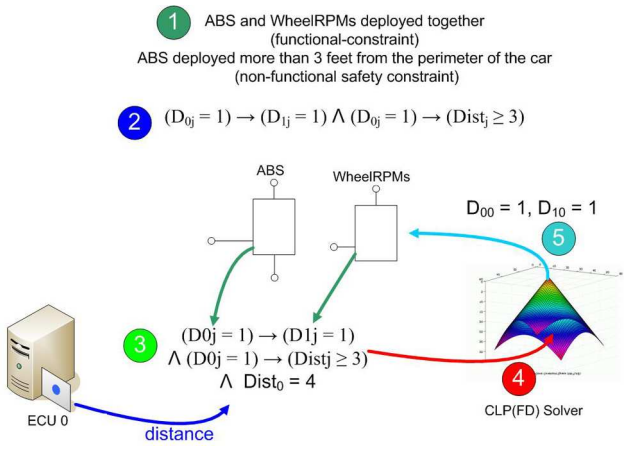


Figure VII.2: Transforming a Model into a Constraint Satisfaction Problem

As shown in Figure VII.2, the non-functional requirements for the software system must first be collected and documented (step 1). Each non-functional requirement must then be translated into a CSP, such as a system of linear equations (step 2). At this point, the data from the model, such as ECU distances to the car perimeter, are collected and bound to variables in the CSP produced in the previous step (step 3). Next, the CSP with some bound variables (such as resource demands) and some unbound variables (such as the Dij variables in Figure VII.2) are input into the constraint solver (step 4). The constraint

solver then produces bindings for the unbound variables and maps them back to changes in the model (step 5).

A crucial element for creating the right translation from non-functional requirements to a set of CSPs is the abstraction used to decompose the model into the variables and facts (i.e. bound variables) that the CSPs operate on. For example, should ECU and component be present in the formulation of the CSP to represent the bin-packing of the model's resources? The metamodel of a language provides the terminology and syntactic rules for a modeling language. Since the metamodel contains a precise definition of the relevant types in a modeling language it is ideal for identifying the key concepts that the CSPs should use. The metamodel of a modeling language can be viewed as a set of model entities and the role-based relationships between them. By using this abstraction based on entities and role-based relationships, a model can be conveniently decomposed for processing by a constraint solver. The idea of relationships between elements is the same as the widely used Resource Description Framework's predicate / argument format.

The role-based relationships of an entity represent both its properties (such as available CPU) and its associations (such as hosted components). Each entity can be decomposed into a unique ID and a set of role-based relationships associated with the ID. A requirement, such as "a component is only deployed to an ECU with the correct OS" can be translated into a CSP involving the Deployment, and OS relationships of a component and ECU. The variables of the CSP for this requirement would be the component and ECU that are being associated through the Deployment relationship. The constraint would be that the OS relationship of the component and the ECU had the same value (i.e. the same OS).

Associating Modeling Actions with the Constraint Solver

An important integration question is how/when to invoke the constraint solver and what CSPs and variable bindings should be passed to it. The goal is to use the constraint solver to provide local guidance and batch processes to bind the endpoints of relationships in

the model. A constraint solver requires a CSP, a set of unbound variables (e.g. unbound endpoints), and a set of bound variables to produce a list of endpoints for relationships. Thus, users' actions and model state must be interpreted to find the correct CSPs, model entities, and unbound endpoints to pass to the solver. By defining the right formal model of the process by which users' actions are interpreted and translated into input data for the constraint solver, the integration process can be more cleanly defined. This section presents a formal abstraction for a user's interaction with a modeling tool and shows the point in the formal specification at which the constraint solver can be integrated and used to automate relationship endpoint binding decisions.

Modeling actions are transactions that take one or more elements of the model and modify the endpoints of the selected elements' role-based relationships. Creating a deployment connection takes a component (the source of the connection) and sets the endpoint of its TargetECU relationship. A modeling action was defined as a transaction by the user that takes a relationship and sets its source and target entities. More formally, a modeling action is a function, action(X, R, E), that takes a model element X, a relationship of the element, R, and produces an endpoint for that relationship E.

The goal of a traditional MDD tool is to take the input produced by the user, such as mouse clicks, and translate them into the values for X, R, and E to update the model. With a traditional MDD tool, the values for E are explicitly bound by modelers. A MDD tool integrated with a constraint solver not only provides this traditional explicit binding capability but also provides a constraint solver binding process, in which the constraint solver deduces the proper endpoints for relationships on behalf of the modeler.

The GEF and EMF frameworks can be used to illustrate how X, R, and E are actually implemented in a modeling framework. GEF provides an MVC framework for displaying and editing EMF models. In GEF, each possible user action, such as connecting two elements with a line in the graphical model, is represented with a Command object. The command object is a part of the Command Pattern (Gamma, 1995), which encapsulates

actions that can affect a model in an object. When the user clicks on an element and then presses the delete key, GEF constructs a DeleteCommand, sets the command's argument to be the element that was click on, and then calls the command's execute() method, which deletes the element from the EMF model. When the user wishes to create a connection, the user selects the connection tool from a tool palette. Selecting the connection tool causes GEF to construct a ConnectionCommand. When the user clicks on the first element for the connection, GEF passes the element to the ConnectionCommand as the source argument. When the user clicks on the endpoint for the connection, GEF passes the command the endpoint as the target argument and calls the command's execute() method, which creates the connection between the two elements. Tool implementers create Command objects to specify how each possible user action is translated into changes of the underlying EMF model.

With GEF's command pattern, R is determined by the type of Command object that GEF instantiates. In the deployment example, when the user selects the DeploymentConnection tool, GEF creates a corresponding DeploymentConnectionCommand object. The Command knows (because it is coded into the command object's execute method) that it is modifying the TargetECU relationship of its source argument. The command also knows that its source argument is the X variable in the action(X,R,E) function. Finally, the command knows that its target endpoint represents the E variable. Each Command object is used to translate a graphical user action (e.g. adding a connection) into values for X, R, and E. The Command is also responsible for modifying the R relationship between X and E in its execute method. The execute() method of a DeploymentConnectionCommand is shown in the Java code below:

```
public class DeploymentConnectionCommand extends Command{
  ....
  //apply action(X,R,E)
  public void execute() {
   Component source = (Component)this.getSource(); //the X
   ECU target = (ECU)this.getTarget(); //the E

   //the R relationship (targetECU) between X and E is set here
```

```
    source.setTargetECU(target);
  }
}
```

In the modified binding process for E, each relationship R is associated with a CSP specifying what is considered a correct value for E. For example, a component could specify that a correct value for its TargetECU's E value requires that the chosen E value and the component both have the same OS type. When a user input is translated into values for X and R, a constraint solver integrated MDD tool uses the CSP associated with R to automatically derive values for E on behalf of the user. In Figure VII.2, the CSP was found in step 2, the values for X and R were produced in step 4 and the bindings for E were delivered by the constraint solver in step 5. The modified modeling transaction process can be seen in Figure VII.3.

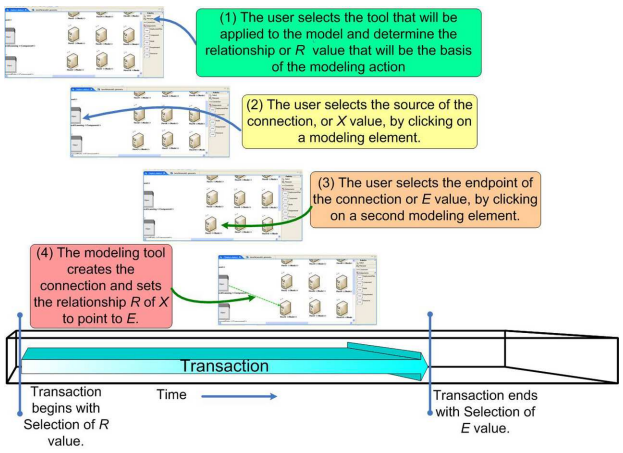


Figure VII.3: A Diagram of a Modeling Transaction with a Constraint Solver

In the first step, the user selects a tool or action that will be applied to the model. The tool determines the R value or relationship that will be modified by the user's actions. In the second step, the user clicks on a modeling element to initiate a connection and hence modify a relationship in the underlying model. The element that the user clicks on becomes the X value that will be passed to the constraint solver. In the third step, the modeling environment looks up the correct CSP that must be satisfied by the endpoints of

the relationship specified by the R value. The modeling environment then passes this CSP, the X, and R values to the solver. The solver finds the endpoints that satisfy the CSP and returns these endpoints as possible E values. Finally, the E values are presented graphically to the user.

The GEF DeploymentConnectionCommand can be modified to incorporate this new process by which the constraint solver chooses the value for E. The Command creation and initial argument setting remains unchanged. However, after the source of the connection has been set, the constraint solver can be invoked to solve for a value for E. If a value is returned, the execute() method can be called immediately. The new DeploymentConnectionCommand is:

```
public class DeploymentConnectionCommand extends Command{
  ....
  public void setSource(Object obj) {
     this.source = obj;

     //the X
     Component source = (Component)obj;

     //call the solver to find valid values for E
     List endpoints = this.solver.findEndpoints(source.getId(),
                                                "targetECU");

     //if there is only one possible value, go ahead and execute
     if(endpoints.size() == 1){
        setTarget(endpoints.get(0));
        execute();
     }
     else if(endpoints.size() > 0) {
        //otherwise, show the user valid E values by
        //modifying their background color
        for(Object obj : endpoints)
           ((ECU)obj).setBackgroundColor(Color.yellow);
     }
else {
   //notify the user that there are no
   //possible deployment locations for the Component
   source.setBackgroundColor(Color.red);
}
  }

  //apply action(X,R,E)
  public void execute() {
   Component source = (Component)this.getSource(); //the X
   ECU target = (ECU)this.getTarget(); //the E
```

```
    //the R relationship (targetECU) between X and E is set here
    source.setTargetECU(target);
  }
 }
```

In the modified DeploymentConnectionCommand, immediately after GEF sets the source of the connection, the command invokes the constraint solver to find valid endpoints. If exactly one endpoint is found, the setTarget method is called with that endpoint and the Command is executed. If more than one valid endpoint is found, each valid target has its background color changed to yellow (a visual queue). If there is no possible deployment location for the Component, its background color is changed to red.

In a traditional process, the user would be required to click first on the source element, decide on a valid deployment location for the source, and then click on the deployment location. With the modified Command object, the object itself attempts to determine the valid targets (E) using the constraint solver. The Command can then either automatically complete the action on the user's behalf, if there is exactly one possible endpoint. If there is more than one possible endpoint, the Command can highlight those endpoints for the user. If no endpoints are found, the Command can notify the user by changing the Component's background color to red.

In many situations, the user will wish to find a valid endpoint for a specified R relationship for every member of a set of modeling elements. For example, the user may wish to select some or all of the Components and have the solver find a valid target ECU for every Component such that no global deployment constraint, such as resource consumption, is violated. Using the GEF framework, a new BatchDeploymentCommand can be created.

Just as with other GEF commands, the BatchDeploymentCommand can have a tool palette entry associated with it that the user can select. When the user selects the corresponding tool entry, the BatchDeploymentCommand is created. The batch command takes

a group of modeling elements, which the user specifies through a group selection, and creates a connection for each member of the group to a valid ECU. The Java code for the BatchDeploymentCommand is:

```
public class BatchDeploymentCommand extends Command{
  ....

  public void execute() {
   //the set of Xs
   Component[] sources = (Component[])this.getSources();

   //the solver deduces an E for each X
   Object[] targets = this.solver.findValidTargets(sources,
                                                   "targetECU");

   if(targets != null){
     for(int i = 0; i < targets.length; i++) {
       sources[i].setTargetECU((ECU)targets[i]);
     }
   }
 }
}
```

CHAPTER VIII

AUTOMATED CONFIGURATION HEALING

Introduction

Service-oriented architectures (SOAs) are emerging as a powerful mechanism to provide loose coupling and software reuse in enterprise applications. SOAs expose individual reusable software applications or components as remotely accessible services that communicate using standardized message-oriented protocols, such as the Simple Object Access Protocol (SOAP). The loose coupling provided by message-oriented communication and standardized protocols allows applications to be rapidly composed from both newly developed custom components and from existing services.

Often, within a single organization or group of collaborating organizations, multiple services are available that can accomplish a particular task. The redundancy in services provides the potential to create applications that can heal themselves by failing over to leverage similar services when a service in their service composition (*i.e.* the services used by the application) fails. Failing over to another equivalent but not necessarily identical service can create robust applications that can adapt to service failures and remain functional.

Designing and implementing a mechanism to build self-healing service compositions is a complex endeavor. Since software development projects already have low success rates and high costs, building a service capable of healing is typically not feasible. Furthermore, building adaptive mechanisms greatly increases the complexity of an application and can be difficult to divorce from application code if the development of the adaptive mechanism is not successful.

Model-driven engineering (MDE) provides a potential solution to managing the complexity of developing adaptive services. In an MDE approach, high-level adaptive models

are used to generate the complex adaptive code required to heal the application when services fail. This approach allows much of the complex healing code to be generated by the MDE tool and in many cases, removed in needed. Numerous approaches have been presented for building MDE models and platforms for enterprise applications but these approaches tend to suffer from one or more of the following problems:

1. they require tight-coupling between application code and adaptation logic or frameworks
2. they require significant development effort to explicitly model the numerous potential error states and recovery paths from an error state to a correct state
3. they require extensive effort to develop the adaptation action implementations for a realistic application

In this paper we present an MDE approach and toolset, called *Refresh*, for designing and implementing self-healing service compositions. Refresh is specifically designed for healing a service composition when:

1. the application is implemented with a component-based technology
2. catastrophic failure is imminent
3. the application and any redundant instances in an application cluster cannot continue functioning correctly in their current configuration
4. the application has alternate composable services, that could potentially be exploited to avoid failure

For each potential error state that an application's service composition could enter, most existing MDE adaptation techniques require explicitly modeling both the error state and the numerous actions to transition from the error state to a correct state. For large enterprise

applications, there are usually a significant number of potential error states and complex nuanced considerations (*e.g.* availability of other services, database locks held, transaction states, etc.) that make it very difficult to create a model for service composition healing. Rather than explicitly modeling error states and recovery actions, Refresh uses *Feature Models* to capture the rules for determining what is or is not a correct configuration/error state.

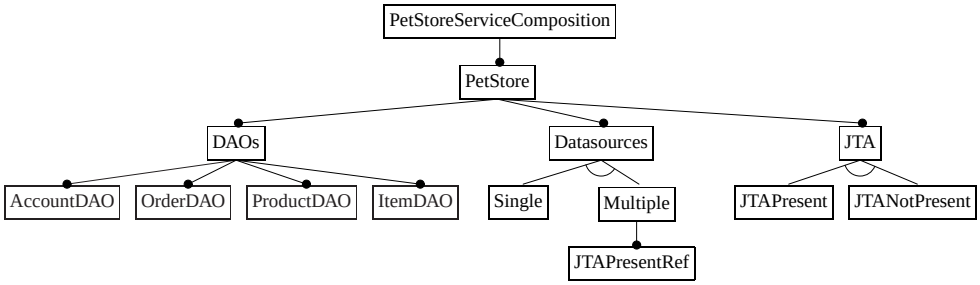


Figure VIII.1: Pet Store Service Composition Feature Model

Feature models describe an application in terms of points of variability and their affect on each other. For example, in an e-commerce application, a feature might be a service for accessing an order database. The order feature can have different sub-features, such as different potential services that can serve as the order database access service. If one particular order database access service is chosen, it excludes the other potential order services from being used (it constrains the other features). If the chosen service fails, a new feature selection can be derived that does not include the failed service's feature.

To avoid the challenges and accidental complexities of both modeling all possible error states and paths to correct states, Refresh uses an approach based on *micro-rebooting* [32]. When a failure, such as the inability to communicate with a dependent service, occurs, Refresh 1) uses the application's feature models to derive a new and valid service composition from the currently available services and components; 2) uses the application's component container to shutdown the failing application subsystem (*e.g.* remote reference to a failed service); 3) and restarts the application subsystem in the newly derived configuration (that

points to a different service and includes any local components needed to communicate with it).

Case Study: The Java Pet Store

To illustrate the complexity of applying existing MDE techniques to creating healing applications, we present a case study based on Sun's Java Pet Store e-commerce application [100]. The Pet Store provides a web-based storefront for selling pets. The store includes multiple catetories of pets, products (*e.g.* Bulldog, Iguana), and individual product items (*e.g.* Female Bulldog Puppy). Customers browse for pets and purchase different items.

Sun and other parties use the Pet Store as a reference application to showcase various frameworks, such as the Java 2 Enterprise Edition frameworks [132]. Because the Pet Store is very widely known and can serve as a reference for comparing different technologies, the Pet Store has been re-implemented in different programming languages and with different frameworks. For example, Microsoft has created the .NET Pet Store [8] and the Java Spring Framework [10, 79] has created the Spring Pet Store. The Spring Framework's version of the Pet Store includes support for integrating web services and is the implementation we have chosen for the case study.

Figure VIII.1, presents a high-level feature model of the features related to the Pet Store's data tier. Features are denoted by the various boxes in the diagram. The levels of hierarchy represent subfeatures. For example, all PetStore instances have *DAOs*, Datasources, and *JTA* as subfeatures (the filled circles at the top of the child features denote required features). The Pet Store Java Transaction API (JTA) feature can either be present, denoted when the child *JTAPresent* feature is selected, or not present. A Feature can also specify rules restricting the selection of other features if the feature is selected. For example, the selection of the *Datasources*/*Multiple* features requires that *JTAPresent* also

be selected. This requirement is denoted by the *JTAPresentRef* required feature reference under *Multiple*.

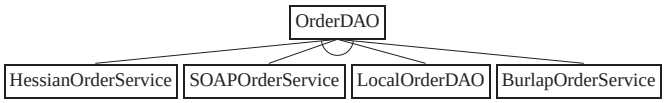


Figure VIII.2: Feature Model of the J2EE Pet Store's OrderDAO

The SpringFramework allows individual components in the Pet Store to be swapped with proxies to remote services. Figure VIII.1 lists the various DAOs that are available in the PetStore. Each of these DAOs can potentially be swapped for a remote service. Figure VIII.2 shows the various options for the OrderDAO. Either the OrderDAO can be implemented by a local component or it can be implemented as a dynamically created Java proxy to a SOAP, Burlap, Hessian, or RMI order service. The case study focuses on failing over from the middle-tier DAOs to different remote services to demonstrate the complexities of applying existing MDE techniques.

Challenges of Creating Self-healing Service Compositions

A very common approach to modeling application healing is to model the individual error states that the application can enter and a recovery path (a sequence of recovery actions) to return the application to a correct state. For example multiple MDE approaches use *State Charts* to capture the various error states of an application and the sequences of recovery actions to return to a correct state. Enumerating each potential error state and each recovery path can require significant modeling complexity. As we will show through the rest of this section, even when an MDE tool can generate the majority of the self-healing code for a service composition, the requirement to model and implement recovery actions places a heavy burden on developers.

Challenge 1: Significant Modeling Complexity to Specify a Recovery Path from an Arbitrary Error State to a Correct State

A healing model must use different error states for each implementation of a service type or component type. The failure of the OrderDAO appears to be a fairly simple error condition to model and specify a recovery path for, but it is not. The problem with modeling each potential error state and recovery path is that the series of recovery actions that need to be invoked is different for the local OrderDAO and remote service implementation. If the local OrderDAO fails, it may simply need to be swapped for another implementation. If a remote service fails, it may be necessary to free resources that were used by a connection to it, such as memory used by caches or network ports.

The type of remote service that is being communicated with can also be important to the recovery action. For example, different recovery paths will be needed to release resources that were used by a connection to a SOAP-based web service as opposed to a Hessian-based web service proxy. Thus, for each type of service or implementation of the OrderDAO, separate error states and recovery paths are needed. Requiring separate error states for each service implementation can cause the number of error states to explode when a real enterprise application is modeled.

If the Pet Store's service composition is modeled using State Charts, as shown in Figure VIII.3, there are 4 different states for each DAO. Futhermore, there are 20 different states needed to represent the potential services and components that can serve as the Pet Store's DAOs. Another property of this model worth noting is that it does not yet include any recovery logic. Instead, the model just includes some placeholder transitions from one potential service to the next.

For every error state that the system needs to recover from, the model must explicitly specify a recovery path. For each of the numerous error states that can be produced, as described above, an individual recovery path must be defined to heal the service composition. For example not only do the failure of a Hessian and SOAP-based order service need

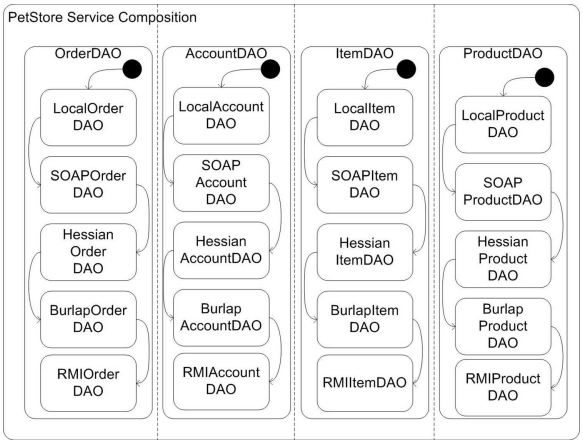


Figure VIII.3: Pet Store Service Composition State Chart

to be modeled separately, but the series of recovery actions attached to each also needs to be modeled separately. As with error states, the number of recovery path specifications produced for healing each component of an enterprise application can be large.

The Pet Store requires a number of recovery actions to take place in order to swap the service used for a DAO. The various actions for swapping the OrderDAO to one of the remote services is modeled in Figure VIII.4. First, to swap a DAO, a Spring `HotSwappableTargetSour` (an object capable of swapping an active component in the application) must be obtained. Next, any resources held by the old DAO implementation or DAO proxy to a remote service must be released. After releasing resources, a new proxy to another remote service can be created. Finally, the newly created proxy can be swapped into the application using the `HotSwappableTargetSource`. Including the recovery paths in the model ups the total number of states per DAO from 4 to 25.

Healing a local error may require evaluating the global application state. In the models thus far, if the OrderDAO fails, it can be replaced with any of the potential viable order services. If the Java Transaction API (JTA) is being used to manage transactions, the Pet Store can fail over to any remote service and still provide proper transaction behavior. If, however, JTA is not being used to manage transactions, the system can only provide transactions across a single datasource, meaning that all of the DAOs must be accessing

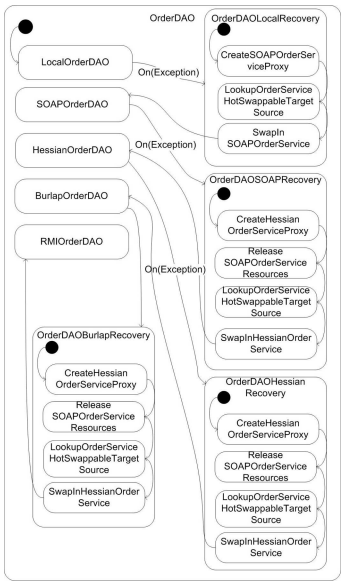


Figure VIII.4: OrderDAO Recovery Paths State Chart

the same database instance. Requiring the use of a single database instance prevents an arbitrary service from being chosen. In the non-JTA situation, the service may only fail over to a remote service if the service is accessing the same database instance as all other referenced remote services.

An extension of the OrderDAO recovery State Chart to include the JTA consideration is show in Figure VIII.5. Each transition to the swap states now includes a guard to ensure that swapping is allowed. A new *GlobalSwapController* has been added to the model to only allow swapping when either JTA is present or a single data source is being referenced by the application's service composition.

Challenge 2: Significant Complexity to Write Re-configuration Code that Can Bring the System from an Arbitrary Error State to a Correct State.

Regardless of the MDE approach used for building the application healing mechanism, developers must always implement the application-specific recovery actions. This requirement parallels the development of enterprise applications and services, where despite the frameworks used, developers are always required to implement the core business logic.

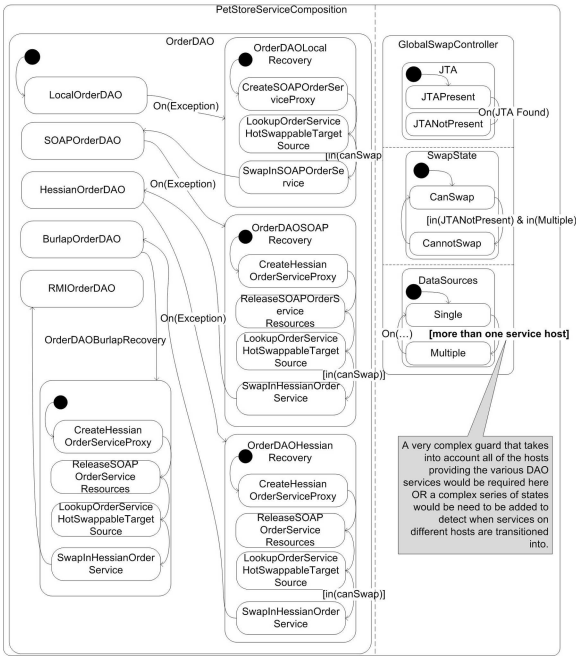


Figure VIII.5: OrderDAO Recovery Paths State Chart when Accounting for JTA

Some specialized MDE tools may provide pre-built recovery actions for very specific domains, but in general, nearly every MDE approach requires developers to write the recovery actions.

For each path from an error state to a recovery state, complex recovery logic must be written. The more error states that are possible in the application, the more recovery actions must be written by developers. These numerous recovery actions can be both expensive to develop and difficult to test - a potential problem when projects are already prone to failure and cost overruns.

In the Pet Store application, there are four separate DAOs that can each be swapped to one of four remote services to avoid failures. To implement a simple swapping mechanism in the Pet Store, the Spring framework provides numerous complex utility classes for hotswapping components and connecting to remote services, such as Apache Axis web services. Despite these numerous utility classes, to create an action to swap just the OrderDAO to one of the four remote services requires 77 lines of Java code to implement the swapping logic and 11 lines of XML code to enable and configure the swapping action in

the Pet Store. Although some level of refactoring and object-oriented design can be used to share common logic across actions, implementing each action still requires significant effort.

Challenge 3: Executing Arbitrary Recovery Actions in Arbitrary Error States can have Numerous Unforeseen Side-effects.

Error states are often specified in such a way that the system as a whole can be in numerous different states that all fall under the definition of the same error state. For example, when the OrderDAO fails, the Pet Store can have orders in progress, category listings in progress, and numerous other nuanced conditions. Building a robust and correct recovery action requires taking into account the side effects of the recovery action on the complex overall state of the application.

For example, what will happen if the local OrderDAO is swapped with a remote service during the submission of one or more customer orders? Can the orders potentially be left in an inconsistent state in the database? Does the safety of the swap depend on whether or not a local or JTA-based transaction mechanism is used? These complex nuanced questions are not easy to answer and must be considered for each recovery action implementation. These intricacies make developing a recovery action that will not lead to unforseen problems hard.

Modeling and Building Healing Adaptations with Refresh

By evaluating the challenges in Sections VIII-VIII, it is apparent that they stem from two causes: 1) the requirement that every error state and recovery path must be explicitly modeled and 2) that developers must implement every complex recovery action. This section describes our MDE toolset, called *Refresh*, that eliminates these two sources of substantial complexity.

Refresh uses feature models to capture the rules for what is a correct system state, which eliminates the need to explicitly model every error state (since each state can be checked

for correctness on-demand). Second, rather than requiring complex recovery actions to be implemented, Refresh uses the application's component container to shutdown the application, reconfigure its service composition, and restart the application in the new and correct state. This reuse of standard container mechanisms for adaptation significantly reduces healing development effort without sacrificing performance.

Overview of Refresh

Refresh is built around the concept of micro-rebooting. When an error is observed in the application, Refresh uses the application's component container to shutdown and reboot the application's components. Using the application container to shutdown the failed subsystem takes milliseconds as opposed to the seconds required for a full application server reboot. Since it is very likely that rebooting in the same configuration (*e.g.* referencing the same failed remote service) will not fix the error, Refresh derives a new application configuration and service composition from the application's feature models that does not contain the failed features (*e.g.* remote services).

The service composition dictates the remote services used by the application. The application configuration determines any local component implementations, such a SOAP messaging classes, needed to communicate and interact properly with the remote services. After deriving the new application configuration and service composition, Refresh uses the application container to reboot the application into the desired configuration. The overall structure of Refresh is shown in Figure VIII.6.

Refresh interacts directly with the application container, as can be seen in Figure VIII.6. During the initial and subsequent container booting processes, Refresh transparently inserts *application probes* into the application to observe the application components. Observations from the application components are sent back to an *event stream processor* that runs queries against the application event data, such as exception events, to identify errors. Whenever an application's service composition needs to be healed, *Environment probes*

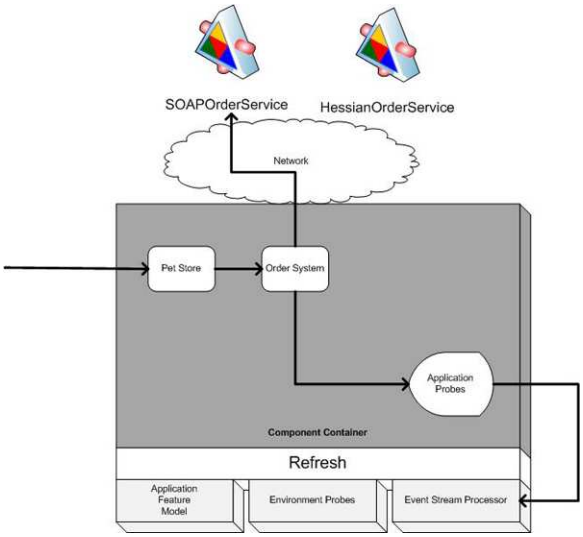


Figure VIII.6: Refresh Structure

are used to determine available remote services and global application constraints, such as whether or not JTA is present. Finally, Refresh includes a *feature model* of the application that dictates the rules for deriving a new application configuration and service composition when the application needs to be healed and rebooted.

Refresh uses event stream processing [91], to run queries against the application's event data and identify feature failures. The initial implementation of Refresh, based on the Spring Frameworks IoC container, uses the Esper event stream processor [4] for Java. Esper is a high-performance event stream processor that is capable of handling 100,000 events a second with 2,000 queries on a single dual-core CPU [3].

Each feature in the feature model that could potentially fail is associated with a group of event stream queries. At runtime, when a query associated with a feature returns a result, Refresh is notified that the associated feature has failed, as shown in Figure VIII.7. The data and objects observed and analyzed by Refresh are determined by the query specifications.

Once Refresh is notified of a feature failure, it has three main tasks: 1) to use the container to shutdown the application's components; 2) to use the application's feature model to derive a new application configuration and service composition; and 3) to use the

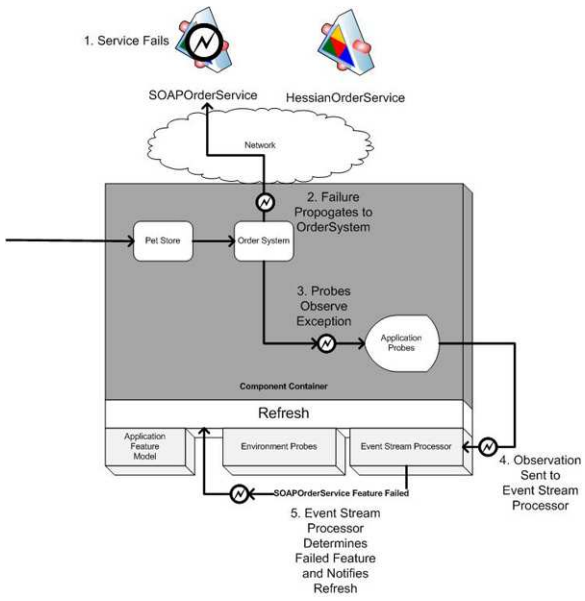


Figure VIII.7: Error Propogation to Refresh

container to reboot the application in the new configuration. The sequence of events from a feature failure notification to the rebooting of the container are shown in Figure VIII.8.

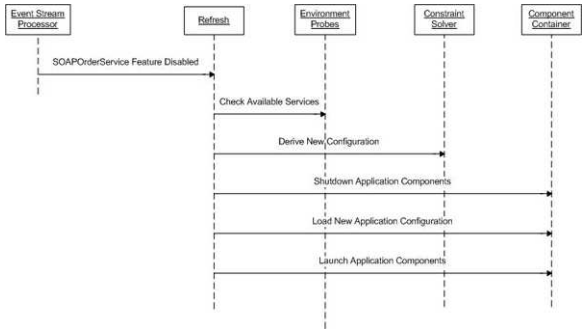


Figure VIII.8: Refresh Reconfiguration, Shutdown, and Launch Recovery Sequence

To derive a new configuration of the application does not include the failed feature, Refresh transforms the feature selection problem into a constraint satisfaction problem (CSP) using techniques that have been developed by us an others in prior work [22, 144, 149]. Once the feature selection problem is transformed into a CSP, a high-performance general purpose constraint solver, such as ILog's JSolver [35], Geocode [124], or Choco [20], is used to derive a new set of features/configuration for the application.

Once the new application configuration and service composition is derived, Refresh invokes the container's shutdown sequence to properly release resources, abort transactions, and perform other critical activities. The new configuration is injected into the container through programmatic calls or by regenerating the application's configuration files [144]. After the configuration is injected into the container, the application is launched in the new configuration without the failed service, as shown in Figure VIII.9.

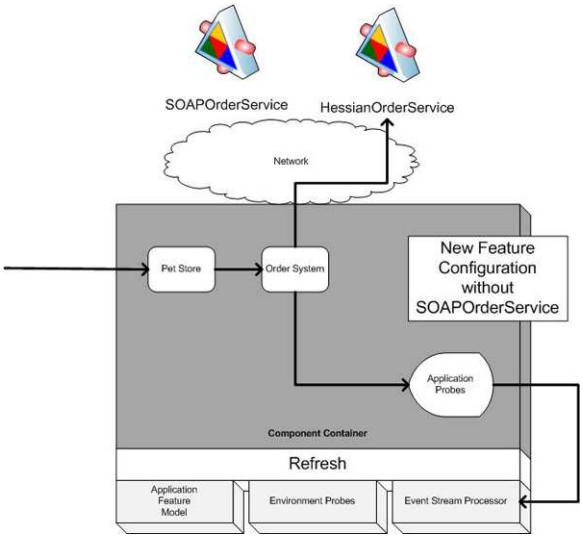


Figure VIII.9: Refresh Launches the Application in the New Configuration

Solution 1: Use Feature Modeling to Capture the Rules for Deriving what is Considered a Correct State

Modeling each individual error state and recovery path is complex. Refresh uses feature modeling to avoid requiring developers to model each individual error state and recovery path. Feature modeling captures the rules–rather than individual error states and recovery paths–for deriving what constitutes a correct application configuration and service composition. In terms of healing, feature modeling describes:

- the component or service types that are needed to compose the application

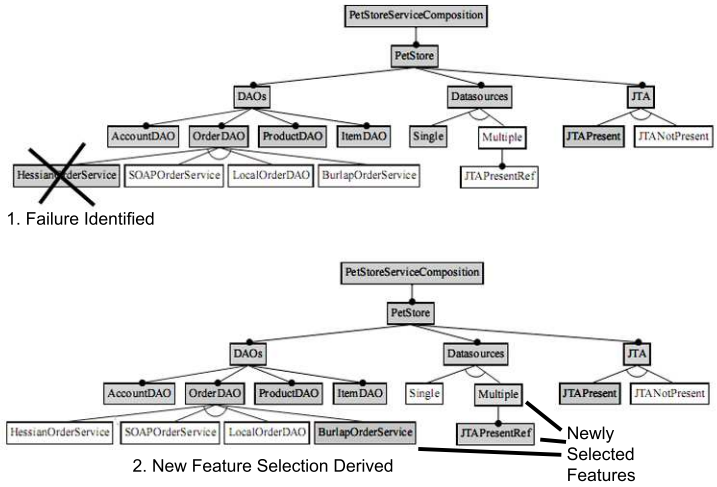


Figure VIII.10: Deriving a new Service Composition from the Pet Store Feature Model

- the sets of components or services that can serve as the implementation of a service type in the application's composition
- the rules dictating the requirements, such as dependent libraries, required by each component or service implementation
- the rules constraining how the choice of one service implementation restricts the choices of other component or service implementations in the same application composition

When the failure of a feature is observed, Refresh uses the feature model of the application to derive an alternate set of features for the application that does not include the failed feature. For example, in the Pet Store, when the *LocalOrderDAO* feature fails, Refresh uses the feature model to derive an alternate feature selection for the Pet Store. In the example shown in Figure VIII.10, Refresh chooses a new feature selection that uses the *HessianOrderDAO* rather than the failed *LocalOrderDAO*.

Automated Feature Selection Using a Constraint Solver: The key to Refresh's healing capabilities is its ability to use a constraint solver to automatically derive a new feature selection for the application. Prior work provides extensive details on the process for transforming a feature selection problem into a constraint satisfaction problem (CSP), which is

the input format of a constraint solver, and deriving a feature selection. In this section, we briefly cover this mapping.

A constraint satisfaction problem is a series of variables and a set of constraints over the variables. For example, "$A + B < C$" is a constraint satisfaction problem over the integer variables *A*, *B*, and *C*. A constraint solver automatically derives a correct labeling (values for the variables). The labeling "$A = 1, B = 2, C = 4$" is a correct labeling of the example CSP.

A selection of features from a feature model can be represented by a set of integer variables with domain 0 or 1. Each variable represents a unique feature from the feature model. If the variable representing the *HessianOrderService* is represented by the variable V_1, then $V_1 = 1$ in a labeling of a feature selection CSP means that the feature is selected in the solution. If the labeling contains $V_1 = 0$, it implies that the feature is not selected in the solution. The configuration of an application and its service composition is represented as a set of these variables that denote which services and application components are enabled in a configuration.

Rules dictating the proper composition of the services are specified as constraints over the V_i variables. For example, since only one of *HessianOrderService* and *SOAPOrderService* can be used at a time by the Pet Store, a constraint can be used to capture this rule. Let, V_2 be the variable representing the *SOAPOrderService*. This rule is specified as the constraint $V_1 = 1 \rightarrow V_2 = 0$. As described in [144], complex rules, such as memory constraints, can be described using a CSP.

When a feature is flagged as failed, Refresh adds a new constraint to the feature selection process preventing the failed feature from being selected (*e.g.*, $V_i = 0$). Refresh then uses a the constraint solver to derive a new feature selection that can be used by the application based on the environmental constraints (*e.g.* JTA vs. No JTA) and feature model composition constraints (*e.g.*, only one of the order services may be selected at a time). When only standard feature modeling rules, like excludes, requires, cardinality, and

attribute values are used to describe constraints, the solver can very quickly produce a correct solution [149]. More complex constraints, such as memory resource constraints, can be added to augment standard feature modeling rules but require more care in the feature model specification process to allow the solver to quickly derive a solution [149].

Eliminating Error State and Recovery Path Modeling Complexity: Because the new feature selection is introduced into the application by shutting down the old references to remote services and launching the new component configuration and service composition, separate recovery actions are not needed. Furthermore, since feature models specify the rules for deriving a correct/incorrect configuration and do not enumerate all possible error configurations, they require significantly fewer modeling elements. The equivalent healing behavior to the 111 state State Chart described earlier can be produced in Refresh using a feature model with 33 features –a roughly 70% reduction in total model elements. The feature models also have 33 connections versus the 102 connections for the State Chart.

Reusing the Component Container's Shutdown/Configuration/Launch Mechanisms for State Transitions

Sections VIII-VIII illustrated the complexity and large development burden of writing recovery actions to heal an application by failing over to alternate services. Refresh attacks the problem with a combination of code reuse and automation. Refresh reuses an application container's ability to shutdown an application's components, reconfigure the components (*i.e.* create the newly desired service composition), and launch the application in the new state (*i.e.* transition the application into the new service composition state). By reusing existing mechanisms that are well-tested and trusted by developers, the need to write custom recovery actions is eliminated.

Second, since rebooting in the same application configuration with the same service composition is unlikely to fix a failed reference to a service, Refresh automatically derives a new and valid application configuration and service composition. This automated approach

to deriving a new service composition from an application's feature model allows micro-rebooting to be applied to service composition healing. Normally, with a manual recovery action implementation process, developers would deduce the correct states to transition the application into and implement the transition logic. Refresh's automated derivation process eliminates the need for a developer to: 1) determine where to transition to, 2) decide how to accomplish the transition, and 3) implement the transition.

Container Rebooting-based Healing Reduces Potential Unintended Side-effects: A key benefit of using the container's built in component management mechanisms for state transitions is that they are guaranteed to bring the non-persistent application state to the desired correct state. This guarantee helps to resolve the problems outlined earler of having to deal with the potential of unintended side-effects from recovery actions.

With Refresh, the application container shuts down components, which releases resources and resets in-memory state, and then re-launches the application with a clean memory state. With recovery actions, there is the potential that one or more of the affects on the application will have unforeseen consequences to the non-persistent in-memory application state. These unforseen side-effects are not possible with a container rebooting approach that resets non-persistent state.

A container rebooting approach does not eliminate the possibility that persistent application state, such as database rows, will not be placed into an inconsistent state. The approach does, however, have a number of properties that make this scenario far less likely than a recovery action approach. First, all components typically *must* implement lifecycle methods that are called by the container to manage the component. If a component does not properly handle persistent state on shutdown, it is a flaw in the implementation of the component that could emerge–even if the application never uses healing mechanisms.

Second, most enterprise applications maintain the consistency of persistent application state through transactions. Furthermore, most enterprise applications use container-managed persistence APIs, such as JTA. Even the Non-JTA examples provided for the Pet

Store still use an alternate container-managed persistence API that works across only a single datasource. When the container is used to as the healing transition mechanism, any transactions that are in process will be properly rolled back or committed by the container during the healing of the application's service composition.

Initial Implementation	Manual	MDE / error state / recovery path	Refresh
Modeling			
Modeled States or Features	0	111	33
Modeled Connections/Transitions	0	104	29
Model Error Identification	0	0	23
Modeling Totals	0	215	85
Implementation			
Implement Recovery Actions	77	77	0
Implement Recovery Path Chooser	31	0	0
Configuration Modifications	96	44	67
Implementation Totals	204	121	67

Figure VIII.11: Comparing Implementation Effort for the Healing Pet Store

Applying Refresh to the Java Pet Store

To compare the development effort of including recovery actions into the Pet Store, we implemented three versions of the Spring Pet Store that provided the ability to swap failed DAOs with remote services and to swap from failed remote services to other remote services (the modifications for the three implementations are available from [143]). One implementation was produced using a purely manual approach that used Spring's proxying and aspect infrastructure to implement the monitoring of the DAOs and Spring *HotSwappableTargetSources* to swap remote services on-the-fly. The second implementation was produced assuming an MDE tool was provided that could model the error states and recovery actions for the Pet Store and generate the required monitoring code and recovery path logic but not the implementations of the recovery actions. We refer to this MDE approach as the *MDE error state/recovery path* approach. Finally, a third implementation was produced using Refresh.

Manual Implementation: The top table in Figure VIII.11 shows the results of the initial implementation efforts. The manual approach required implementing two key classes a *ServiceSwapper* capable of 1) looking up the Spring HotSwappableTargetSource for a DAO; 2) connecting to a Hessian, Burlap, SOAP, or RMI remote service; and 3) swapping in the new service for the failed component/service. As is shown in the results figure, the class required 77 lines of code. The second class implemented was a Spring MethodInterceptor that was used to monitor each invocation on a DAO or remote service for Exceptions and call the appropriate ServiceSwapper when an Exception occurred. This class required 20 lines of code. Finally, the components were included in the Pet Store by adding them to the XML configuration files for the Pet Store, which required adding 96 lines of XML code.

MDE Error State / Recovery Path Implementation: The analysis for the MDE error state/recovery path approach was based on a generic model of the minimum effort that would be required for any MDE adaptation modeling tool and framework that did not provide Spring-specific recovery action implementations. The models were built using State Charts, since it is arguably the most widely used and mature state modeling language. State Charts also have a number of powerful concepts, such as parallel states, which reduce the total modeling complexity.

For the MDE implementation effort analysis, we measured only the lines of code required to implement the ServiceSwapper and to integrate the needed ServiceSwappers into the configuration files of the Pet Store. We assumed that all of the logic for choosing the correct ServiceSwapper to execute, the implementation of the MethodInterceptor, and all configuration code required to integrate the method interceptors and their dependent proxies into the configuration file would be generated by the tool. Thus, our experiments were measuring only the cost of modeling error states and recovery actions and implementing them.

The MDE error state/recovery action approach used the State Charts presented earlier.

The full State Chart healing specification requires 111 states and 102 transitions between states. As can be seen in Figure VIII.11, the MDE approach still requires 77 lines of code to implement the ServiceSwapper recovery action but eliminates the 31 lines of code needed to implement the recovery path execution logic and the 20 lines of code required for the monitoring implementation. Furthermore, an MDE approach reduces the lines of XML configuration code that must be added from 96 to 44.

Refresh Implementation: Finally, we implemented the swapping capabilties in the Pet Store using Refresh. Refresh's use of Feature models required a total of 33 model elements (features) and 29 connections versus the MDE approach's 111 model elements (states) and 102 connections (transitions). Refresh also required 16 lines of code to specify the Esper queries over the event stream of the Pet Store to map queries to the failure of one of the Pet Store features. Refresh's use of the container's built-in shutdown/configuration/launch mechanisms for healing, eliminated the need to implement the code for the ServiceSwapper.

Refresh automatically generates the required monitoring code for the Pet Store (this was assumed for the other MDE approach as well). Refresh did require 23 more lines of code to be modified in the configuration file of the Pet Store versus the other MDE approach. These extra lines of configuration code are a result of adding the Refresh annotations dictating how to dynamically modify the application's configuration based on a feature selection. Overall, the Refresh approach required 55% less implementation effort than the other MDE approach and 60% less modeling effort.

Refresh Performance: We used Apache JMeter to simulate the concurrent access of 40 different customers to the Pet Store and the time required to complete 4,000 orders. We simulated the failure of different DAOs to force Refresh to heal the Pet Store by swapping remote services for the failed DAOs. After the DAOs were swapped to remote services, we iteratively shutdown the services used by the Pet Store to force the failover to alternate

remote services. Over the tests, Refresh averaged 151ms from the time an exception indicating a failure was observed until the Pet Store was reconfigured and rebooted with a new service composition. These times were obtained by running the Pet Store on a 2.0ghz Intel Core DUO on Windows XP with 2 gigabytes of RAM. The average time required by the constraint solver to derive a new feature selection was 12ms. These times indicate that Refresh can provide high-performance application healing while reducing modeling and implementation effort.

CHAPTER IX

SCALING CONFIGURATION AUTOMATION TO LARGE MODELS

This chapter focuses on the challenges associated with selecting feature sets subject to resource constraints. For these types of configuration problems, current exact techniques, such as CSP solvers, do not work. This chapter presents an approach for using heuristic knapsack algorithms to automate feature model configuration.

Introduction

Choosing the correct set of architectural features for an application is hard because even small numbers of design variables (*i.e.*, small feature sets) can produce an exponential number of design permutations. For example, the relatively simple feature model shown in Figure IX.2, contains 30 features that can be combined into 300 different distinct architectures. Requirement specifications often try to meet certain goals, such as maximizing face recognition accuracy, that further complicates architectural feature choices.

Resource constraints, such as the maximum available memory or total budget for a system, also add significant complexity to the architectural design process. Finding an optimal architectural variant that adheres to both the feature model constraints and a system's resource constraints is an NP-hard problem [42]. The manual processes commonly used to select architectural feature sets scale poorly for NP-hard problems.

For large-scale systems—or in domains where optimization is critical—algorithmic techniques are needed to help product-line engineers make informed architectural feature selections. For example, developers can choose the features that are deemed critical for the system or driven by physical concerns that are hard to quantify (such as camera types and their arrangement). An algorithmic technique can then be used to make the remaining architectural feature selections that maximize accuracy while not exceeding the remaining

budgetary allocation. Moreover, developers may want to evaluate tradeoffs in architectures, *e.g.*, use a specific camera setup that minimizes memory consumption as opposed to maximizing accuracy.

Existing algorithmic techniques for aiding developers in the selection of architectural variants rely on exact methods, such as integer programming, that exhibit exponential time complexity and poor scalability. Since industrial-size architectural feature models can contain thousands of features, these exact techniques are impractical for providing algorithmic architectural design guidance, such as automated architectural feature selection optimization. With existing techniques, automated feature selection can take hours, days, or longer depending on the problem size. For large problem sizes, this slow solving time makes it hard for developers to evaluate highly optimized design variations rapidly.

This chapter presents a polynomial time approximation algorithm, called *Filtered Cartesian Flattening*, that can be used to derive an optimal architectural variant subject to resource constraints. Using Filtered Cartesian Flattening, developers can quickly derive and evaluate different architectural variants that both optimize varying system capabilities and honor resource limitations. Moreover, each architectural variant can be derived in seconds as opposed to the days, hours, or longer that would be required with an exact technique, thereby allowing the evaluation of more architectural variants in a shorter time frame.

This chapter provides the following contributions to the study of applying the Filtered Cartesian Flattening algorithm to assist developers in selecting SPL architectural variants:

1. We prove that optimally selecting architectural feature sets that adhere to resource constraints is an NP-hard problem.

2. We present a polynomial time approximation algorithm for optimizing the selection of architectural variants subject to resource constraints.

3. We show how any arbitrary Multi-dimensional Multiple-choice Knapsack (MMKP) algorithm [103,114,128] can be used as the final step in Filtered Cartesian Flattening,

which allows for fine-grained control of tradeoffs between solution optimality and solving speed.

4. We present empirical results from experiments performed on over 500,000 feature model instances that show how Filtered Cartesian Flattening averages 92.56%+ optimality on feature models with 1,000 to 10,000 features.

5. We provide metrics that can be used to examine an architectural feature selection problem instance and determine if Filtered Cartesian Flattening should be applied.

Overview of Feature Modeling

Feature modeling [82] is a modeling technique that describes the variability in an SPL architecture with a set of architectural features arranged in a tree structure. Each architectural feature represents an increment in functionality or variation in the product architecture. For example, Figure IX.1 shows a feature model describing the algorithmic variability in a system for identifying faces [113] in images. Each box represents a feature. For example, Linear Discriminant Analysis (LDA) is an algorithm [112] for recognizing faces in images that is captured in the LDA feature.

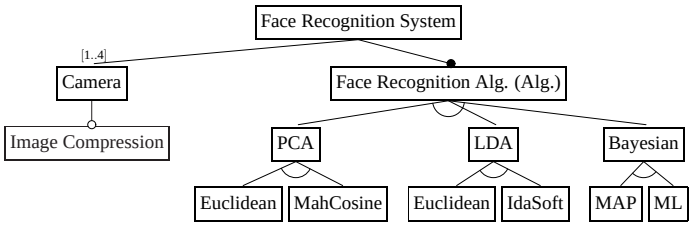


Figure IX.1: Example Feature Model

A feature can (1) capture high-level variability, such as variations in end-user functionality, or (2) document low-level variabilities, such as *software variability* (*e.g.*, variations in software implementation) [98]. Each complete architectural variant of the SPL is described as a set of selected features. For example, the feature model in Figure IX.1 shows

how the feature set [Face Recognition System, Camera, Face Recognition Alg, PCA, MahCosine] would constitute a complete and correct feature selection.

The constraints on what constitutes a valid feature selection are specified by the parent child relationships in the tree. Every correct feature selection must include the root feature of the tree. Moreover, if a feature is selected, the feature's parent must also be selected. A feature can have required sub-features indicating refinements to the feature. For example, Face Recognition System has a required sub-feature called Face Recognition Alg. that must also be selected if Face Recognition System is selected. The required relationship is denoted by the filled oval above Face Recognition Alg..

The parent child relationships can indicate variation points in the SPL architecture. For example, LDA requires the selection of either of its Euclidean or IdaSoft sub-features, but not both. The Euclidean and IdaSoft features form an exclusive-or subgroup, called an *XOR group*, of the Linear Discriminant Analysis (LDA) feature that allows the selection of only one of the two children. The exclusive-or is denoted with the arc crossing over the connections between Euclidean, IdaSoft, and their parent feature. Child features may also participate in a *Cardinality group*, where any correct selection of the child features must satisfy a cardinality expression.

Feature models can also specify a cardinality on the selection of a sub-feature. For example, at least 1 and at most 4 instances of the Camera feature must be selected. An unfilled oval above a feature indicates a completely optional sub-feature. For example, a camera can optionally employ Image Compression. Finally, a feature can refer to another feature that it requires or excludes that is not a direct parent or child. These constraints are called cross-tree constraints.

Motivating Example

A key need with SPL architectures is determining how to select a good set of architectural features for a requirement set. For example, given a face recognition system that includes a variety of potential camera types, face recognition algorithms, image formats, and camera zoom capabilities, what is the most accurate possible system that can be constructed with a given budget? The challenge is that with hundreds or thousands of architectural features—and a vastly larger number of architectural permutations—it is hard to analyze the resource consumption and accuracy tradeoffs between different feature selections to find an optimal architectural variant.

Motivating Example

As a motivating example of the complexity of determining the best set of architectural features for a requirement set, we provide a more detailed example of the face recognition system for identifying known cheaters in a casino. A small example feature model of the face recognition system's architectural features is shown in Figure IX.2. The system can

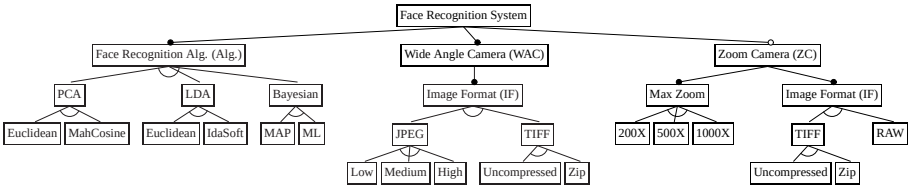


Figure IX.2: Face Recognition System Arch. Feature Model

leverage a variety of algorithms ranging from versions of Linear Discriminant Analysis (LDA) to Bayesian networks. The system requires a wide angle camera, but can be supplemented with a zoom camera to provide closer images of specific faces in the environment. Each camera can produce images in a variety of image formats ranging from lossy low quality JPEG images to lossless RAW images from the camera's CCD sensor.

Each variability point in the architecture, such as the type of face recognition algorithm,

affects the overall accuracy and resource consumption of the system. For example, when higher resolution images are obtained by a camera, the overall accuracy of the system can improve. Higher resolution images, however, consume more memory and require more CPU time to process. Depending on the overall system requirements, therefore, choosing higher resolution images to improve accuracy may or may not be possible, depending on the available memory and the memory consumed by other features.

Table 1 captures example information on the accuracy provided—and resources consumed—by some of the architectural features. Each feature is identified by the path through the feature model to reach the feature. For example, the high resolution JPEGs feature is identified by `WAC/IF/JPEG/High`. The choice of architectural features is governed by the overall goal of the system. In this example, we want to maximize face recognition accuracy without exceeding the available memory, CPU, development budget, or development staff. Our architectural goal and resource limits are shown in Table 2.

Arch. Feature	Accuracy	CPU	Memory	Cost	Devel. Staff
WAC/IF/JPEG/High	0.10	8	1024	2	0
WAC/IF/JPEG/Low	0.03	2	128	2	0
...					
ZC/IF/TIFF/Zip	0.13	16	256	30	1
...					
Alg/LDA/Euclidean	0.85	112	2048	300	1
Alg/LDA/IdaSoft	0.84	97	1024	120	0

Table IX.1: Software Feature Resource Consumption, Cost, and Accuracy

Table 2 lists the architectural resource constraints and goal for the design of the system. The first column lists the goal, which is to maximize the accuracy of the system. Each subsequent column lists a resource, such as total system memory, and the amount of that resource that is available for an architectural variant's features to consume.

Accuracy	CPU	Memory	Cost	Devel. Staff
Maximize	≤ 114	≤ 4096	≤ 330	≤ 1

Table IX.2: Example Architectural Requirements: Maximize Accuracy Subject to Resource Constraints

Challenges of Feature Selection Problems with Resource Constraints

To make well-informed architectural decisions, developers need the ability to easily create and evaluate different architecture variations tuned to maximize or minimize specific system capabilities, such as minimizing total cost or required memory. Generating and evaluating a range of architectures allows developers to gain insights into not only what architectural variants optimize a particular system concern, but also other design aspects, such as patterns that tend to lead to more or less optimal variants. The chief barrier to creating and evaluating a large set of optimized architectural feature models is that generating highly optimized variants is computationally complex and time consuming.

Optimally selecting a set of architectural features subject to a set of resource constraints is challenging because it is an NP-hard problem. To help understand why optimal feature selection problems with resource constraints is NP-hard, we first need to formally define these problems. An architectural feature selection problem with resource constraints is a five-tuple composed of a set of features (F), a set of dependency constraints on the features (C) defined by the arcs in the feature model graph, a function (Fr(i,j))that computes the amount of the j_{th} resource consumed by the i_{th} feature, a set of values or benefits associated with each feature (Fv), and a list of the resource limits for the system (R):

$$P =< F, C, Fr(i, j), Fv, R >$$

The features (F) correspond to the the feature nodes in the feature model graph shown in Figure IX.2, such as `Bayesian` and `LDA`. The dependency constraints (C) correspond to the arcs connecting the feature nodes, such as `Face Recognition Alg` is a required sub-feature of `Facial Recognition System`. The resource consumption function

(Fr) corresponds to the values in columns 3-6 of Table 1, such as the amount of memory consumed by each feature. The feature values set (Fv) corresponds to the accuracy column in Table 1. Finally, the resource limits set (R) corresponds to the resource limits captured in columns 2-4 of Table 2.

We define the solution space to a feature selection problem with resource constraints as a set of binary strings (S) where for any binary string ($s \subset S$) the i_{th} position is 1 if the i_{th} feature in F is selected and 0 otherwise. The subset of these solutions that are valid ($Vs \subset S$) is the set of solutions that satisfy all of the feature model constraints (1) and adhere to the resource limits (2):

$$Vs = \{s \subset S|$$

$$s \rightarrow C, \quad (1)$$

$$\forall j \subset R, (\sum_{i=0}^{n} s_i * Fr(i,j)) \leq R_j\} \quad (2)$$

To prove that optimally selecting a set of architectural features subject to resource constraints is NP-hard, we show below how any instance of an NP-complete problem, the Multi-dimensional Multiple-choice Knapsack Problem (MMKP), can be reduced to an instance of this definition of the optimal feature selection problem with resource constraints.

A traditional knapsack problem is defined as a set of items with varying sizes and values that we would like to put into a knapsack of limited size. The goal is to choose the optimal set of items that fits into the knapsack while maximizing the sum of the items' values. An MMKP problem is a variation on the traditional knapsack problem where the items are divided into sets and at most one item from each set must be placed into the knapsack. The goal remains the same, *i.e.*, to maximize the sum of the items' values in the knapsack.

We provide a simple example of transforming an MMKP problem into a feature selection problem with resource constraints. Figure X.2 shows a simple MMKP problem with six items divided into two sets. At most one one of the items A, B, and C can be in the knapsack at a given time. Moreover, at most one of the items D, E, and F can be in the sack.

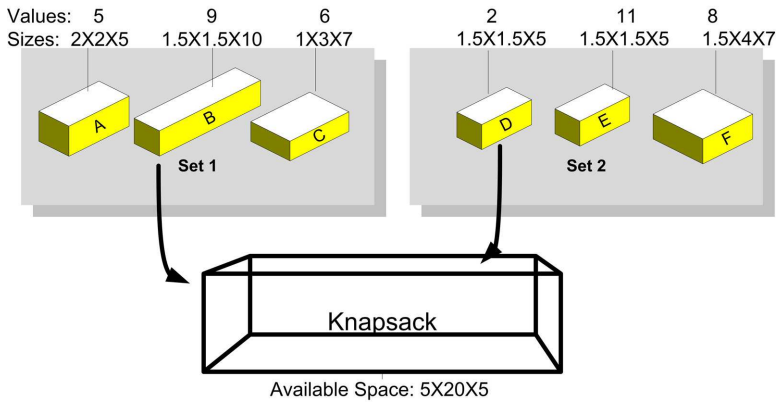


Figure IX.3: A Multi-dimensional Multiple-choice Knapsack Problem

To transform the MMKP problem into a feature selection problem with resource constraints, we create a feature model to represent the possible solutions to the MMKP problem, as shown in Figure IX.4. The generalized algorithm for converting an instance of an

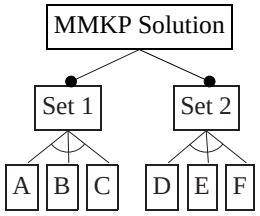


Figure IX.4: A Feature Model of an MMKP Problem Instance

MMKP problem into an equivalent feature selection problem with resource constraints is as follows:

1. Create a root feature denoting the MMKP solution,
2. For each set, create a mandatory sub-feature of the root feature,
3. For each set, add an XOR group of sub-features corresponding to the items in the set,
4. For each item, initialize its feature's resource consumption value entries in the feature properties table to the length, width, and height of the item,

5. For each item, initialize its feature's value entry in the feature properties table, shown in Table 3, to the item's value, and

6. Set the total available resources to be the length, width, and height of the knapsack.

Feature	Value	Resource 1 (length)	Resource 2 (width)	Resource 3 (height)
A	5	2	2	5
B	9	1.5	1.5	10
C	6	1	3	7
D	2	1.5	1.5	5
E	11	1.5	1.5	5
F	8	1.5	4	7

Table IX.3: MMKP Feature Properties Table

Steps 1&2 define the sets (F) and (C) for our feature selection problem. Step 3 creates a table, shown in Table 3, that can be used to define the function ($Fr(i, j)$) to calculate the amount of each resource consumed by a feature. Step 4 initializes the set of values (Fv) defining the value associated with selecting a feature. Finally, Step 5 creates the set of available resources (R).

With this generalized algorithm, we can translate any instance of an MMKP problem into an equivalent feature selection problem with resource constraints. Since any instance of an MMKP problem can be reduced to an equivalent feature selection problem with resource constraints, then feature selection problems with resource constraints must be NP-hard. Any exact algorithm for solving feature selection with resource constraints will thus have exponential time complexity.

Filtered Cartesian Flattening

This section presents the Filtered Cartesian Flattening (Filtered Cartesian Flattening) approximation technique for optimal feature selection subject to resource constraints. Filtered Cartesian Flattening transforms an optimal feature selection problem with resource constraints into an approximately equivalent MMKP problem, which is then solved using an MMKP approximation algorithm. The MMKP problem is designed such that any correct answer to the MMKP problem is also a correct solution to the feature selection problem (but not necessarily vice-versa). Filtered Cartesian Flattening allows developers to generate highly optimal architectural variants algorithmically in polynomial-time (roughly $\sim$10s for 10,000 features), rather than in the exponential time of exact algorithmic techniques, such as integer programming.

As shown below, Filtered Cartesian Flattening addresses the main challenge, *i.e.*, the difficulty of selecting a highly optimal feature selection in a short amount of time. The key to Filtered Cartesian Flattening's short solving times is that it is a polynomial time approximation algorithm that trades off some solution optimality for solving speed and scalability.

The Filtered Cartesian Flattening algorithm, which we will describe in the following subsections, is listed in the APPENDIX.

Step 1: Cutting the Feature Model Graph

The first step in Filtered Cartesian Flattening, detailed in code listing (2) of the APPENDIX, is to begin the process of producing a number of independent MMKP sets. We define a choice point as a place in an architectural feature model where a configuration decision must be made (*e*.g., XOR Group, Optional Feature, etc.). A choice point, *A*, is independent of another choice point, *B*, if the value chosen for choice point *A* does not affect the value chosen for choice point *B*. An MMKP problem must be stated so that the choice of an item from one set does not affect the choice of item in another set.

For example, the choice point containing `Image Compression` in Figure IX.1 is independent of the choice point containing `MAP` and `ML`, *i.e.*, whether or not image compression is enabled does not affect the type of `Bayesian` algorithm chosen. The choice point of the type of face recognition algorithm, which contains the feature `Bayesian`, is not independent of the choice point for the type of Bayesian algorithm (*e.g.*, the XOR group with `MAP` and `ML`).

Filtered Cartesian Flattening groups choice points into sets that must be independent. Each group will eventually produce one MMKP set. Starting from the root, a depth-first search is performed to find each optional feature that has no ancestors that are choice points. A cut is performed at each of these optional features with no choice point ancestors to produce a new independent sub-tree, as shown in Figure IX.5. After these cuts are made, if the sub-trees have cross-tree constraints, they may not yet be completely independent. These cross-tree constraints are eliminated in Step 4.

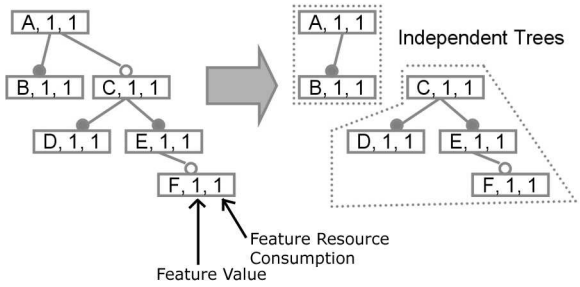


Figure IX.5: Cutting to Create Independent Sub-trees

Step 2: Converting to XOR

Each MMKP set forms an XOR group of elements. Since MMKP does not support any other relationship operators, such as cardinality, we must convert the configuration solution space captured in each feature model sub-tree into an equivalent representation as a series of partial configurations related through XOR. Since a feature model allows hierarchical modeling and cardinality constraints, the conversion to XOR can require an exponential

number of partial configurations for the XOR representation.[1] The filtering process of Filtered Cartesian Flattening is an approximation step that puts a polynomial bound on the number of configuration permutations that are encoded into the XOR representation to avoid this state explosion.

The first step in converting to XOR is to convert all Cardinality groups and optional features into XOR groups. Cardinality groups are converted to XOR by replacing the cardinality group with an XOR group containing all possible combinations of the cardinality group's elements that satisfy the cardinality expression. Since this conversion could create an exponential number of elements, we bound the maximum number of elements that are generated to a constant number K. Rather than requiring exponential time, therefore, the conversion can be performed in constant time.

The conversion of cardinality groups is one of the first steps where approximation occurs. We define a filtering operation that chooses which K elements from the possible combinations of the cardinality group's elements to add to the XOR group. All other elements are thrown away.

Any number of potential filtering options can be used. Our experiments evaluated a number of filtering strategies, such as choosing the K highest valued items, a random group of K items, and a group of K items evenly distributed across the items's range of resource consumptions. The best results occurred when selecting the K items with the best ratio of $\frac{Value}{\sqrt{\sum rc_i^2}}$, where rc_i is the amount of the i_{th} resource consumed by the partial configuration. This sorting criteria has been used successfully by other MMKP algorithms [11]. An example conversion with $K = 3$ and random selection of items is shown in Figure IX.6.

Individual features with cardinality expressions attached them are converted to XOR using the same method. The feature is considered as a Cardinality group containing M copies of the feature, where M is the upper bound on the cardinality expression (*e.g.* [$L..M$] or [M]). The conversion then proceeds identically to cardinality groups.

[1] This state explosion is similar to what happens when a State Chart with hierarchy is converted to its equivalent Finite State Machine representation [67].

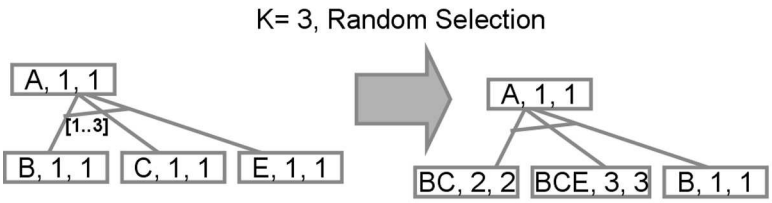


Figure IX.6: Converting a Cardinality Group to an XOR Group with K=3 and Random Selection

Optional features are converted to XOR groups by replacing the optional feature O with a new required feature O'. O' in turn, has two child features, O and $\emptyset$ forming an XOR group. O' and $\emptyset$ have zero weight and value. An example conversion is shown in Figure IX.7.

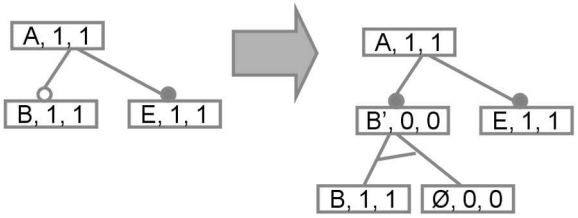


Figure IX.7: Converting an Optional Feature into an XOR Group

Step 3: Flattening with Filtered Cartesian Products

For each independent sub-tree of features that now only have XOR and required relationships, an MMKP set needs to be produced. Each MMKP set needs to consist of a number of partial configurations that could be produced from each sub-tree. To create the partial configurations that constitute each MMKP set, we perform a series of recursive flattening steps using filtered Cartesian products, as shown in code listing (4) in the APPENDIX.

The procedure `flatten` takes a feature and recursively flattens its children into a MMKP set that is returned as a list. The list is constructed such that each item represents a complete and correct configuration of the feature and its descendants. The first step in

the algorithm (5) simply takes a feature with no children and returns a list containing that feature, *i.e.*, if the feature's subtree contains only a single feature, the only valid configuration of that subtree is the single feature. The second step (6) merges the valid partial configurations of two nested XOR groups into a single partial configuration by merging their respective partial configuration sets into a single set. A visualization of this step is shown in Figure IX.8.

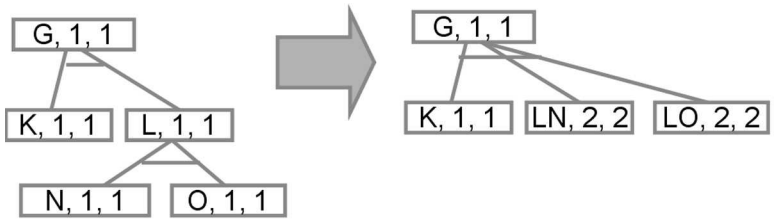


Figure IX.8: Flattening an XOR Group

The third step (7) takes all required children of a feature and produces a partial configuration containing a filtered Cartesian product of the feature's children, *i.e.*, the step selects a finite number of the valid configurations from the set of all possible permutations of the child features' configurations. A visualization of this step is shown in Figure IX.9. In code

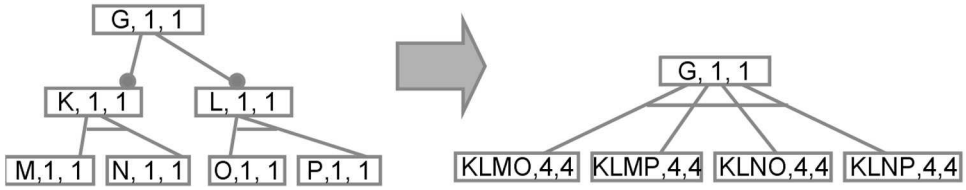


Figure IX.9: A Cartesian Product of Required Children

listing (8) in the APPENDIX, the Cartesian product is filtered identically to the way filters were used previously. The filter chooses *K* elements from the Cartesian product of the two sets using a selection strategy. The experiments in Our results show that a value of K=400 produced a good blend of speed and optimality.

Once each independent sub-tree has been converted into a set of partial configurations,

we must mark those sets that represent optional configuration choices. For each set that does not include the root feature, we add an item $\emptyset$ with zero weight and zero value indicating that no features in the set are chosen. Either a partial configuration from the set is selected or $\emptyset$ (representing no selection) is chosen. This method is a standard MMKP technique for handling situations where choosing an item from some sets is optional. Since the root feature must always be chosen, a partial configuration from its sub-tree's set must also be chosen, so the $\emptyset$ item is not added to its set.

Step 4: Handling Cross-tree Constraints

If any of the partial configurations in the MMKP sets contain cross-tree constraints, these constraints must eliminated before the MMKP solver is used. There are two cases for the cross-tree constraints that must be handled:

1. A partial configuration has a cross-tree constraint that refers to a feature in a sub-tree other than the sub-tree that produced its containing MMKP set.

2. A partial configuration has a cross-tree constraint that refers to a feature within the same sub-tree that produced its containing MMKP set.

The first case is handled by applying a series of filtered Cartesian products to each series of two sets that is connected through one or more cross-tree constraints. During the process of calculating the Cartesian product, when two partial configurations are chosen from each of the two sets, the combination of the configurations is validated to ensure that it does not violate any cross-tree exclusionary constraints. If the combination violates a cross-tree excludes constraint, the combined configuration is not added to the filtered Cartesian product of the two sets. In the case that a violation occurs, a constant number of retries, w, can be performed to find an alternate pair of compatible configurations. If no compatible pair is found within w tries, K is decremented for that set, and the Cartesian product continues.

The second case is handled by checking the validity of each partial configuration that contains one or more cross-tree constraints. Each of these partial configurations is checked to ensure that it adheres to its cross-tree constraints. If the configuration is valid, no changes are made. Invalid configurations are removed from their containing MMKP set. Cross-tree constraints within the same sub-tree are always handled after cross-tree constraints between sub-trees have been eliminated.

Step 5: MMKP Approximation

The first four steps produce an MMKP problem where each set contains items representing potential partial configurations of different parts of the feature model. One set contains partial configurations for the mandatory portions of the feature model connected to the root. The remaining sets contain partial configurations of the optional sub-trees of the feature model.

The final steps in deriving an optimal architectural feature selection involve running an existing MMKP approximation algorithm to select a group of partial configurations to form the architectural feature selection and then to combine these partial configurations into a complete architectural variant. For our implementation of Filtered Cartesian Flattening, we used a simple modification of the Modified Heuristic (M-HEU) algorithm [11] that puts an upper limit on the number of upgrades and downgrades that can be performed. Since Filtered Cartesian Flattening produces an MMKP problem, we can use any other MMKP approximation algorithm, such as the Convex Hull Heuristic algorithm (C-HEU) [104], which uses convex hulls to search the solution space. Depending on the algorithm chosen, the solution optimality and solving time will vary.

The items in the MMKP sets are built by concatenating the partial configurations of feature sub-trees during Cartesian products. With this arrangement, architectural feature configuration solutions can readily be extracted from the MMKP solution since they consist

of a partial configurations represented as a series of strings containing the labels of features that should be selected.

Algorithmic Complexity

The algorithmic complexity of Filtered Cartesian Flattening's constituent steps can be decomposed as follows (where n is the number of features):

- The first step in the Filtered Cartesian Flattening algorithm—cutting the tree—requires O(n) time to traverse the tree and find the top-level optional features where cuts can be made.

- The second step of the algorithm requires O($Kn * S$) steps, where S is the time required to perform the filtering operation. Simple filtering operations, such as random selection, add no additional algorithmic complexity. In these cases, at most n sets of K items must be created to convert the tree to XOR groups, yielding O(Kn). Our experiments selected the K items with the best value to resource consumption ratio. With this strategy, the sets must be sorted, yielding O($Kn * n \log n$).

- The third step in the algorithm requires flattening at most n groups using filtered Cartesian products, which yields a total time of O($Kn * S$).

- The fourth step in the algorithm requires producing filtered Cartesian products from at most n sets with w retries. Each configuration can be checked in O($c \log n$), where c is the maximum number of cross-tree constraints in the feature model. The total time to eliminate any cross-tree constraints between sets is O($wKn * S * c \log n$). The final elimination of invalid configurations within individual sets requires O($cn \log n$), yielding a total time of O($wKn * S * c \log n + cn \log n$)

- The solving step incurs the algorithmic complexity of the MMKP approximation algorithm chosen. With M-HEU, the algorithmic complexity is O($mn^2(l-1)^2$), where

m is the number of resource types, n is the number of sets, and l is maximum items per set.

- The final step, extracting the feature selection, can be performed in O(n) time.

This analysis yields a total general algorithmic complexity of O($n + (Kn * S) + (Kn * S) + (wKn * S) + MMKP + n$) = O($wKn * S * c \log n + cn \log n + MMKP$). If there are no cross-tree constraints, the complexity is reduced to O($Kn * S + MMKP$). Both algorithmic complexities are polynomial, which means that Filtered Cartesian Flattening scales significantly better than exponential exact algorithms. The results show that this translates into a significant decrease in running time compared to an exact algorithm.

Technique Benefits

Beyond the benefit of providing polynomial-time approximation for optimal feature selection problems with resource constraints, Filtered Cartesian Flattening exhibits the following other desirable properties:

One-time Conversion to MMKP: The Filtered Cartesian Flattening flattening process to create an MMKP problem need only be performed once per feature model. As long as the structure and resource consumption characteristics of the features do not change, the same MMKP problem representation can be used even when the resource allocations (we merely update the knapsack size) or desired system property to maximize change.

Flexible Filtering and Solving Strategies: Due to the speed of the Filtered Cartesian Flattening process, a number of different filtering strategies can be used and each resultant MMKP problem stored and used for optimization. In fact, to produce the most optimal results, a number of MMKP problems can be produced from each feature model and then each MMKP problem solved with several different MMKP techniques, and the most optimal solution produced can be used. Since there are multiple problem representations and

multiple algorithms used to solve the problem, there is a much lower probability that all of the representation/algorithm combinations will produce a solution with low optimality.

Flattening Parallelization: Another desirable property of Filtered Cartesian Flattening is that it is amenable to parallelization during the phase that populates the MMKP sets with partial configurations. After each subtree is identified, the Filtered Cartesian Flattening flattening process for each subtree can be run in parallel on a number of independent processors or processor cores.

Exact MMKP Algorithms Compatiblity: Finally, although we have focused on approximation algorithms for the MMKP phase of Filtered Cartesian Flattening, exact methods, such as integer programming, can be used to solve the MMKP problem. In this hybrid scenario, Filtered Cartesian Flattening would produce an approximate representation of the architectural feature model solution space using an MMKP problem and the exact optimal MMKP answer would be obtained. Filtered Cartesian Flattening allows the use of a wide variety of both Cartesian flattening strategies and MMKP algorithms to tailor solving time and optimality.

Results

This section presents empirical results from experiments we performed to evaluate the types of architectural feature selection problem instances on which Filtered Cartesian Flattening performs well and those for which it does not. When using an approximation algorithm, such as Filtered Cartesian Flattening, that does not guarantee an optimal answer a key question is how close the algorithm can get to the optimal answer. Another important consideration is what problem instance characteristics lead to more/less optimal answers from the algorithm. For example, if the algorithm attempts to derive an architectural variant for the face recognition system, will a more optimal variant be found when there is a larger or smaller budget constraint?

We performed the following two sets of experiments to test the capabilities of Filtered Cartesian Flattening:

- **Effects of MMKP problem characteristics.** Since Filtered Cartesian Flattening uses an MMKP approximation algorithm as its final solving step, we first performed experiments to determine which MMKP problem characteristics had the most significant impact on the MMKP approximation algorithm's solution optimality.

- **Effects of feature selection problem characteristics.** Our next set of experiments were designed to test which problem characteristics most influenced the entire Filtered Cartesian Flattening technique's solution optimality. These experiments also included a large experiment that derived Filtered Cartesian Flattening's average and worst optimality on a set of 500,000 feature models.

All experiments used 8 dual processor 2.4ghz Intel Xenon nodes with 2 GB RAM on Vanderbilt University's ISISLab cluster (`www.isislab.vanderbilt.edu`). Each node was loaded with Fedora Core 4. A total of two processes (one per processor) were launched on each machine enabling us to generate and solve 16 optimal feature selection with resource constraints problems in parallel.

Testing MMKP Problem Characteristics

To determine the extent to which the various attributes of MMKP problems would affect the ability of the solver to generate a highly optimal solution, we generated several MMKP problems with a single parameter adjusted. These problems were then solved using the MMKP approximation algorithm. Solutions were rated by their percentage of optimality vs. the optimal solution ($\frac{MMKPApproximationAnswer}{OptimalAnswer}$) (we used the problem generation technique devised by [11] to generate random MMKP problem instances for which we knew the optimum answer). Our test problems included a mix of problems with a correlation between value and total resource consumption and those without any correlation.

MMKP problem instances can vary across a number of major axes. Problem instances can have larger and smaller numbers of sets and items per set. The range of values and resource consumption characteristics across the items can follow different distributions. We examined each of these MMKP problem attributes to determine which ones lead to the generation of solutions with a higher degree of optimality. Each experiment was executed thirty times and averaged to normalize the data.

First, we manipulated the total number of sets in an MMKP problem. The Filtered Cartesian Flattening algorithm produces one set for each independent subtree in the feature model. This experiment allowed us to test how feature models with a large number of independent subtrees and hence a large number of MMKP sets would affect solution optimality. Figure IX.10 shows that as the total number of sets was increased from 10 to 100, the solution optimality only varied a small amount, staying well above 95% optimal. These results

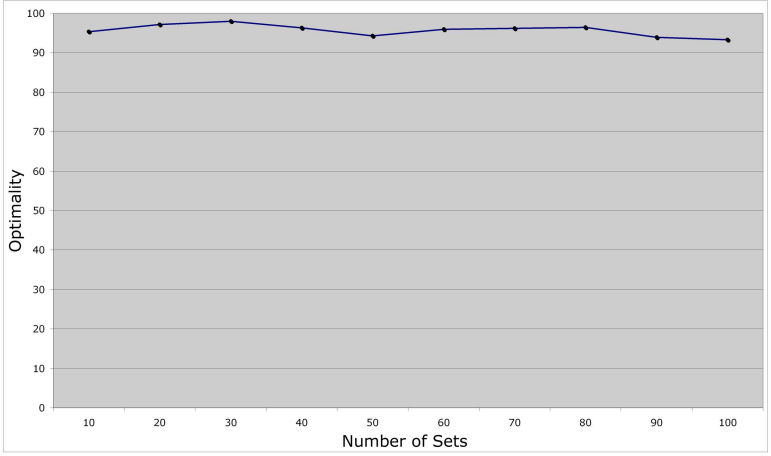


Figure IX.10: Total Number of Sets

are nearly identical to [11], where the M-HEU MMKP approximation algorithm, which was the basis of our MMKP solver, produced solutions well above 98% optimal regardless of the number of sets or items per set.

We next varied the number of items in each MMKP set. Figure IX.11 shows that an

increase from 500 to 10,000 items per set has almost no affect the optimality of the solution. Regardless of the number of items per set, the generated solution was well over 90%

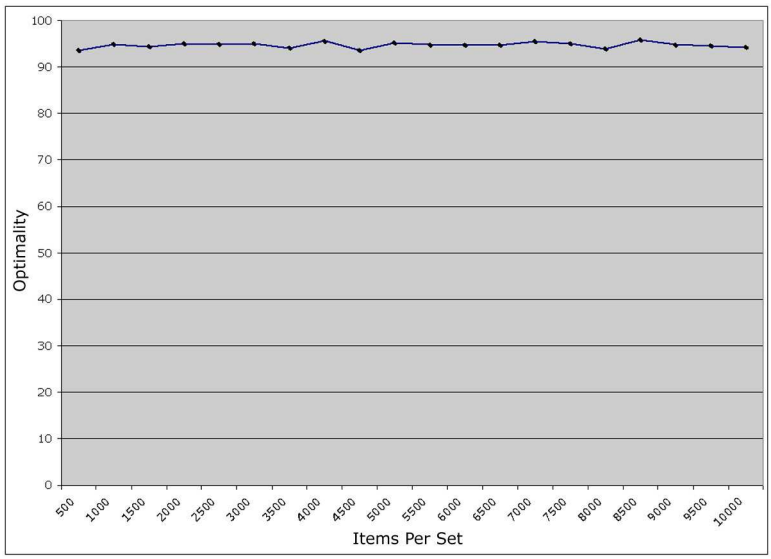


Figure IX.11: Items per Set

optimal. Based on this data, we conclude that the number of sets and total items per set do not significantly impact the optimality of the solution produced by the MMKP solver. This result implies that architectural feature models for very large industrial systems will not be problematic for the MMKP phase of Filtered Cartesian Flattening.

While the items per set and number of sets have little affect on the optimality of a solution, the number of resources, and the amount of resources consumed by items were found to negatively impact the ability of the solver to find a solution with high optimality. Figure IX.12 shows the affect of raising the minimum amount of resources consumed by an item. The optimality drops drastically as the minimum amount of resources consumed by an item becomes a larger percentage of the total available resources. For a solution to maintain a forecasted optimality of over 80% percent, the minimum amount of resources consumed by an item must be less than 10% percent of the total amount of available resources. Increasing the minimum amount of resources consumed by an item causes more items to consume a relatively large share of the total available resources.

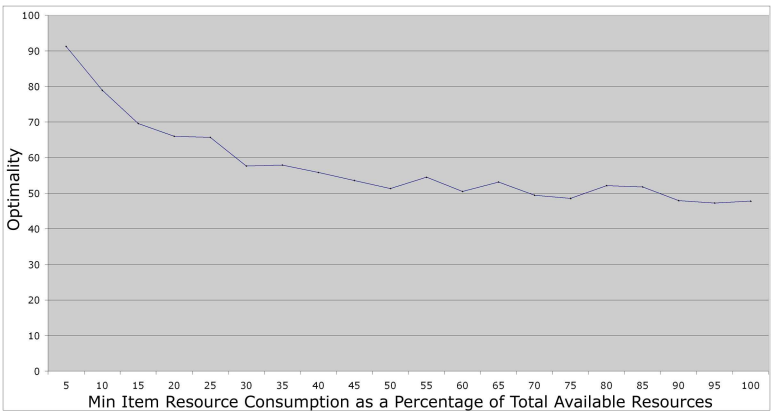


Figure IX.12: Minimum Resource Consumption per Item

The results from the experiment that gradually increased the minimum item resource consumption led us to hypothesize that the MMKP solver will produce less optimal solutions when the average item consumes a very large percentage of the available resources. We performed another experiment where we (1) calculated a resource tightness metric that measured the average resource consumption of the items and (2) estimated how many items with the average resource consumption could fit into the available resource allocation, *i.e.*, how many of the average sized items could be expected to fit into the knapsack. Our tightness metric was calculated as:

$$\frac{\sqrt{R_0^2 + \ldots R_m^2}}{\sqrt{(\sum_{i=0}^{n} r(i,0)^2 + \ldots r(i,m)^2)/n}}$$

where m is the total number of resource types, R_i is the maximum available amount of the i_{th} resource, and $r(i,j)$ is the amount of the j_{th} resource consumed by the i_{th} item.

The results from the resource tightness experiment are shown in Figure IX.13. The x-axis shows the estimated number of average sized items that are expected to fit into the knapsack for a feature model with 50 sets. As shown in the figure, there is a dramatic dropoff in optimality when less than 1.65 average sized items can fit in the knapsack. The exact value for the tightness metric at which the dropoff occurs varies based on the number of MMKP sets. With 100 sets, the value was $\sim$1.83.

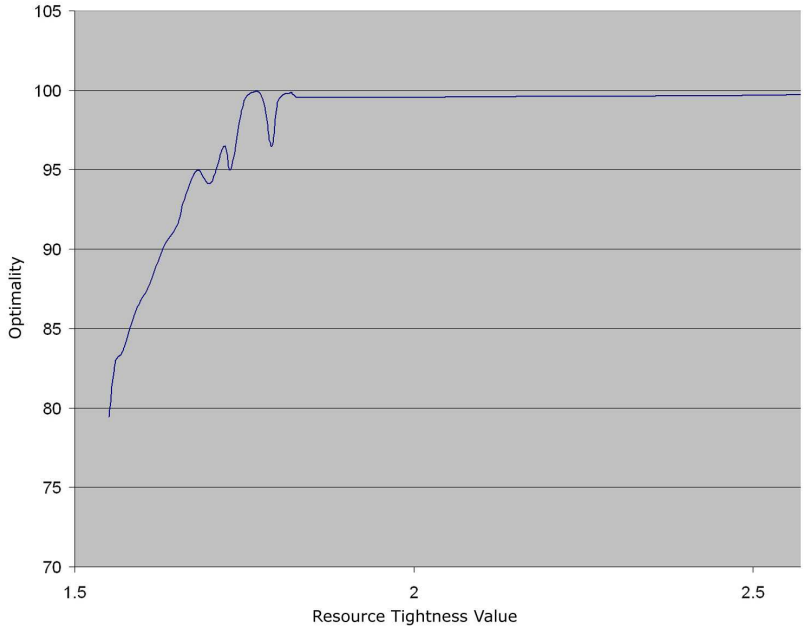


Figure IX.13: Effect of Resource Constraint Tightness on MMKP Optimality

The fewer average items that can fit into the knapsack, the more likely the solver is to make a mistake that will fill up the knapsack and widely miss the optimal value. This result implies that the Filtered Cartesian Flattening algorithmic approach works well when making are a relatively large number of finer-grained feature selection decisions. For architectures with a few very coarse-grained decisions, a developer or exact technique [21] is more likely to pick a more appropriate architectural variant.

Resource tightness also played a role in how the total number of resource types affected solution optimality. Figure IX.14 shows how the optimality of solving problems with 50 sets was affected as the total number of resource types climbed from 2 to 95. For this experiment, the tightness metric was kept above the 1.65 dropoff threshold. As can be seen, the total number of resources had a relatively slight impact of approximately 5% on solution optimality. The results in Figure IX.15, however, are quite different. In the experiment that produced Figure IX.15, the tightness metric was kept at a relatively constant 1.55, *i.e.*, below the dropoff value. As shown by the results, the total number of resource types had a significant impact on solution optimality.

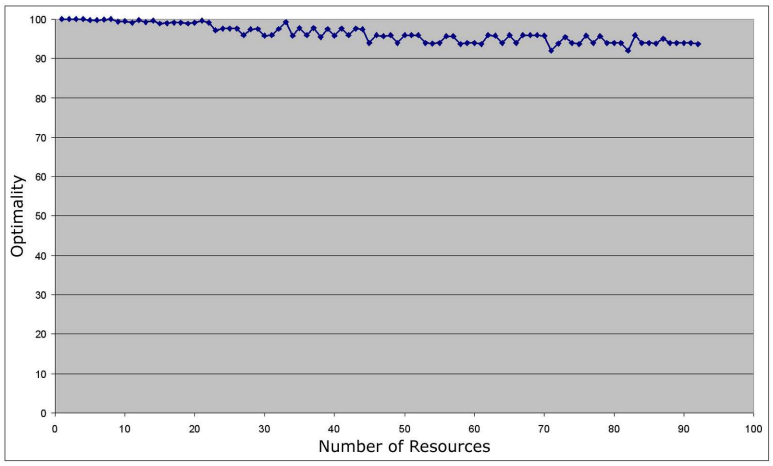


Figure IX.14: Total Number of Resources

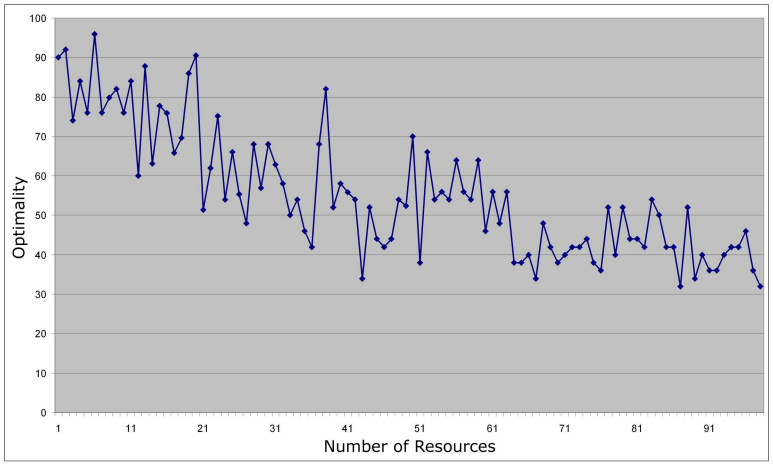


Figure IX.15: Total Number of Resources

Comparing Filtered Cartesian Flattening to CSP-based Feature Selection

Our initial tests with Filtered Cartesian Flattening compared its performance and optimality on small-scale feature selection problems to the Constraint Satisfaction Problem (CSP) based feature selection technique described in [22]. This technique uses a general-purpose constraint solver to derive a feature selection. For these small scale-problems, we tracked the time required for Filtered Cartesian Flattening to find a solution vs. the CSP-based technique based on open-source Java Choco constraint solver (`choco-solver.net`). For each solution, we compared Filtered Cartesian Flattening's answer to the guaranteed optimal answer generated by the CSP-based technique.

Figure IX.16 shows the time required for Filtered Cartesian Flattening and the CSP-based technique to find architectural variants in feature models with varying numbers of XOR groups. The x-axis shows the number of XOR groups in the models and the y-axis

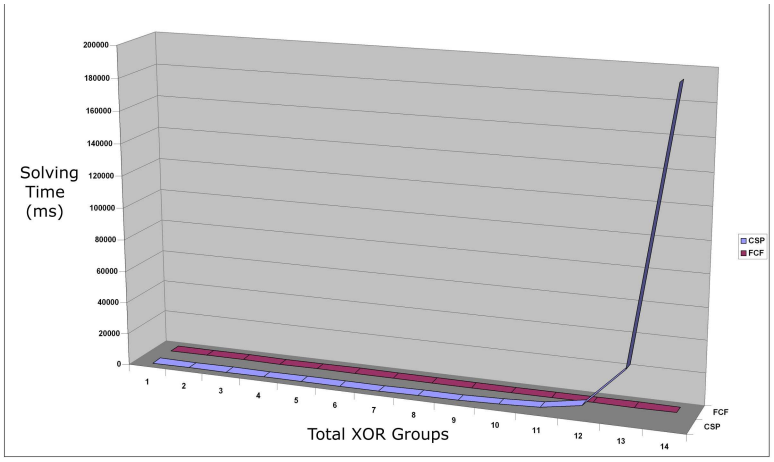


Figure IX.16: Comparison of Filtered Cartesian Flattening and CSP-based Feature Selection Solving Times

displays the time required to find an architectural variant. The total features in each model was ∼3-10 times the number of XOR groups (the maximum size was < 140 features). Each feature-model had a maximum of 10% of the features involved in a cross-tree constraint, $c \leq 0.1n$. As shown in the figure, the CSP-based technique initially requires approximately

30ms to find a solution. The CSP technique's time, however, quickly grows at an exponential rate to over 198,000ms. In contrast, Filtered Cartesian Flattening required less than 1ms for every feature model.

Even though Filtered Cartesian Flattening ran substantially faster than the CSP-based technique, it still provided a high level of optimality. Overall, the solutions generated by Filtered Cartesian Flattening were 92% optimal compared to 100% optimal for the CSP-based technique. The Filtered Cartesian Flattening solution with the lowest optimality was 80% optimal. Although Filtered Cartesian Flattening does not provide 100% optimal results, it can be used to derive good architectural variants for architectures that are too large to solve with an exact technique.

Filtered Cartesian Flattening Test Problem Generation

Due to the exponential time curve required to solve a feature selection problem using an exact technique, it was not possible to solve large-scale problems using both Filtered Cartesian Flattening and an exact technique. This section presents the problem generation technique we used to create large-scale feature selection problems for which we knew the optimal answer. This problem generation approach allowed us to generate extremely large problems with a known optimal solution that were not feasible to solve with an exact technique.

Filtered Cartesian Flattening problem instances vary based on the structural properties of the feature model tree, such as the percentage of XOR groups, max depth, and maximum number of children per feature. The MMKP properties tested, such as the resource tightness of the problem, can also vary based on how features consume resources. We tested the effect of these problem characteristics by both generating problem instances that exhibited a specific characteristic and by performing post-mortem analysis on the results of solving over 500,000 random Filtered Cartesian Flattening problem instances. The post-mortem

analysis determined the problem characteristics associated with the problem instances that were solved with the worst optimality.

To create test data for the Filtered Cartesian Flattening technique, we generated random feature models and then created random feature selection problems with resource constraints from the feature models. For example, we first generated a feature model and then assign each feature an amount of RAM, CPU, etc. that it consumed. Each feature was also associated with a value. We then randomly generated a series of available resource values and ask Filtered Cartesian Flattening to derive the feature selection that maximized the sum of the value attributes while not exceeding the randomly generated available resources. Finally, we compared the Filtered Cartesian Flattening answer to the optimum answer. No models included any cross-tree constraints because there are no known methods for generating large feature selection problems that include cross-tree constraints and have a known optimal solution.

In an effort to make the feature models as representative of real architectural feature models as possible, we created models with a number of specific characteristics. For example, developers with significant object-oriented development experience often create models where commonality is factored into parent features, identical to how an inheritance hierarchy is built. Figure IX.2 shows a hierarchy used to categorize the various facial recognition algorithms. SPL architectural analysis techniques, such as *Scope, Commonality, Variability Analysis* [41] are used to derive these hierarchies.

Developers desires to provide a well structured hierarchy has two important ramifications for the feature model. First, feature models typically have a relatively limited number of child features for each feature. Hierarchies are used to model a large number of child features as subtrees rather than simply a long list of alternatives. Second, the actual features that consume resources and provide value are most often the leaves of the feature model. In the categorization of facial recognition algorithms shown in Figure IX.2, the actual resource consumption and accuracy of the algorithm is not specifically known until

reaching one of the leaves, such as Euclidean or MahCosine. To mirror these properties of developer-created feature models, we limited the number of child features of a feature to 10 and heavily favored the association of resource consumption and value with the leaves of the feature model.

We used a feature model generation infrastructure that we developed previously [158]. A key challenge was determining a way to randomly assign resource consumption values and values to features such that we knew the exact optimum value for the ideal feature selection. Moreover, we needed to ensure that the randomly generated problems would not exhibit characteristics that would make them easily solved by specific MMKP algorithms. For example, if every feature in the optimum feature selection also had the highest value in its selection set, the problem could be solved easily with a greedy algorithm.

To assign resource consumption values to features and generate random available resource allocations, we used a modified version of the algorithm in [11] to ensure that the highest valued features were no more likely part of the optimal solution than any other feature. The steps to generate a feature selection problem with k different resource types and n features were as follows:

1. Generate a k-dimensional vector, r_a, containing random available allocations for the k resource types,
2. Randomly generate a slack value v_s,
3. Randomly generate an optimum value v_{opt},
4. For each top-level XOR group, q, in each independent sub-tree, randomly choose a feature, f_{qj}, to represent the optimal configuration and assign it value $opt_{qj} = v_{opt}$,
5. For each optimal feature, assign it a k dimensional resource consumption vector, r_{qj}, such that the sum of the components of the optimal resource consumption vectors exactly equal the available resource allocation vector, $\sum r_{qj} = r_a$,

6. For each top-level XOR group member f_i that is not the optimal feature f_{qj} in its group either:

   - assign the feature value v_i, where $v_i < (opt_{qj} - v_s)$ and randomly assign it a resource consumption vector
   - assign the feature value v_i, where $opt_{qj} < v_i < opt_{qj} + v_s$, and randomly assign f_i a resource consumption vector such that each component is greater than the corresponding component in r_{qj}. After each XOR group's features are completely initialized, set $v_s = max(v_i) - opt_{qj}$, where $max(v_i)$ is the the highest value of any item in the XOR group.

7. For each feature in a top-level XOR group, reset the available resources vector to the feature's resource consumption vector, reset the optimum value to the feature's value, and recursively apply the algorithm, treating the feature as the root of a new sub-tree

Filtered Cartesian Flattening Optimality

After determining the key MMKP problem characteristics that influence the optimality of the MMKP phase of Filtered Cartesian Flattening, we ran a series of experiments to evaluate the parameters that affect the feature model flattening phase. Figure IX.17 presents results illustrating how the percentage of features involved in XOR groups within the feature model affects solution optimality. As shown in this figure, as the percentage of features in XOR groups increases from 10% to 90% of features, there is a negligible impact on optimality of the solutions produced by Filtered Cartesian Flattening.

We tested a wide range of other Filtered Cartesian Flattening properties, such as the maximum depth and the maximum branching factor of the feature model tree, and saw

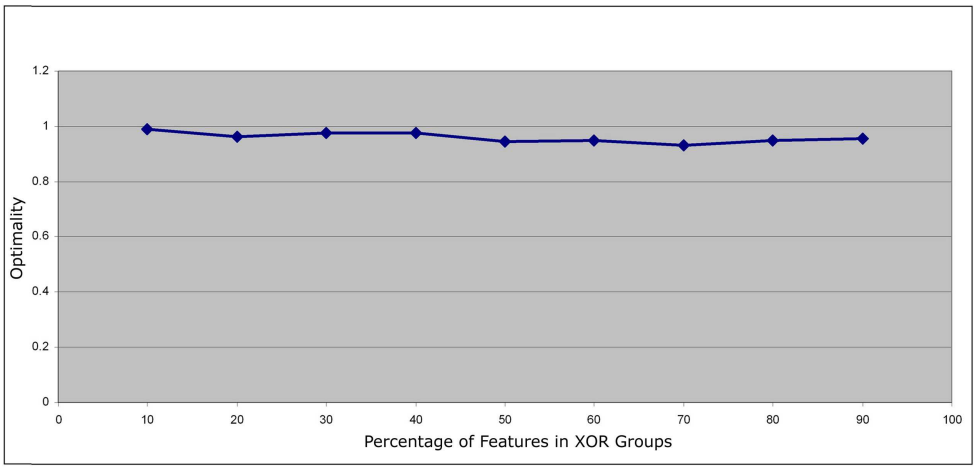


Figure IX.17: Effect of Feature Model XOR Percentage on Filtered Cartesian Flattening Optimality

no impact on solution optimality. Other experiments included tests that assigned and distributed value and resource consumption to sub-trees in correlation to the size of the sub-tree. We also experimented with feature models that evenly distributed value and resource consumption across all features as opposed to clustering resource consumption and value towards the leaves. The effect of different value ranges was also tested.

In each case, we observed no affect on solution optimality. The result graphs from these experiments have been omitted for brevity. Our resource tightness metric had the most significant impact on Filtered Cartesian Flattening solution optimality, just as it did with MMKP approximation optimality.

Our largest experiment checked the range of solution optimalities produced by using Filtered Cartesian Flattening to solve 450,000 optimal feature selection problems with resource constraints. The total number of features was set to 1,000, the XOR Group percentage to 50%, $K = 2500$, and the resource tightness metric was greater than 2.0 for the majority of the problem instances (well above the dropoff point). As shown in Figure IX.18, the results are presented with a histogram showing the number of problem instances that were solved with a given optimality. The overall average optimality across all instances was 95.54%. The lowest solution optimality observed was 72%.

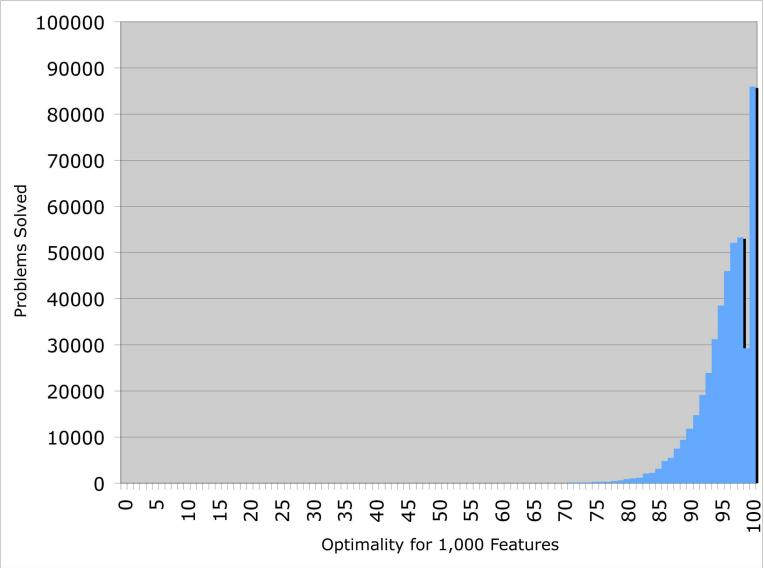


Figure IX.18: A Histogram Showing the Number of Problems Solved with a Given Optimality from 450,000 Feature Models with 1,000 Features

Figure IX.19 presents data from solving approximately 8,000 feature selection problems with 10,000 features. Again, we used a filtering scheme with $K = 2500$ that chose the

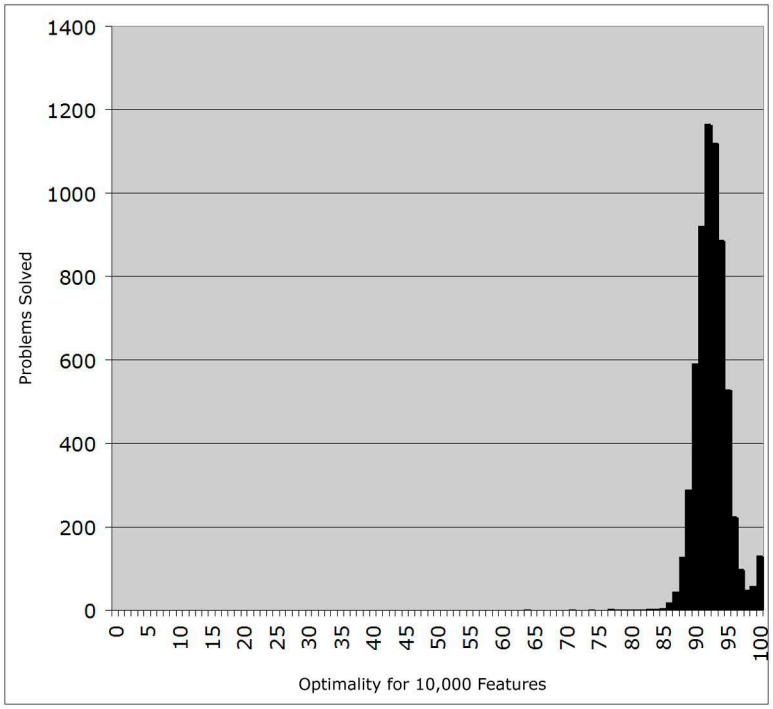


Figure IX.19: A Histogram Showing the Number of Problems Solved with a Given Optimality from 8,000 Feature Models with 10,000 features

K items with the best ratio of value to weight. The average optimality across all problem instances was approximately 92.56%.

Across all feature model sizes (both 1,000 and 10,000 features), 90% of the problem instances were solved with an optimality greater than ∼91%. Moreover, 99% were solved with an optimality greater than ∼80%. These result cutoffs only hold when the tightness metric is above the drop-off value.

An interesting result can be seen by comparing Figures IX.19 and IX.18. As the number of features increases, the range of solution optimalities becomes much more tightly clustered around the average solution optimality. Akbar's results [11] showed an increase in M-HEU solution optimality as the number of sets and items per set increased. Our results showed a slight decrease of 3% in average solution optimality for Filtered Cartesian Flattening as the total features increased from 1,000 to 10,000. We expect that the slight decrease is a result of more potentially good partial configurations being filtered out during the Filtered Cartesian Flattening Cartesian flattening phase.

Summary and Analysis of Experiment Results

From the data we obtained from our Filtered Cartesian Flattening experiments, we confirmed that the key predictor of MMKP solution optimality—resource tightness—was also applicable to Filtered Cartesian Flattening problems. For all experiments we ran, those problems that were solved with less than 70% optimality had an average resource tightness metric of 0.94, which is well below the dropoff point of roughly 1.65 that we observed for 50 sets. Moreover, the max tightness value for these problems was 1.67, which is right at the edge of the dropoff.

Although a low value for the resource tightness metric indicates that a low optimality is possible, it does not guarantee it. Some problems with tightness metrics below the drop-off were solved with 100 or 90%+ optimality. Once the MMKP problem representation is produced, calculating the tightness metric is an O(n) operation. Due to the ease of calculating

the resource tightness metric, developers should always use it to rule out problem instances were Filtered Cartesian Flattening is unlikely to produce an 80-90%+ optimal solution.

CHAPTER X

CONFIGURING HARDWARE AND SOFTWARE IN TANDEM

Introduction

Current trends and challenges. Increasing levels of programming abstraction, middleware, and other software advancements have expanded the scale and complexity of software systems that we can develop. At the same time, the ballooning scale and complexity have created a problem where systems are becoming so large that their design and development can no longer be optimized manually. Current large-scale systems can contain an exponential number of potential design configurations and vast numbers of constraints ranging from security to performance requirements. Systems of this scale and complexity—coupled with the increasing importance of non-functional characteristics [36] (such as end-to-end response time)—are making software design processes increasingly expensive [110].

Search-based software engineering [68, 69] is an emerging discipline that aims to decrease the cost of optimizing system design by using algorithmic search techniques, such as genetic algorithms or simulated annealing, to automate the design search. In this paradigm, rather than performing the search manually, designers iteratively produce a design by using a search technique to find designs that optimize a specific system quality while adhering to design constraints. Each time a new design is produced, designers can use the knowledge they have gleaned from the new design solution to craft more precise constraints to guide the next design search. Search-based software engineering has been applied to the design of a number of software engineering aspects, ranging from generating test data [97] to project management and staffing [13, 16] to software security [37].

A common theme in domains where search-based software engineering is applied is that the design solution space is so large and tightly constrained that the time required to

find an optimal solution grows at an exponential rate with the problem size. These vast and constrained solutions spaces make it hard for designers to derive good solutions manually. This chapter examines a common problem from the domain of distributed real-time and embedded (DRE) systems that exhibits these complexity characteristics. The problem we focus on is the need to derive a design that maximizes a specific system capability subject to constraints on cost and the production and consumption of resources, such as RAM, by the hardware and software, respectively.

For example, when designing a satellite to earth's magnetosphere [45], the goal may be to maximize the accuracy of the sensor data processing algorithms on the satellite without exceeding the development budget and hardware resources. Ideally, to maximize the capabilities of the system for a given cost, system software and hardware should be designed in tandem to produce a design with a precise fit between hardware capabilities and software resource demands. The more precise the fit, the less budget is expended on excess hardware resource capacity.

A key problem in these design scenarios is that they create a complex cost-constrained producer/consumer problem involving the software and hardware design. The hardware design determines the resources, such as processing power and memory, that are available to the software. Likewise, the hardware consumes a portion of the project budget and thus reduces resources remaining for the software (assuming a fixed budget). The software also consumes a portion of the budget and the resources produced by the hardware configuration. The perceived value of system comes from the attributes of the software design, *e.g.*, image processing accuracy in the satellite example. The intricate dependencies between the hardware and software's production and consumption of resources, cost, and value makes the design solution space so large and complex that finding an optimal and valid design configuration is hard.

Solution approach → Automated Solution Space Exploration. This chapter presents

a heuristic search-based software engineering technique, called the *Allocation-baSed Configuration Exploration Technique* (ASCENT), for solving cost-constrained hardware/software producer/consumer co-design problems. ASCENT models these co-design problems as two separate knapsack problems [75]. Since knapsack problems are NP-Hard [42], ASCENT uses heuristics to reduce the solution space size and iteratively search for near optimal designs by adjusting the budget allocations to software and hardware. In addition to outputting the best design found, ASCENT also generates a data set representing the trends it discovered in the solution space.

A key attribute of the ASCENT technique is that, in the process of solving, it generates a large number of optimal design configurations that present a wide view of the trends and patterns in a system's design solution space. This chapter shows how this wide view of trends in the solution space can be used to iteratively search for near optimal co-design solutions. Moreover, our empirical results show that ASCENT produces co-design configurations that average 95%+ optimal for problems with more than 7 points of variability in each of the hardware and software design spaces.

Motivating Example

This section presents a satellite design example to motivate the need to expand search-based software engineering techniques to encompass cost-constrained hardware/software producer/consumer co-design problems. Designing satellites, such as the satellite for NASA's Magnetospheric Multiscale (MMS) mission [45], requires carefully balancing hardware/-software design subject to tight budgets. Figure X.1 shows a satellite with a number of possible variations in software and hardware design. For example, the software design has a point of variability where a designer can select the resolution of the images that are processed. Processing higher resolution images improves the accuracy but requires more RAM and CPU cycles.

Another point of variability in the software design is the image processing algorithms

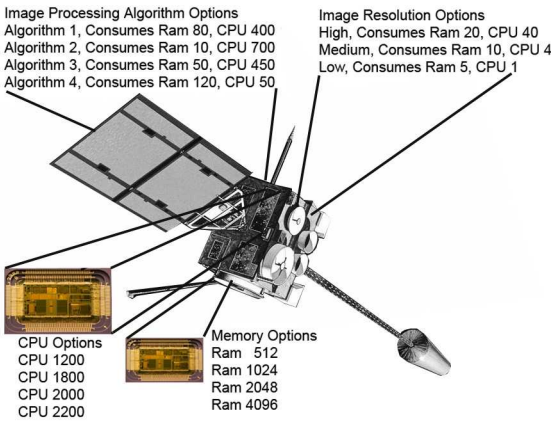


Figure X.1: Software/Hardware Design Variability in a Satellite

that can be used to identify characteristics of the images captured by the satellite's cameras. The algorithms each provide a distinct level of accuracy, while also consuming different quantities of RAM and CPU cycles. The underlying hardware has a number of points of variability that can be used to increase or decrease the RAM and CPU power to support the resource demands of different image processing configurations. Each configuration option, such as the chosen algorithm or RAM value, has a cost associated with it that subtracts from the overall budget. A key question design question for the satellite is: *what set of hardware and software choices will fit a given budget and maximize the image processing accuracy*.

Many similar design problems involving the allocation of resources subject to a series of design constraints have been modeled as *Multidimensional Multiple-Choice Knapsack Problems* (MMKPs) [12, 74, 76]. A standard knapsack problem [75] is defined by a set of items with varying sizes and values. The goal is to find the set of items that fits into a fixed sized knapsack and that simultaneously maximizes the value of the items in the knapsack. An MMKP problem is a variation on a standard knapsack problem where the items are divided into sets and at most one item from each set may be placed into the knapsack.

Figure X.2 shows an example MMKP problem where two sets contain items of different sizes and values. At most one of the items A,B, and C can be put into the knapsack.

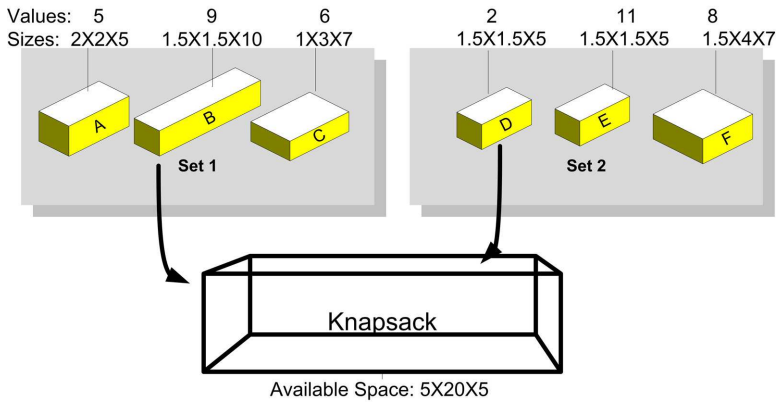


Figure X.2: An Example MMKP Problem

Likewies, only one of the items D, E, and F can be put into the knapsack. The goal is to find the combination of two items, where one item is chosen from each set, that fits into the knapsack and maximizes the overall value. A number of resource related problems have been modeled as MMKP problems where the sets are the points of variability in the design, the items are the options for each point of variability, and the knapsack/item sizes are the resources consumed by different design options [11, 34, 76, 86, 142].

The software and hardware design problems are hard to solve individually. Each design problem consists of a number of design variability points that can be implemented by exactly one design option, such as a specific image processing algorithm. Each design option has an associated resource consumption, such as cost, and value associated with it. Moreover, the design options cannot be arbitrarily chosen because there is a limited amount of each resource available to consume.

It is apparent that the description of the software design problem directly parallels the definition of an MMKP problem. An MMKP set can be created for each point of variability (*e*.g., Image Resolution and Algorithm). Each set can then be populated with the options for its corresponding point of variability (*e*.g., High, Medium, Low for Image Resolution). The items each have a size (cost) associated with them and there is a limited size knapsack (budget) that the items can fit into. Clearly, just selecting the optimal set of software features subject to a maximum budget is an instance of the NP-Hard [42] MMKP problem.

For the overall satellite design problem, we must contend with not one but two individual knapsack problems. One problem models the software design and the second problem models the hardware design. We can model the satellite co-design problem using two MMKP problems. The first of the two MMKP problems for the satellite design is its software MMKP problem. The hardware design options are modeled in a separate MMKP problem with each set containing the potential hardware options. An example mapping of the software and hardware design problems to MMKP problems is shown in Figure X.3.

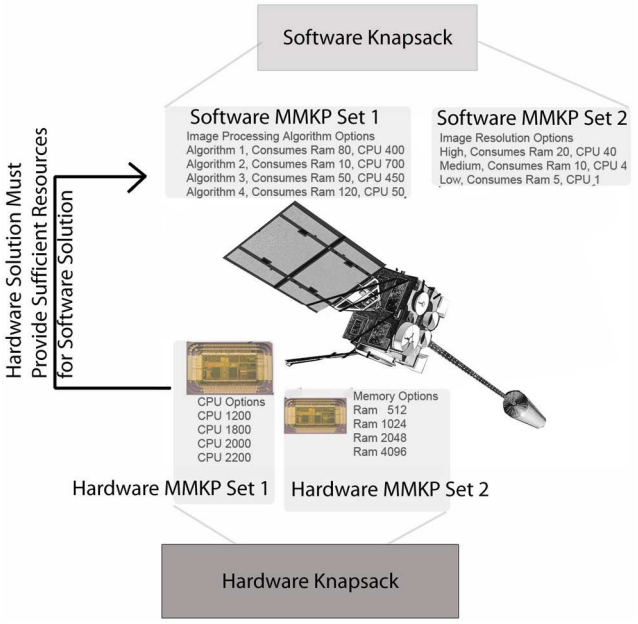


Figure X.3: Modeling the Satellite Design as Two MMKP Problems

We call this combined two problem MMKP model a *MMKP co-design problem*. With this MMKP co-design model of the satellite, a design is reached by choosing one item from each set (*e*.g., an Image Resolution, Algorithm, RAM value, and CPU) for each problem. The correctness of the design can be validated by ensuring that exactly one item is chosen from each set and that the items fit into their respective software and hardware knapsacks. This definition, however, is still not sufficient to model the cost-constrained hardware/software producer/consumer co-design problem since we have not accounted for

the constraint on the total size of the two knapsacks or the production and consumption of resources by hardware and software.

A correct solution must also uphold the constraint that the items chosen for the solution to the software MMKP problem do not consume more resources, such as RAM, than are produced by the items selected for the solution to the hardware MMKP problem. Moreover, the cost of the entire selection of items must be less than the total development budget. We know that solving the individual MMKP problems for the optimal hardware and software design is NP-Hard but we must also determine how hard solving the combined co-design problem is.

In this simple satellite example, there are 192 possible satellite configurations to consider. For real industrial scale examples, there are a significantly larger number of possibilities. For example, a system with design choices that can be modeled using 64 MMKP sets, each with 2 items, will have 2^{64} possible configurations. For systems of this scale, manual solving methods are clearly not feasible, which motives the need for a search-based software engineering technique.

MMKP Co-design Complexity

Below, we show that MMKP co-design problems are NP-Hard and in need of a search-based software engineering technique. We are not aware of any approximation techniques for solving MMKP co-design problems in polynomial time. This lack of approximation algorithms—coupled with the poor scalability of exact solving techniques—hinders DRE system designers's abilities to optimize software and hardware in tandem.

To show that MMKP co-design problems are NP-Hard, we must build a formal definition of them. We can define an MMKP co-design problem, *CoP*, as an 8-tuple:

$$CoP = < Pr, Co, S_1, S_2, S, R, Uc(x,k), Up(x,k) >$$

where:

- Pr is the producer MMKP problem (*e*.g., the hardware choices).
- Co is the consumer MMKP problem (*e*.g., the software choices).
- S_1 is the size of the producer, Pr, knapsack.
- S_2 is the size of the consumer, Co, knapsack.
- R is the set of resource types (*e*.g., RAM, CPU, etc.) that can be produced and consumed by Pr and Co, respectively.
- S is the total allowed combined size of the two knapsacks for Pr and Co (*e*.g., total budget).
- $Uc(x,k)$ is a function which calculates the amount of the resource $k \subset R$ **c**onsumed by an item $x \subset Co$ (*e*.g., RAM consumed).
- $Up(x,j)$ is a function which calculates the amount of the the resource $k \subset R$ **p**roduced by an item $x \subset Pr$ (*e*.g., RAM provided).

Let a solution to the MMKP co-design problem be defined as a 2-tuple, $< p, c >$, where $p \subset Pr$ is the set of items chosen from the producer MMKP problem and $c \subset Co$ is the set of items chosen from the consumer MMKP problem. A visualization of a solution tuple is shown in Figure X.4. We define the value of the solution as the sum of the values of the elements in the consumer solution:

$$V = \sum_{0}^{j} valueof(c_j)$$

where j is the total number of items in c, c_j is the j_{th} item in c, and $valueof()$ is a function that returns the value of an item in the consumer soution.

We require that p and c are valid solutions to Pr and Co, respectively. For p and c to be valid, exactly one item from each set in Pr and Co must have been chosen. Moreover, the

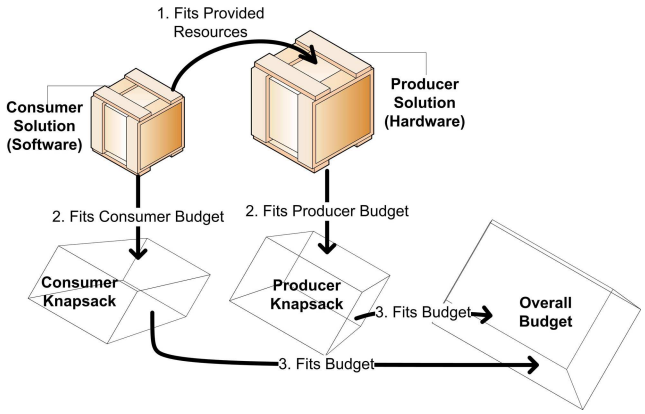


Figure X.4: Structure of an MMKP Co-design Problem

items must fit into the knapsacks for *Pr* and *Co*. [1]This constraint corresponds to Rule (2) in Figure X.4 that each solution must fit into the budget for its respective knapsack.

The MMKP co-design problem adds two additional constraints on the solutions p and c. First, we require that the items in c do not consume more of any resource than is produced by the items in p:

$$(\forall k \subset R), \sum_{0}^{j} Uc(c_j,k) \leq \sum_{0}^{l} Up(p_l,k)$$

where j is the total number of items in c, c_j is the j_{th} item in c, l is the total number of items in p, and p_j is the j_{th} item in p. Visually, this means that the consumer solution can fit into the producer solution's resources as shown in Rule (1) in Figure X.4.

The second constraint on c and p is an interesting twist on traditional MMKP problems. For a MMKP co-design problem, we do not know the exact sizes, S_1, S_2, of each knapsack. Part of the problem is determining the sizes as well as the items for each knapsack. Since we are bound by a total overall budget, we must ensure that the sizes of the knapsacks do not exceed this budget:

$$S_1 + S_2 \leq S$$

This constraint on the overall budget corresponds to Rule (3) in Figure X.4.

To show that solving for an exact answer to the MMKP problem is NP-Hard, we will

[1]A more formal definition of MMKP solution correctness is available from [12].

show that we can reduce any instance of the NP-complete *knapsack decision problem* to an instance of the MMKP co-design problem. The knapsack decision problem asks if there is a combination of items with value at least V that can fit into the knapsack without exceeding a cost constraint.

A knapsack problem can easily be converted to a MMKP problem as described by Akbar et al. [12]. For each item, a set is created containing the item and the $\emptyset$ item. The $\emptyset$ item has no value and does not take up any space. Using this approach, a knapsack decision problem, K_{dp}, can be converted to a MMKP decision problem, M_{dp}, where we ask if there is a selection of items from the sets that has value at least V.

To reduce the decision problem to an MMKP co-design problem, we can use the MMKP decision problem as the consumer knapsack ($Co = M_{dp}$), set the producer knapsack to an MMKP problem with a single item with zero weight and value ($\emptyset$), and let our set of produced and consumed resources, R, be empty, $R = \emptyset$. Next, we can let the total knapsack size budget be the size of the decision problem's knapsack, $S = sizeof(M_{dp})$.

The co-design solution, which is the maximization of the consumer knapsack solution value, will also be the optimal answer for the decision problem, M_{dp}. We have thus setup the co-design problem so that it is solving for a maximal answer to M_{dp} without any additional producer/consumer constraints or knapsack size considerations. Since any instance of the NP-complete knapsack decision problem can be reduced to an MMKP co-design problem, the MMKP co-design problem must be NP-Hard.

Challenges of MMKP Co-design Problems

This section describes two key challenges to building an approximation algorithm to solve MMKP co-design problems. The first challenge is that determining how to set the budget allocations of the software and hardware is not straightforward since it involves figuring out the precise size of the software and hardware knapsacks where the hardware knapsack produces sufficient resources to support the optimal software knapsack solution

(which itself is unknown). The second challenge is that the tight-coupling between producer and consumer MMKP problems makes them hard to solve individually, thus motivating the need for a heuristic to de-couple them.

Challenge 1: Undefined Producer/Consumer Knapsack Sizes

One challenge of the MMKP co-design problem is that the individual knapsack size budget for each of the MMKP problems is not predetermined, *i.e.*, we do not know how much of the budget should be allocated to software versus hardware, as shown in Figure X.5. The only constraint is that the sum of the budgets must be less than or equal to

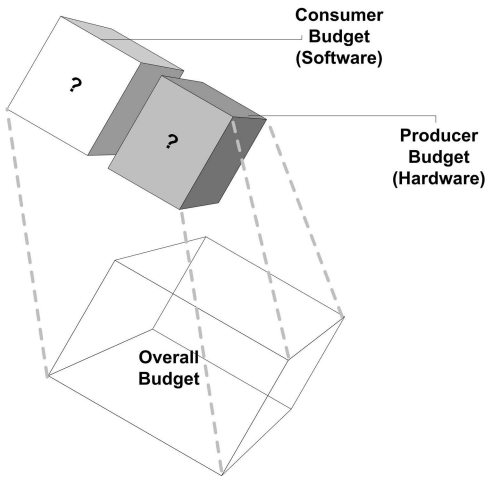


Figure X.5: Undefined Knapsack Sizes

the an overall total budget. Every pair of budget values for hardware and software results in two new unique MMKP problems. Even minor transfers of capital from one problem budget to the other can therefore completely alter the solution of the problem, resulting in a new maximum value. Existing MMKP techniques assume that the exact desired size of the knapsack is known.

There is currently no information to aid designers in determining the allocation of the budgets. As a result, many designers may choose the allocation arbitrarily without realizing the profound impact it may have. For example, a budget allocation of 75% software and

25% software may result in a solution that, while valid, provides far less value and costs considerably more than a solution with a budget allocation of 74% and 26% percent.

There are, however, trends in the solution optimality that can be determined by solving instances of the problem with unique sequential divisions of the total budget. These trends can give the designer an idea of what budget divisions will result in favorable system designs. This data can also show which budget allocations to avoid. A key challenge is figuring out how to shed light on these nuances in the solution space and present them to designers.

Challenge 2: Tight-coupling Between the Producer/Consumer

Another key issue to contend with is how to rank the solutions to the producer MMKP problem. Per the definition of an MMKP co-design problem, the producer solution does not directly impart any value to the overall solution. The producer's benefit to a solution is its ability to make a good consumer solution viable. MMKP solvers must have a way of ranking solutions and items. The problem, however, is that the value of a producer solution or item cannot be calculated in isolation.

A consumer solution must already exist to calculate the value of a particular producer solution. For example, whether or not 1,024 kilobytes of memory are beneficial to the overall solution can only be ascertained by seeing if 1,024 kilobytes of memory are needed by the consumer solution. If the consumer solution does not need this much memory, then the memory produced by the item is not helpful. If the consumer solution is RAM starved, the item is desperately needed. A visualization of the problem is shown in Figure X.6.

The inability to rank producer solutions in isolation of consumer solutions is problematic because it creates a chicken and the egg problem. A valid consumer solution cannot be chosen if we do not know what resources are available for it to consume. At the same time, we cannot rank the value of producer solutions without a consumer solution as a context. This tight-coupling between the producer/consumer is a challenging problem.

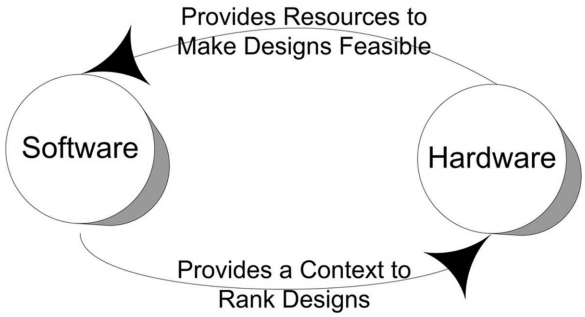


Figure X.6: Producer/Consumer MMKP Tight-coupling

The ASCENT Algorithm

This section presents our polynomial-time approximation algorithm, called the *Allocation-baSed Configuration ExploratioN Technique* (ASCENT), for solving MMKP co-design problems. The pseudo-code for the ASCENT algorithm is shown in Figure X.7 and explained throughout this section.

Producer/Consumer Knapsack Sizing

The first issue to contend with when solving an MMKP co-design problem is Challenge 2, which involves determining how to allocate sizes to the individual knapsacks. ASCENT addresses this problem by dividing the overall knapsack size budget into increments of size *D*. The size increment is a parameter provided by the user. ASCENT then iteratively increases the consumer's budget allocation (knapsack size) from 0% of the total budget to 100% of the total budget in steps of size *D*. The incremental expansion of the producer's budget can be seen in the `while` loop in code listing (1) of Figure X.7 and the incrementation of `ConsumerBudget` in code listing (8).

For example, if there is a total size budget of 100 and increments of size 10, ASCENT firsts assign 0 to the consumer and 100 to the producer, 10 and 90, 80 and 20, and so forth until 100% of the budget is assigned to the consumer. The allocation process is shown in Figure X.8. ASCENT includes both the 0%,100% and 100%,0% budget allocations to

```
MMKPProblem ConsumerMMKP
MMKPProblem ProducerMMKP
int StepSize
int ConsumerBudget = 0
int ProducerBudget = 100
int TotalBudget
Solution BestSolution
Solutions AllSolutions

while(ConsumerBudget <= TotalBudget)                              (1)
  IdealizedSolution = solveMMKPCostOnly(ConsumerMMKP,             (2)
                                        ConsumerBudget)
  double[] Ratios = calculateResourceRatios(IdealizedSolution)    (3)

  for each Item in ProducerMMKP                                   (4)
     for i = 0, i < Ratios.size, i++
        Item.Value += Ratios[i] * Item.ProducedResourceValue[i]

  ProducerBudget = TotalBudget - ConsumerBudget
  HardwareSolution =                                              (5)
               solveMMKPCostOnly(ProducerMMKP,
                                 ProducerBudget)

  int[] AvailableResources =                                      (6)
               HardwareSolution.ProducedResourceValues.Sum
  SoftwareSolution =                                              (7)
               solveMMKP(ProducerMMKP,
                         AvailableResources,
                         ConsumerBudget)
  ConsumerBudget += StepSize                                      (8)

  Solution = Tuple<SoftwareSolution,
                   HardwareSolution>
  Solutions.add(Solution)                                         (9)
  if(Solution.Value > BestSolution.Value)
    BestSolution = Solution

Return BestSolution and Solutions                                 (10)
```

Figure X.7: The ASCENT Algorithm

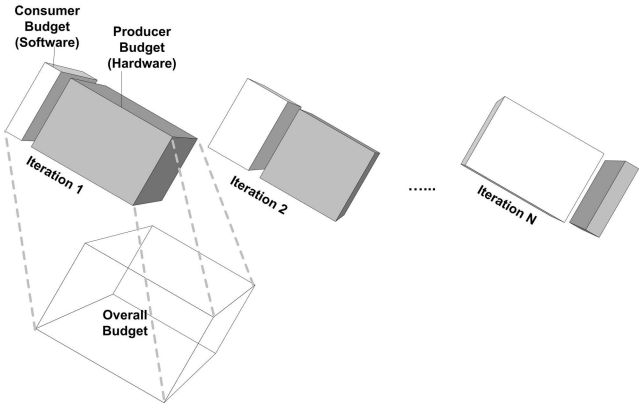


Figure X.8: Iteratively Allocating Budget to the Consumer Knapsack

handle cases where the optimal configuration includes producer or consumer items with zero cost.

Ranking Producer Solutions

At each allocation iteration, ASCENT has a fixed set of sizes for the two knapsacks. In each iteration, ASCENT must solve the coupling problem, which is: how do we rank producer solutions without a consumer solution. After the coupling is loosened, ASCENT can solve for a highly valued solution that fits the given knapsack size restrictions.

To break the tight-coupling between producer and consumer ordering, ASCENT employs a special heuristic. Once the knapsack size allocations are fixed, ASCENT solves for a maximal consumer solution that only considers the current size constraint of its knapsack and not produced/consumed resources. This step is shown in code listing (2) of Figure X.7.

The method `solveMMKPCostOnly` uses an arbitrary MMKP approximation algorithm to find a solution that only considers the consumer's budget. This approach is similar to asking "what would the best possible solution look like if there were unlimited produced/consumed resources." Once ASCENT has this idealized consumer solution, it calculates a metric for assigning a value to producer solutions.

The metric that ASCENT uses to assign value to producer items is: *how valuable are the resources of a producer item to the idealized consumer solution*. This metric is calculated by the `calculateResourceRatios` method call in code listing (3) of Figure X.7. We calculate the value of a resource as the amount of the resource consumed by the idealized consumer solution divided by the sum of the total resources consumed by the overall solution:

$$V_r = \frac{\sum_0^j Uc(c_j,k)}{\sum_0^k \sum_0^j Uc(c_j,k)}$$

In code listing (4) of Figure X.7, the resource ratios (V_r values) are known and each item in the producer MMKP problem is assigned a value by multiplying each of its provided

resource values by the corresponding ratio and summing these values:

$$valueof(p_l) = \sum_{0}^{k} (Up(p_l, k) * V_k)$$

The overall solving workflow at each budget allocation ratio is shown in Figure X.9.

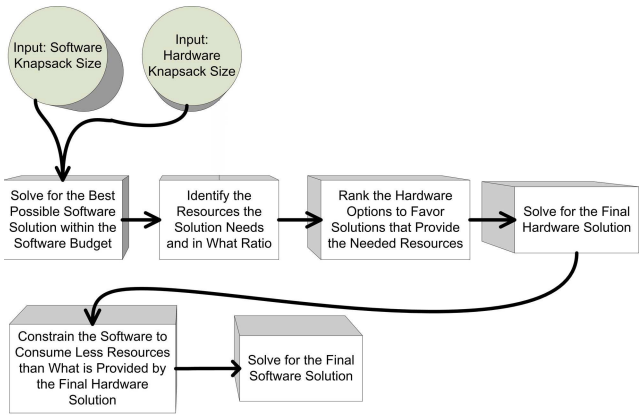


Figure X.9: ASCENT Solving Workflow at Each Budget Allocation Step

Solving the Individual MMKP Problems

Once sizes have been set for each knapsack and the valuation heuristic has been applied to the producer MMKP problem, existing MMKP solving approaches can be applied. First, the producer MMKP problem, with its new item values, is solved for an optimal solution, as shown in code listing (5) of Figure X.7. We use the `solveMMKPCostOnly` method to solve the producer problem since it does not consume any resources other than budget. In code listing (6), the consumer MMKP problem is then updated with constraints reflecting the maximum available amount of each resource produced by the solution from the producer MMKP problem. The consumer MMKP problem is then solved for an optimal solution in code listing (7). The producer and consumer solutions are then combined into the 2-tuple, $< p, c >$ and saved in code listing (9).

In each iteration, ASCENT assigns sizes to the producer and consumer knapsacks and

the solving process is repeated. A collection of the 2-tuple solutions is compiled during the process. The output of ASCENT, returned in code listing (10) of Figure X.7, is both the 2-tuple with the greatest value and the collection of 2-tuples. The overall solving approach is shown in Figure X.10.

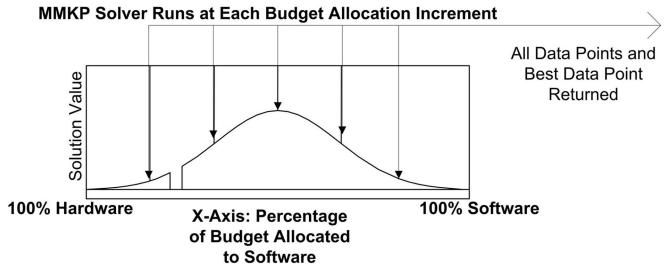


Figure X.10: ASCENT Solving Approach

The reason that the 2-tuples are saved and returned as part of the output is that they provide valuable information on the trends in the solution space of the co-design problem. Each 2-tuple contains a high-valued solution to the co-design problem at a particular ratio of knapsack sizes. This data can be used to graph and visualize how the overall solution value changes as a function of the ratio of knapsack sizes. This information can be used to ascertain a number of useful solution space characteristics, such as determining how much it costs to increase the value of a specific system property to a given level or finding the design with the highest value per unit of cost.

Algorithmic Complexity

The overall algorithmic complexity of ASCENT can be broken down as follows:

1. there are T iterations of ASCENT
2. in each iteration there are 3 invocations to an MMKP approximation algorithm
3. in each iteration, values of at most n producer items must be updated.

This breakdown yields an algorithmic complexity of O($T(n + MMKP)$), where MMKP is the algorithmic complexity of the chosen MMKP algorithm. With M-HEU (one of the most accurate MMKP approximation algorithms [12]) the algorithmic complexity is O($mn^2(l-1)^2$), where m is the number of resource types, n is the number of sets, and l is maximum items per set. Our experiments used $T = 100$ and found that it provided excellent results. With our experimental setup that used M-HEU, the overall algorithmic complexity was therefore O($100(mn^2(l-1)^2 + n)$). This algorithmic complexity is polynomial and thus ASCENT should be able to scale up to very large problems, such as the co-design of production satellite hardware and software.

Analysis of Empirical Results

This section presents empirical data we obtained from experiments using ASCENT to solve MMKP co-design problems. The empirical results demonstrate that ASCENT provides near optimal results. The results also show that ASCENT can not only provide near optimal designs for the co-design problems, such as the satellite example, but also scale to the large problem sizes of a production satellite design. Moreover, we show that the data sets generated by ASCENT—which contain high valued solutions at each budget allocation—can be used to perform a number of important search-based software engineering studies on the co-design solution space.

Each experiment used a total of 100 budget iterations ($T = 100$). We also used the M-HEU MMKP approximation algorithm as our MMKP solver. All experiments were conducted on an Apple Powerbook with a 2.4 GHz Intel Core 2 Duo processor, 2 gigabyes of RAM, running OS X version 10.4.11, and a 1.5 Java Virtual Machine (JVM) run in client mode. The JVM was launched with a maximum heap size of 64mb (-Xmx=64m).

MMKP Co-design Problem Generation

A key capability needed for the experiments was the ability to randomly generate MMKP co-design problems for test data. For each problem, we also needed to calculate how good ASCENT's solution was as a percentage of the optimal solution: $\frac{value of (ASCENT Solution)}{value of (Optimal Solution)}$. For small problems with less than 7 sets per MMKP problem, we were able to use a branch-and-bound linear programming (LP) [135] technique built on top of the Java Choco constraint solver (`choco-solver.net`) to derive the optimal solution.

For larger scale problems the LP technique was simply not feasible, *e*.g., solutions might take years to find. For larger problems, we developed a technique that randomly generated MMKP co-design problems with a few carefully crafted constraints so we knew the exact optimal answer. Others [12] have used this general approach, though with a different problem generation technique.

Ideally, we would prefer to generate completely random problems to test ASCENT. We our confident in the validity of this technique, however, for two reasons: (1) the trends we observed from smaller problems with truly random data were identical to those we saw in the data obtained from solving the generated problems and (2) the generated problems randomly placed the optimal items and randomly assigned their value and size so that the problems did not have a structure clearly amenable to the heuristics used by our MMKP approximation algorithm. We did not use Akbar's technique [12] because the problems it generated were susceptible to a greedy strategy.

Our problem generation technique worked by creating two MMKP problems for which we knew the exact optimal answer. First, we will discuss how we generated the individual MMKP problems. Let S be the set of MMKP sets for the problem, $\vec{R}$ be a K-dimensional vector describing the size of the knapsack, I_{ij} be the j_{th} item of the i_{th} set, $size(I_{ij},k)$ be the k_{th} component of I_{ij}'s size vector $\vec{Siz}_{ij}$, and $size(S,k)$ be the k_{th} component of the knapsack size vector, the problem generation technique for each MMKP problem worked as follows:

1. Randomly populate each set, $s \subset S$, with a number of items

2. Generate a random size, $\vec{R}$, for the knapsack

3. Randomly choose one item, $Iopt_i \subset OptItems$ from each set to be the optimal item. $Iopt_i$ is the optimal item in the i_{th} set.

4. Set the sizes of the items in *OptItems*, so that when added together they exactly consume all of the space in the knapsack:

$$(\forall k \subset R), (\sum_{0}^{i} size(Iopt_i, k)) = size(S, k)$$

5. Randomly generate a value, $Vopt_i$, for the optimal item, $Iopt_i$, in each set

6. Randomly generate a value delta variable, $V_d < min(Vopt_i)$, where $min(Vopt_i)$ is the optimal item with the smallest value

7. Randomly set the size and values of the remaining non-optimal items in the sets so that either:

   - The item has a greater value than the optimal item in its set. In this case, each component of the item's size vector, is greater than the corresponding component in the optimal item's size vector: $(\forall k \subset R), size(Iopt_i, k) < size(I_{ij}, k)$
   - The item has a smaller value than the optimal item's value minus V_d, $valueof(I_{ij}) < Vopt_i - V_d$. This constraint will be important in the next step. In this case, each component of the item's size vector is randomly generated.

At this point, we have a very random MMKP problem. What we have to do is further constrain the problem so that we can guarantee the items in *OptItems* are truly the optimal selection of items. Let $MaxV_i$ be the item with the highest value in the i_{th} set. We further constrain the problem as follows:

For each item $MaxV_i$, we reset the values of the items (if needed) to ensure that the sum of the differences between the max valued items in each set and the optimal item are less

than V_d:

$$\sum_{0}^{i}(MaxV_i - Vopt_i) < V_d$$

A visualization of this constraint is shown in Figure X.11.

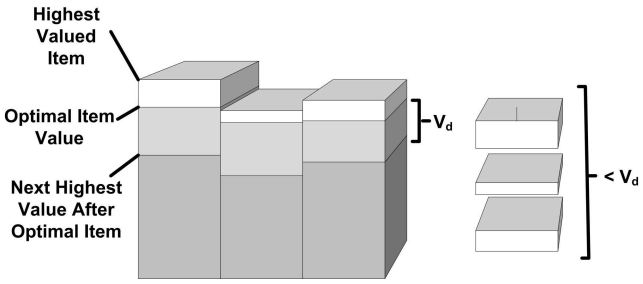


Figure X.11: A Visualization of V_d

This new valuation of the items guarantees that the items in *OptItems* are the optimal items. We can prove this property by showing that if it does not hold, there is a contradiction. Assume that there is some set of items, *Ibetter*, that fit into the knapsack and have a higher value. Let Vb_i be the value of the better item to choose than the optimal item in the i_{th} set. The sum of the values of the better items from each set must have a higher value than the optimal items.

The items $Ib_i \subset Ibetter$ must fit into the knapsack. We designed the problem so that the optimal items exactly fit into the knapsack and that any item with a higher value than an optimal item is also bigger. This design implies that at least one of the items in *Ibetter* is smaller and thus also has a smaller value, *Vsmall*, than the optimal item in its set (or *Ibetter* wouldn't fit). If there are Q sets in the MMKP problem, this implies that at most $Q-1$ items in *Ibetter* have a larger value than the optimal item in their set, and thus:

$$Vopt_Q + \sum_{0}^{Q-1} Vopt_i < Vsmall + \sum_{0}^{Q-1} Vb_i$$

We explicitly revalued the items so that:

$$\sum_{0}^{i}(MaxV_i - Vopt_i) < V_d$$

By subtracting the $\sum_0^{Q-1} Vopt_i$ from both sides, we get:

$$Vopt_Q < Vsmall + \sum_{0}^{Q-1}(Vb_i - Vopt_i)$$

the inequality will still hold if we substitute V_d in for $\sum_0^{Q-1}(Vb_i - Vopt_i)$, because V_d is larger:

$$Vopt_Q < Vsmall + V_d$$

$$Vopt_Q - V_d < Vsmall$$

which is a contradicton of the rule that we enforced for smaller items: $valueof(I_{ij}) < Vopt_i - V_d$

This problem generation technique creates MMKP problems with some important properties. First, the optimal item in each set will have a random number of larger and smaller valued items (or none) in its set. This property guarantees that a greedy strategy will not necessarily do well on the problems.

Moreover, the optimal item may not have the best ratio of value/size. For example, an item valued slightly smaller than the optimal item may consume significantly less space because its size was randomly generated. Many MMKP approximation algorithms use the value/size heuristic to choose items. Since there is no guarantee on how good the value/size of the optimal item is, MMKP approximation algorithms will not automatically do well on these problems.

To create an MMKP co-design problem where we know the optimal answer, we generate a single MMKP problem with a known optimal answer and split it into two MMKP problems to create the producer and consumer MMKP problems. To split the problem, two

new MMKP problems are created. One MMKP problem receives E of the sets from the original problem and the other problem receives the remaining sets. The total knapsack size for each problem is set to exactly the size required by the optimal items from its sets to fit. The sum of the two knapsack sizes will equal the original knapsack size. Since the overall knapsack size budget does not change, the original optimal items remain the overall optimal solution.

Next, we generate a set of produced/consumed resource values for the two MMKP problems. For the consumer problem, we randomly assign each item an amount of each produced resource $k \subset R$ that the item consumes. Let $TotalC(k)$ be the total amount of the resource k needed by the optimal consumer solution and $Vopt(p)$ be the optimal value for the producer MMKP problem. We take the consumer problem and calculate a resource production ratio, $Rp(k)$, where

$$Rp(k) = \frac{TotalC(k)}{Vopt(p)}$$

For each item, I_{ij}, in the producer problem, we assign it a production value for the resource k of: $Produced(k) = Rp(k) * valueof(I_{ij})$.

The optimal items have the highest feasible total value based on the given budget and the sum of their values times the resource production ratios exactly equals the needed value of each resource k:

$$TotalC(k) = \frac{TotalC(k)}{Vopt(p)} * \sum_{0}^{i} Vopt_i$$

Any other set of items must have a smaller total value and consequently not provide sufficient resources for the optimal set of consumer items. To complete the co-design problem, we set the total knapsack size budget to the sum of the sizes of the two individual knapsacks.

ASCENT Scalability and Optimality

Experiment 1: Comparing ASCENT scalability to an exact technique. When designing a satellite it is critical that designers can gauge the accuracy of their design techniques. Moreover, designers of a complicated satellite system need to know how different design techniques scale and which technique to use for a given problem size. This first set of experiments evalutes these questions for ASCENT and a branch-and-bound linear programming (LP) co-design technique.

Although LP solvers can find optimal solutions to MMKP co-design problems they have exponential time complexity. For large-scale co-design problems (such as designing a complicated climate monitoring satellite) LP solvers thus quickly become incapable of finding a solution in a reasonable time frame. We setup an experiment to compare the scalability of ASCENT to an LP technique. We randomly generated a series of problems ranging in size from 1 to 7 sets per hardware and software MMKP problem. Each set had 10 items. We tracked and compared the solving time for ASCENT and the LP technique as the number of sets grew. Figure X.12 presents the results from the experiment. As shown

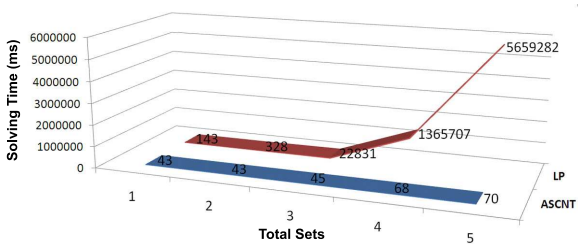


Figure X.12: Solving Time for ASCENT vs. LP

by the results, ASCENT scales significantly better than an LP-based approach.

Experiment 2: Testing ASCENT's solution optimality. Clearly, scalability alone is not the only characteristic of a good approximation algorithm. A good approximation algorithm must also provides very optimal results. We created an experiment to test the accuracy of ASCENT's solutions. We compared the value of ASCENT's answer to the

optimal answer,

$$\frac{valueof(ASCENTSolution)}{valueof(OptimalSolution)}$$

for 50 different MMKP co-design problem sizes with 3 items per set. For each size co-design problem, we solved 50 different problem instances and averaged the results.

It is often suggested, due to the Central Limit Theorem [73], to use a sample size of 30 or larger to produce an approximately normal data distribution [64]. We chose a sample size of 50 to remain well above this recommended minimum sample size. The largest problems, with 50 sets per MMKP problem, would be the equivalent of a satellite with 50 points of software variability and an additional 50 points of hardware variability.

For problems with less than 7 sets per MMKP problem, we compared against the optimal answer produced with an LP solver. We chose a low number of items per set to decrease the time required by the LP solver and make the experiment feasible. For problems with more than 7 sets, which could not be solved in a timely manner with the LP technique, we used our co-design problem generation technique. The problem generation technique allowed us to create random MMKP co-design problems that we knew the exact optimal answer for and could compare against ASCENT's answer.

Figure X.13 shows the results of the experiment to test ASCENT's solution value verusus the optimal value over 50 MMKP co-design problem sizes. With 5 sets, ASCENT

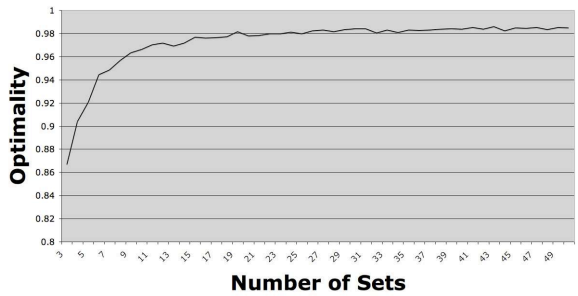


Figure X.13: Solution Optimality vs Number of Sets

produces answers that average 90% optimal. With 7 sets, the answers average ∼95% optimal. Beyond 20 sets, the average optimality is ∼98% and continues to improve. These results are similar to MMKP approximation algorithms, such as M-HEU, that also improve with increasing numbers of sets [12]. We also found that increasing the number of items per set also increased the optimality, which parallels the results for our solver M-HEU [12].

Experiment 3: Measuring ASCENT's solution space snapshot accuracy. As part of the solving process, ASCENT not only returns the optimal valued solution for a co-design problem but it also produces a data set to graph the optimal answer at each budget allocation. For the satellite example, the graph would show designers the design with the highest image processing accuracy for each ratio of budget allocation to software and hardware. We created an experiment to test how optimal each data point in this graph was.

For this experiment, we generated 100 co-design problems with less than 7 sets per MMKP problem and compared ASCENT's answer at each budget allocation to the optimal answer derived using an LP technique (more sets improves ASCENT's accuracy). For problems with 7 sets divided into 98 different budget allocations, ASCENT finds the same, optimal solution as the LP solver more than 85% of the time. Figure X.14 shows an example that compares the solution space graph produced by ASCENT to a solution space graph produced with an LP technique. The X-axis shows the percentage of the budget allocated to

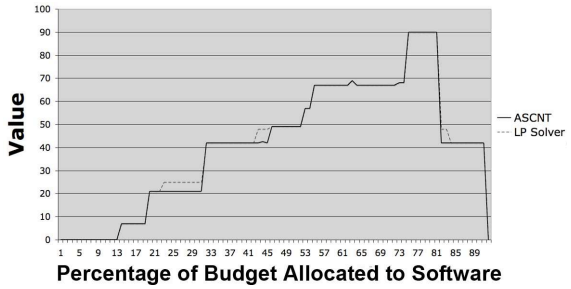


Figure X.14: Solution Value vs. Budget Allocation

the software (consumer) MMKP problem. The Y-axis shows the total value of the MMKP

co-design problem solution. The ASCENT solution space graph closely matches the actual solution space graph produced with the LP technique.

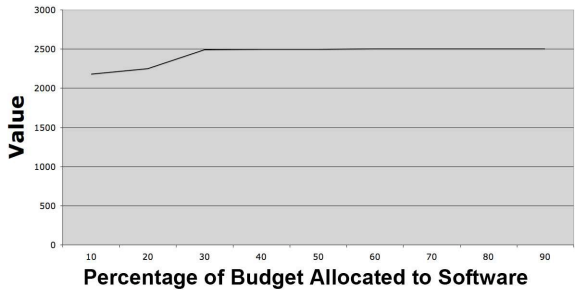
Solution Space Snapshot Resolution

Experiment 4: Demonstrating the importance of solution space snapshot resolution. A complicated challenge of applying search-based software engineering to hardware/-software co-design problems is that design decisions are rarely as straightforward as identifying the design configuration that maximizes a specific property. For example, if one satellite configuration provides 98% of the accuracy of the most optimal configuration for 50% less cost, designers are likely to choose it. If designers have extensive experience in hardware development, they may favor a solution that is marginally more expensive but allocates more of the development to hardware, which they know well. Search-based software engineering techniques should therefore allow designers to iteratively tease these desired designs out of the solution space.

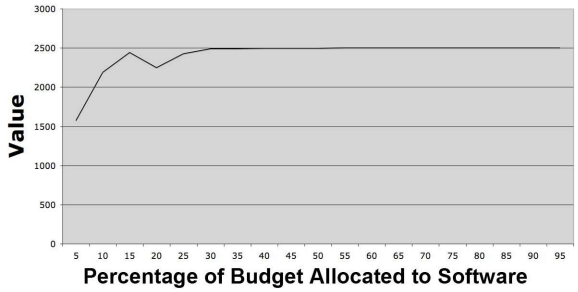
ASCENT has a number of capabilities beyond simply finding the optimal solution for a problem to help designers find desirable solutions. First, as we describe below, ASCENT can be adjusted to produce different resolution images of the solution space by adjusting the granularity of the budget allocation steps (*e*.g., make smaller and more allocation changes).

The granularity of the step size greatly impacts the resolution or detail that can be seen in the solution space. To obtain the most accurate and informative solution space image, a small step size should be used. Figure X.15(a) shows a solution space graph generated through ASCENT using 10 allocation steps. The X-axis is the percentage of budget allocated to software, the Y-axis is the total value of the solution. It appears that any allocation of 30% or more of the budget to software will produce a satellite with optimal image processing accuracy.

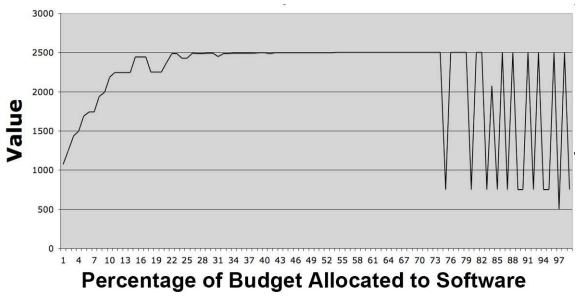
Figure X.15(b), however, shows the graph that results from solving the same problem with a 20 allocation steps. It is important to note that while allocating 30% or more of the



(a) Low Resolution Solution Space Snapshot



(b) Medium Resolution Solution Space Snapshot



(c) High Resolution Solution Space Snapshot

Figure X.15: A Solution Space Graph at Varying Resolutions

budget to software still results in an optimal solution, there is another point that was absent from the previous graph. It can clearly be seen that an allocation of 15% of the budget for software will also result in a near optimal solution, which is an unanticipated good solution that favors hardware.

The importance of a small step size is further demonstrated in Figure X.15(c), which was produced with 100 allocation steps. Both previous graphs also suggest that any allocation of greater than 30% for software would result in an optimal satellite design. Figure X.15(c) shows that there are many pitfalls in the 70% to 99% range that must be avoided. At these precise budget allocation points, there is not a good combination of hardware and software that will produce a good solution.

This result may seem counter-intuitive. At these points, the previous good hardware solution is too expensive, but a different more expensive software configuration with less resource consumption to fit on the cheaper available hardware configurations is also not within budget. If any of these software allocation percentages were chosen arbitrarily without creating a high quality graph of the solution space, the designer could unknowingly create a system that has 25% of the value for the same cost.

Solution Space Analysis with ASCENT

Although ASCENT's ability to provide variable resolution solution space images is important, its greatest value stems from the variety of questions that can be answered from its output data. In the following results, we present representative solution space analyses that can be performed with ASCENT's output data.

Design analysis 1: Identifying low-cost viable designs. A common software engineering scenario is that a design need not necessarily be optimal as long as it provides a minimum required value or capability. For example, satellite designers want to find the cheapest designs that provide the required level of image processing accuracy. Figure X.16(a) shows a graph that can be produced by taking the output data from ASCENT

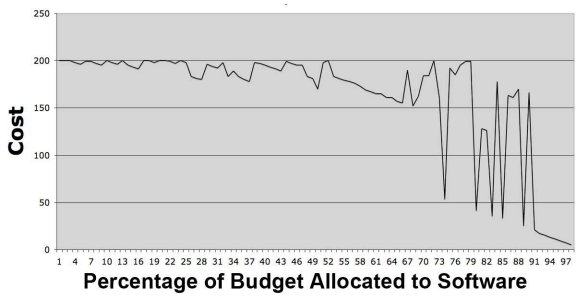
and graphing total actual solution cost as a function of budget allocation, rather than graphing value as a function of budget allocation. This graph allows designers to ascertain key low cost designs in the solution space and can be further filtered to eliminate any solutions that do not meet a minimum value threshold. The resulting graph allows designers to find the lowest cost satellite co-design solution with a given image processing accuracy.

Design analysis 2: Determining budget allocation ratios. An important question to ask when designing a system is what budget allocations and solutions give the most value per unit of cost. In terms of the satellite example, the question would be what design gives the most accuracy for the money. Figure X.16(c) shows another set of ASCENT output data that has been regraphed to show $\frac{value}{cost}$ as a function of the percentage of the budget allocated to software. It can clearly be seen that the designs with the best ratio of value to cost assign more of the value to software. This graph can also easily be filtered to eliminate designs that do not provide a minimum level of value.

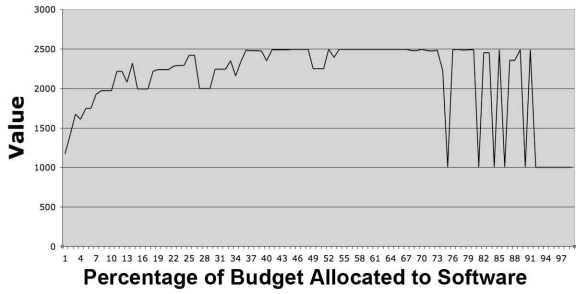
Design analysis 3: Finding designs that produce budget surpluses. Designers may wish to know how the resource slack values, such as how much RAM is unused, with different satellite designs. Another related question is how much of the budget will be left-over for designs that provides a specified minimal level of image processing accuracy. We can use the same ASCENT output data to graph the budget surplus at a range of allocation values.

Figure X.16(d) shows the budget surplus from choosing various designs. The graph has been filtered to adhere to a requirement that the solution provide a value of at least 1600. Any data point with a value of less than 1600 has had its surplus set to 0. Looking at the graph, we can see that the cheapest design that provides a value of at least 1,600 is found with a budget allocation of 80% software and 20% hardware. This design has a value of 1,600 and produces budget savings of 37%.

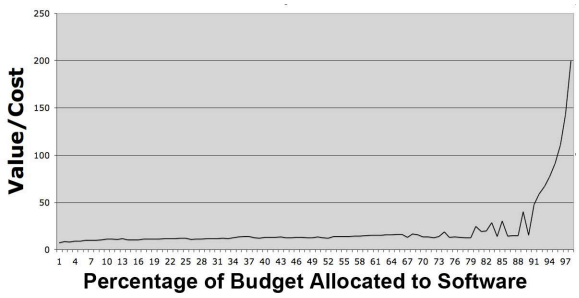
Design analysis 4: Evaluating design upgrade/downgrade cost. In some situations, designers may have a given solution and want to know how much it will cost or save to



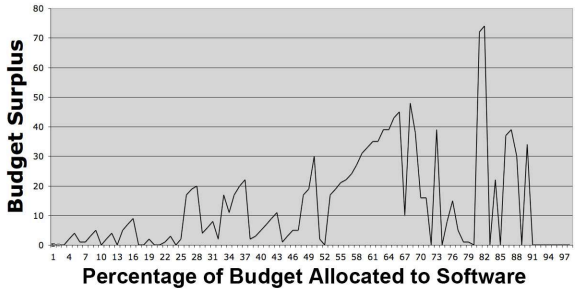
(a) Solution Cost vs. Budget Allocation



(b) Solution Value vs. Budget Allocation



(c) Cost Effectiveness vs. Budget Allocation



(d) Budget Surplus vs. Budget Allocation

Figure X.16: Satellite Design Solution Space Analysis Graphs

upgrade or downgrade the solution to a different image processing accuracy. For example, designers may be asked to provide a range of satellite options for their superiors that show what level of image processing accuracy they can provide at a number of price points. Figure X.17 depicts another view of the ASCENT data that shows how cost varies in relation to the minimum required solution value. This graph shows that 5 cost units can finance a

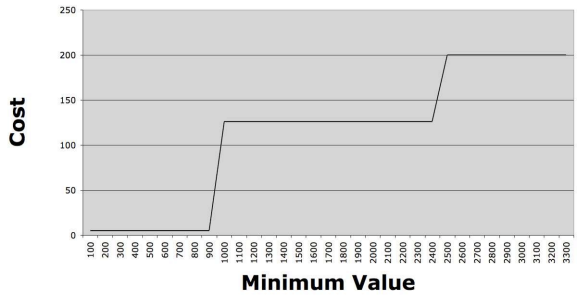


Figure X.17: Cost of Increasing Solution Value

design with a value up to 900, but a design of a value of 1,000 units will cost at least 124 cost units. This information graph demonstrates the increased financial burden of requiring a slightly higher valued design. Alternatively, if the necessary value of the system is near the left edge of one of these plateaus, designers can make an informed decision on whether the increased value justifies the significantly increased cost.

Summary of Empirical Results

The following is a summary of the empirical results presented above.

• **ASCENT Produces Answers that are 98% Optimal:** As seen from the results in Figure X.13, ASCENT generates answers that average 98% optimal for problems with a large number of sets in each MMKP problem. This result implies that ASCENT will perform well on large-scale MMKP co-design problems, such as the design of a large and complex satellite. Moreover, the larger the problem, the more accurate ASCENT's results.

Systems of this scale would be nearly impossible to optimize without the search-based software engineering method provided by ASCENT.

• **High Resolution Solution Space Snapshots Can Identify Near-optimal Alternative Solutions:** Another important result is that we demonstrated that by capturing a high resolution solution space snapshot we can identify unanticipated near optimal designs. These unanticipated nearly optimal designs correspond to peaks in the solution space graph at local maxima. In future work, we plan to develop algorithms that automatically increase the solution space snapshot resolution at and around these local maxima. Solving the large numbers of problems to produce a highly detailed solution space snapshot is too time-consuming and error-prone to perform manually.

• **ASCENT Output Data Can Answer Numerous Cost-based Design Questions to Iteratively Improve Solution Design:** Since many design criteria cannot be completely formalized for a search solver, search-based software engineering should allow desigeners to iteratively hone in on the solutions they desire. The results demonstrated that each run of ASCENT allowed designers to answer key questions related to the allocation of budget to hardware and software. For example, designers of a satellite could answer questions such as *what allocation of budget to hardware and software produces the highest valued solution*. Designers can also answer other previously difficult questions related to how expensive it is to produce a solution with a given optimality.

CHAPTER XI

AUTOMATED CONFIGURATION DEBUGGING

Challenge Overview

This chapter investigates the problems that arise when invalid configurations are created by modelers. Existing research has focused on ensuring that features chosen from feature models are correct and consistent with the SPL and variant requirements. For example, work has been done on using boolean circuit satisfiability techniques [93] or Constraint Satisfaction Problems (CSPs) [22,144] to automate the derivation of a feature set that meets a requirement set. Numerous tools have also been developed, such as Big Lever Software Gears [31], Pure::variants [23], FeAture Model Analyser (FAMA) [21], and the Feature Model Plug-in [46], to support the construction of feature models and correct selection of feature configurations.

Introduction

Regardless of what tools and processes are used to configure SPL variants, however, there is always the possibility that mistakes will occur. For example, large SPLs often use *staged configuration* [48, 49], where features are selected in multiple stages to form a complete configuration iteratively, rather than choosing all features at once. At a late stage in the configuration process, developers may realize that a critically needed feature cannot be selected due to one or numerous decisions in some previous stages. It is hard to debug a configuration to figure out how to change decisions in previous stages to make the critical feature selectable [18].

Another challenging situation can arise when multiple participants are involved in the feature selection process and their desired feature selections conflict. For example, hardware developers for an automobile may desire a lower cost set of Electronic Control Units

(ECUs) that cannot support the features needed by the software developer's embedded controller code. In these situations, methods are needed to evaluate and debug conflicts between participants. Methods are also needed to recommend modifications to the participants feature selections to make them compatible.

Although prior research has shown how to identify flawed configurations [17, 93], conventional debugging mechanisms cannot pinpoint configuration errors and identifying corrective actions. More specifically, techniques are lacking that can take an arbitrary flawed configuration and produce the minimal set of feature selections and deselections to bring the configuration to a valid state. This challenge focuses on addressing these gaps in existing research.

Challenges of Debugging Feature Model Configurations

This section evaluates different challenges that arise in realistic configuration scenarios.

Challenge 1: Staged Configuration Errors

Staged configuration is a configuration process whereby developers iteratively select features to reduce the variability in a feature model until a variant is constructed. Czarnecki et al. [48, 49] use the context of software supply chains for embedded software in automobiles to demonstrate the need for staged configuration. In the first stage, software vendors provide software components that can be provided in different configurations to actuate brakes, control infotainment systems, etc. In the second stage, hardware vendors of the Electronic Control Units (ECUs) that the software runs on must provide ECUs with the correct features and configuration to support the software components selected in the first stage.

The challenge with staged configuration is that feature selection decisions made at some point in time T have ramifications on the decisions made at all points in time $T' > T$. For example, it is possible for software vendors to choose a set of software component

features for which there are no valid ECU configurations in the second configuration stage. Identifying the fewest number of configuration modifications to remedy the error is hard because there can be significant distance between T and T'.

This challenge also appears in larger models, such as those for software to control the automation of continuous casting in steel manufacture [116]. In large-scale models, configuration mimics staged configuration since developers cannot immediately understand the ramifications of their current decisions. At some later decision point, critical features that developers need may no longer be selectable due to some previous choice. Again, it is hard to identify the minimal set of configuration decisions to reverse in this scenario.

Challenge 2: Mediating Conflicts

In many situations the desired features and needs of multiple stakeholders involved in configuring an SPL variant may conflict. For example, when configuring automotive systems, software developers may want a series of software component configurations that cannot be supported by the ECU configurations proposed by the hardware developers. To each party, their individual needs are critical and finding the middle ground to integrate the two is hard.

Another conflict scenario arises when configuration decisions made for an SPL variant must be reconciled with constraints of the legacy environment in which it will run. For example, when configuring automotive software for next year's car model, a variant may initially be configured to provide the most desired customer features, such as digital infotainment. New model cars are rarely complete redesigns, however, so developers must determine out how to run new software configurations on existing ECU configurations from previous models. If the new software configuration is not compatible with the legacy ECU configuration, developers must derive the lowest cost set of modifications to either the new software or the legacy ECU configuration.

Challenge 3: Viewpoint-dependent Errors

The feature labeled as the source of an error in a feature model configuration may vary depending on the viewpoint used to debug it. In the feature model shown in Figure XI.1, for example, if a configuration is created that includes both *Non-ABS Controller* and *1 Mbit/s CAN Bus*, either feature can be viewed as the feature that is the source of the error.

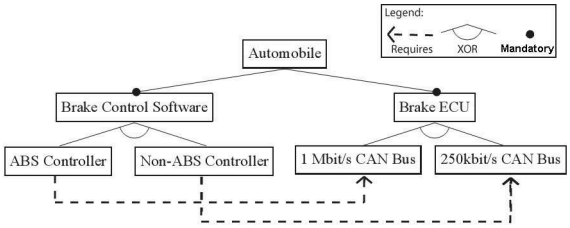


Figure XI.1: Simple Feature Model for an Automobile

If we debug the configuration from the viewpoint that software trumps ECU hardware decisions, then the *1 Mbit/s CAN Bus* feature is the error. If we assume that ECU decisions precede software, however, then the *Non-ABS Controller* feature is the error.

A feature model may therefore require debugging from multiple viewpoints since diagnosing the feature that causes an error in a feature model depends on the viewpoint used to debug it. For small feature models, debugging from different viewpoints is relatively simple. When feature models contain hundreds or thousands of features, the complexity of diagnosing a configuration from multiple viewpoints increases greatly.

Solution Approach

Our solution approach, called Configuration Understanding and REmedy (CURE), is based on creating automated SPL variant diagnosis tools. Developers can use these tools to identify the minimal set of features to select or deselect to transform an invalid configuration into a valid configuration. Moreover, depending on the input provided to CURE, a flawed configuration can be debugged from different viewpoints or conflicts between multiple stakeholder decisions in a configuration process can be mediated.

The key component of CURE is the application of a CSP-based error diagnostic technique. In prior work, Benavides et al. [22] have shown how feature models can be transformed into CSPs to automate feature selection with a constraint solver [77]. Trinidad et al. [131] subsequently described how to extend this CSP technique to identify *full mandatory features, void features,* and *dead feature models* using Reiter's theory of diagnosis [119]. This section presents an alternate diagnostic model for deriving the minimum set of features that should be selected or deselected to eliminate a conflict in a feature configuration.

Background: Feature Models and Configurations as CSPs

A CSP is a set of variables and a set of constraints over those variables. For example, $A+B \leq 3$ is a CSP involving the integer variables A and B. The goal of a constraint solver is to find a valid *labeling* (set of variable values) that simultaneously satisfies all constraints in the CSP. $(A = 1, B = 2)$ is thus a valid labeling of the CSP.

To build the CSP for the error diagnosis technique, we construct a set of variables, F, representing the features in the feature model. Each configuration of the feature model is a set of values for these variables, where a value of 1 indicates the feature is present in the configuration and a value of 0 indicates it is not present. More formally, a configuration is a labeling of F, such that for each variable $f_i \subset F$, $f_i = 1$ indicates that the i_{th} feature in the feature model is selected in the configuration. Correspondingly, $f_i = 0$ implies that the feature is not selected.

Given an arbitrary configuration of a feature model as a labeling of the F variables, developers need the ability to ensure the correctness of the configuration. To achieve this constraint checking ability, each variable f_i is associated with one or more constraints corresponding to the configuration rules in the feature model. For example, if f_j is a required subfeature of f_i, then the CSP would contain the constraint: $f_i = 1 \Leftrightarrow f_j = 1$.

Configuration rules from the feature model are captured in the constraint set C. For

any given feature model configuration described by a labeling of F, the correctness of the configuration can be determined by seeing if the labeling satisfies all constraints in C. A more detailed description of the steps for transforming a feature model to a CSP are described in [22].

Configuration Diagnostic CSP

When diagnosing configuration conflicts, developers need a list of features that should be selected or deselected to make an invalid configuration a valid configuration. The output of CURE is this list of features to select and deselect, as shown in Figure XI.2.

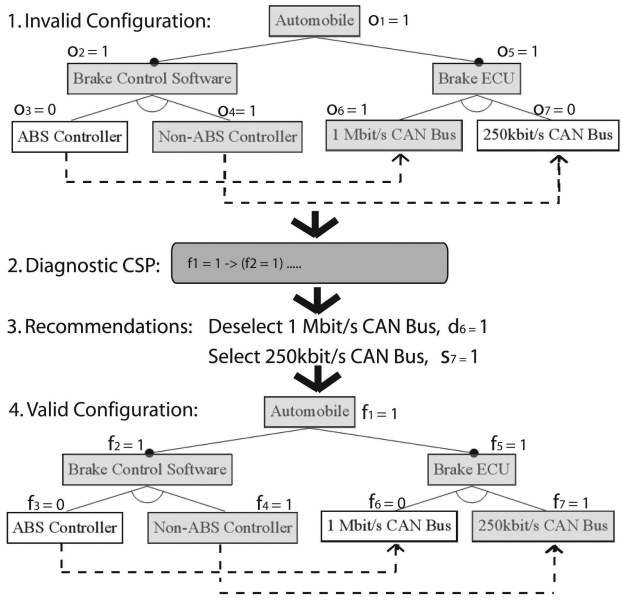


Figure XI.2: Diagnostic Technique Architecture for CURE

In Step 1 of Figure XI.2, the rules of the feature model and the current invalid configuration are transformed into a CSP. For example, $o_1 = 1$ because the *Automobile* feature is selected in the current invalid configuration. In Step 2, the solver derives a labeling of the diagnostic CSP. Step 3 takes the output of the CSP labeling and transforms it into a series of recommendations of features to select or deselect to turn the invalid configuration into a valid configuration. Finally, in Step 4, the recommendations are applied to the

invalid configuration to create a valid configuration where each variable f_i equals 1 if the corresponding feature is selected in the new and valid configuration. For example, $f_7 = 1$, meaning that the *250 Kbit/s CAN Bus* is selected in the new valid configuration.

To enable the constraint solver to recommend features to select and deselect, two new sets of recommendation variables, *S* and *D*, are introduced to capture the features that need to be selected and deselected, respectively, to reach a valid configuration. For example, a value of 1 for variable $s_i \subset S$ indicates that the feature f_i should be added to the current configuration. Similarly, $d_i = 1$ implies that the feature f_i should be removed from the configuration.

Thus, for each feature $f_i \subset F$, there are variables $s_i \subset S$ and $d_i \subset D$. After the diagnosis CSP is labeled, the values of *S* and *D* serve as the output recommendations to the user as to what features to add or remove from the current configuration, as shown in Table 1. This table shows the complete inputs and outputs to diagnose the invalid configuration scenario shown in Figure XI.2.

The next step is to allow developers to input their current configuration into the solver for diagnosis. Rather than directly setting values for the variables in F, developers use a special set of input variables called the *observations*, which are contained in the set of variables *O*. For each feature f_i present in the current flawed configuration, $o_i = 1$; if f_i is not selected in the current invalid configuration, $o_i = 0$. Table 1 shows how observations capture the current invalid configuration provided as input to the solver. Observations can also be made for a correct configuration, in which case CURE will state that no changes are needed. The rest of this chapter assumes that the observations represent an invalid configuration.

To diagnose the CSP, we want to find an alternate but valid configuration of the feature model and suggest a series of changes to the current invalid configuration to reach the valid configuration. A valid configuration is a labeling of the variables in F (a configuration)

Variables	
Variable Explanations	$f_i \subset F$: feature variables for the valid configuration that will be transitioned to; $o_i \subset O$: the features selected ($o_i = 1$) in the current invalid configuration; $s_i \subset S$: features to select ($s_i = 1$) to reach the valid configuration; $d_i \subset D$: features to deselect ($d_i = 1$) to reach the valid configuration
Inputs	
Current Config.	$o_1 = 1, o_2 = 1, o_3 = 0, o_4 = 1, o_5 = 1, o_6 = 1, o_7 = 0$
Feature Model Rules	$f_1 = 1 \Leftrightarrow (f_2 = 1), f_1 = 1 \Leftrightarrow (f_5 = 1),\ f_2 = 1 \Rightarrow (f_3 = 1) \oplus (f_4 = 1),\ f_5 = 1 \Rightarrow (f_6 = 1) \oplus (f_7 = 1), (f_6 = 1) \vee (f_7 = 1) \Rightarrow (f_5 = 1), (f_3 = 1) \vee (f_4 = 1) \Rightarrow (f_2 = 1), f_3 = 1 \Rightarrow (f_6 = 1), f_4 = 1 \Rightarrow (f_7 = 1)$
Diagnostic Rules	$(f_i \subset F \mid \{(f_i = 1) \Rightarrow (o_i = 1 \oplus s_i = 1) \wedge (d_i = 0), (f_i = 0) \Rightarrow (o_i = 0 \oplus d_i = 1) \wedge (s_i = 0)\})$
Outputs	
Features to Select	$s_1 = 0, s_2 = 0, s_3 = 0, s_4 = 0, s_5 = 0, s_6 = 0, s_7 = 1$
Features to Deselect	$d_1 = 0, d_2 = 0, d_3 = 0, d_4 = 0, d_5 = 0, d_6 = 1, d_7 = 0$
New Valid Config.	$f_1 = 1, f_2 = 1, f_3 = 0, f_4 = 1, f_5 = 1, f_6 = 0, f_7 = 1$

Table XI.1: Diagnostic CSP Construction

such that all of the feature model constraints are satisfied. For each variable f_i, the value should be 1 if the feature is present in the new valid configuration that will be transitioned to. If a feature is not in the new configuration, f_i should equal 0.

We always require $f_1 = 1$ to ensure that the root feature is always selected. For void feature models, there will be no valid solution and the solver will respond that no solution was found. CURE could be used to detect void feature models but it would be more appropriate to use a technique designed for this purpose, such as [131].

One key input to CURE is the CSP describing the set of all valid feature selections from the feature model (the Feature Model Rules in Table 1). Since these valid feature selections are described as constraints over the variables in F, **a valid labeling of F will always yield a valid feature selection**. Once a valid labeling of F is found, the goal is to determine how to modify the labeling of O to match the valid feature selection denoted by the labeling of F.

First, a constraint must be introduced to model when a feature in the current invalid configuration needs to be deselected to reach the correct configuration. If the i_{th}feature is included in the current configuration ($o_i = 1$), but is not in the new valid configuration ($f_i = 0$), we want the solver to recommend that it be deselected ($d_i = 1$). For every feature, we introduce the following constraint to determine if the i_{th} feature in O needs to be deselected[1]:

$$(f_i = 0) \Rightarrow (o_i = 0 \oplus d_i = 1) \wedge (s_i = 0)$$

If f_i is not selected in the correct configuration ($f_i = 0$), then either the feature was also not selected in the current invalid configuration ($o_i = 0$), or the feature needs to be deselected ($d_i = 1$). Furthermore, if a feature is not needed in the valid configuration ($f_i = 0$) then clearly it should not be a recommended selection ($s_i = 0$).

The solver must also recommend features to select. If the i_{th} feature is selected in the correct and valid configuration $f_i = 1$, and not selected in the current invalid configuration ($o_i = 0$), then it needs to be selected ($s_i = 1$). For each feature, we introduce the constraint:

$$(f_i = 1) \Rightarrow (o_i = 1 \oplus s_i = 1) \wedge (d_i = 0)$$

If a feature is needed by the correct configuration ($f_i = 1$), then either the feature was present in the invalid configuration ($o_i = 1$) or the feature was not present in the invalid

[1]The symbol "$\oplus$" denotes *exclusive or*

configuration and needs to be selected ($s_i = 1$). Clearly, a feature should not be deselected if $f_i = 1$ and thus $d_i = 0$.

The state of each feature, o_i, in the current invalid configuration is compared against the correct state of the feature, f_i, in the valid feature configuration. The behavior of each comparison can fall into four cases:

1. **A feature is selected and does not need to be deselected.** If the i_{th} feature is in the current invalid configuration ($o_i = 1$), and also in the new valid configuration ($f_i = 1$), no changes need be made to it ($s_i = 0$, $d_i = 0$)

2. **A feature is selected and needs to be deselected.** If the i_{th} feature is in the current invalid configuration ($o_i = 1$) but not in the new valid configuration ($f_i = 0$), it must be deselected ($d_i = 1$)

3. **A feature is not selected and does not need to be selected.** If the i_{th} feature is not in the current invalid configuration ($o_i = 0$) and is also not needed in the new configuration ($f_i = 0$) it should remain unchanged ($s_i = 0$, $d_i = 0$)

4. **A feature is not selected and needs to be selected.** If the i_{th} feature is not selected in the current invalid configuration ($o_i = 0$) but is present in the new correct configuration ($f_i = 1$), it must be selected ($s_i = 1$)

Optimal Diagnosis Method

The next step in the CURE diagnosis process is to use the solver to label the variables and produce a series of recommendations. For any given configuration with a conflict, there may be multiple possible ways to eliminate the problem. For example, in the automotive example, the valid corrective actions were either (1) remove the *1 Mbit/s CAN Bus* and select the *250 Kbit/s CAN Bus* or (1) remove the *Non-ABS Controller* and select the *ABS Controller*. We must therefore tell the solver how to select which of the (many) possible corrective solutions to suggest to developers.

The most basic suggestion selection criteria developers can use to guide the solver's diagnosis is to tell it to minimize the number of changes to make to the current configuration, *i.e.*, prefer suggestions that require changing as few things as possible in the current invalid configuration. To implement this approach, we solve for a CSP labeling that minimizes the sum of variables in $S \cup D$, which is the total number of changes that the solution requires the developer to make. By minizing this sum we therefore minimize the total number of required changes.

Each labeling of the diagnostic CSP will produce two sets of features corresponding to the features that should be selected (S) and deselected (D) to reach the new valid configuration. Developers can ask the solver to cycle through the different potential labelings of the diagnostic CSP to evaluate potential remedies. Furthermore, each new labeling (new diagnosis) also causes the solver to backtrack and create new values for F, which allows developers to evaluate not only the suggested modifications but the configuration that the remedy will produce. Another way to further refine the guidance for the diagnosis is to constrain the new state captured in the labeling of F.

Table 1 shows a complete set of inputs and output suggestions for diagnosing the automotive software example. If there are multiple labelings of the CSP, initially only one will be returned. After the first solution has been found, however, the solver can much more efficiently cycle through the other equally ranked sets of corrective suggestions.

Solution Extensibility and Benefits

This section presents different benefits of CURE and possible ways of extending it.

Bounding Diagnostic Method

Due to time constraints, it may not be possible to find the optimal number of changes for extremely large feature models. In these cases, a more scalable approach is to attempt to find any suggestion that requires fewer than K changes or with a cost less than K. Rather

than directly asking for an optimal answer, we add the following constraint to the CSP and ask the solver for any solution:

$$\sum_{i=1}^{n} s_i + d_i \leq K$$

The sum of all variables $s_i \subset S$ and $d_i \subset D$ represents the total number of feature selections and deselections that need to be made to reach the new valid configuration. Therefore, the sum of both of these sets is the total number of modifications that must be made to the original invalid configuration. The new constraint, ensures that the solver only accepts diagnosis solutions that require the developer to make K or fewer changes to the invalid solution.

The solver is asked for **any** answer that meets the new constraints. In return, the solver will provide a solution that is not necessarily perfect, but which fits our tolerance for change. If no solution is found, we can increment K by a factor and re-invoke the solver or reassess our requirements. As earlier, searching for a bounded solution rather than an optimal solution is significantly faster.

If the solver cannot find a diagnosis that makes fewer than K modifications, it will state that there is no valid solution that fits a K change budget.

Debugging from Different Viewpoints

As we discussed previously, we need the ability to debug the configuration from different viewpoints. Each viewpoint represents a set of features that the solver should avoid suggesting to add or remove from the current configuration. For example, using the automobile scenario, the solver can debug the problem from the point of view that hardware decisions trump software by telling the solver not to suggest selecting or deselecting any hardware features.

Debugging from a viewpoint works by pre-assigning values for a subset of the variables in F and O. For example, to force the feature f_i currently in the configuration to remain

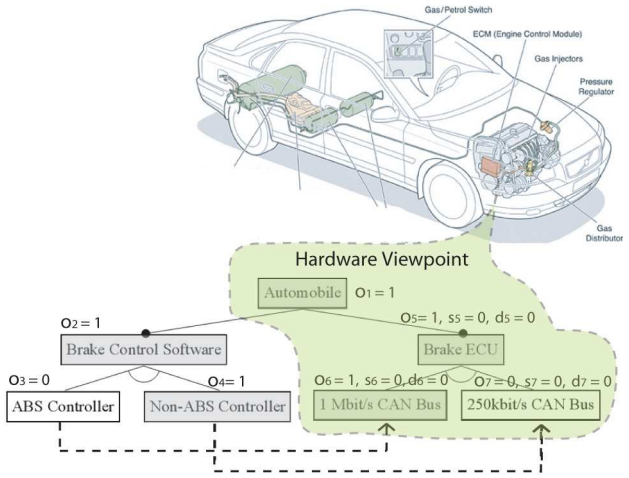


Figure XI.3: Debugging from a Viewpoint

unaltered by the diagnosis, the values $f_i = 1$ and $o_i = 1$ are provided to the solver. Since $(f_i = 1) \Rightarrow (o_i = 1 \oplus s_i = 1) \wedge (d_i = 0)$, pre-assigning these values will force the solver to label $s_i = 0$ and $d_i = 0$.

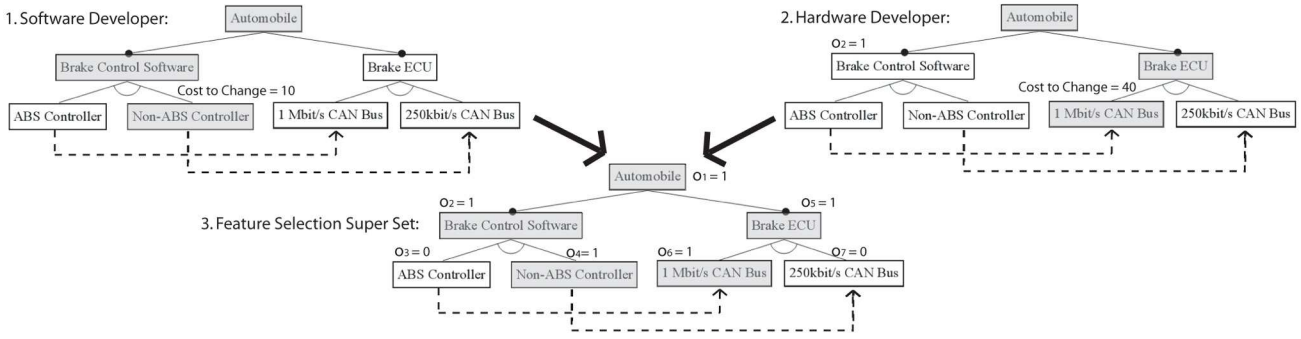


Figure XI.4: Constructing the Feature Selection Superset for Conflict Mediation

To debug from a given point of view, for each feature f_v, in that viewpoint, we first add the constraints, $f_v = 1$, $o_v = 1$, $s_v = 0$, and $d_v = 0$, as shown in Figure XI.3. The solver then derives a diagnosis that recommends alterations to other features in the configuration and maintains the state of each feature f_v. The CURE diagnostic model can therefore be used to debug from different viewpoints and address Challenge 3.

Pre-assigning values for variables in F and O can also be used to debug staged configuration errors from Challenge 1. With staged configuration errors, at some point in time T', developers need to select a feature that is in conflict with one or more features selected at time $T < T'$. To debug this type of conflict, developers pre-assign the desired (but currently unselectable) feature at time T' the value of 1 for its o_i and f_i variables. Developers can also pre-assign values for one or more other features decisions from previous stages of the configuration that must not be altered. The solver is then invoked to find a configuration that includes the desired feature at T' and minimizes the number of changes to feature configuration decisions that were made at all points in time $T < T'$.

Cost Optimal Conflict Resolution

Conflicts can occur when multiple stakeholders in a configuration process pull the solution in different directions. Debugging tools are therefore needed to mediate the conflict in a cost conscious manner. For example, when a car's software configuration is incompatible with the legacy ECU configuration, it is (probably) cheaper to change the software configuration than to change the ECU configuration and the assembly process of the car. The solver should therefore try to minimize the overall cost of the changes.

We can extend the CSP model to perform cost-based feature selection and deselection optimization. First, we extend the CURE model to associate a cost variable, $b_i \subset B$, with each feature in the feature model. Each cost variable represents how expensive (or conversely how beneficial) it is for the solver to recommend that the state of that feature be changed. Before each invocation of the debugger, the stakeholders provide these cost variables to guide the solver in its recommendations of features to select or deselect.

Next, we construct the superset of the features that the various stakeholders desire, as shown in Figure XI.4. The superset represents the ideal, although incorrect, configuration that the stakeholders would like to have. The goal is to find a way to reach a correct

configuration from this superset of features that involves the lowest total cost for changes. The superset is input to the solver as values for the variables in O.

Finally, we alter our original optimization goal so that the solver will attempt to minimize (or maximize) the cost of the features it suggests selecting or deselecting. We define a global cost variable G and let G capture the sum of the costs of the changes that the solver suggests:

$$G = \sum_{i=1}^{n} (d_i * b_i) + (s_i * b_i)$$

G is thus equal to the sum of the costs of all features that the solver either recommends to select or deselect. Rather than instructing the solver to minimize the sum of $S \cup D$, we ask it to minimize or maximize G.

The result of the labeling is a series of changes needed to reach a valid configuration that optimally integrates the desires and decisions of the various stakeholders. Of course, one particular stakeholder may have to incur more cost than another in the interest of reaching a globally better solution. Further constraints, such as limiting the maximum difference between the cost incurred by any two stakeholders, could also be added. The mediation process can be tuned to provide numerous types of behavior by providing different optimization goals. This CSP diagnostic method enables CURE to address Challenge 2.

Empirical Results

Effective automated diagnostic methods should scale to handle feature models of production systems. This section presents empirical results from experiments we performed to evaluate the scalability of CURE. We compare the scalability of both CURE's optimal and bounding methods from Sections XI and XI.

Experimental Platform

To perform our experiments, we used the implementation of CURE that is provided by the Model Intelligence libraries from the Eclipse Foundation's Generic Eclipse Modeling System (GEMS) project [160]. Internally, the GEMS Model Intelligence implementation of CURE uses the Java Choco Constraint Solver [2] to derive labelings of the diagnostic CSP. The experiments were performed on a computer with an Intel Core DUO 2.4GHZ CPU, 2 gigabytes of memory, Windows XP, and a version 1.6 Java Virtual Machine (JVM). The JVM was run in client mode using a heap size of 40 megabytes (-Xms40m) and a maximum memory size of 256 megabytes (-Xmx256m).

A challenging aspect of the scalability analysis is that CSP-based techniques can vary in solving time based on individual problem characteristics. In theory, CSP's have exponential worst case time complexity, but are often much faster in practice. To evaluate CURE, therefore, it was necessary to apply it to as many models as possible. The key challenge with this approach is that hundreds or thousands of real feature models are not readily available and manually constructing them is impractical.

To provide the large numbers of feature models needed for our experiments, therefore, we built a feature model generator that randomly creates feature models with the desired branching and constraint characteristics. We also imbued the generator with the capability to generate feature selections from a feature model and probabilistically insert a bounded number of errors/conflicts into the configuration. The feature model generator and code for these experiments is available in open-source form from [5].

From preliminary feasibility experiments we conducted, we observed that the branching factor of the tree had little effect on the algorithm's solving time. We also compared diagnosis time using models with 0%, 10%, and 50% cross-tree constraints and saw that the each increment in the percentage of cross-tree constraints improved performance. For example, with the optimal method and 1,000 feature models, the average diagnosis time gradually decreased from 47 seconds with 0% cross-tree constraints to 36 seconds with

50% cross-tree constraints. The key indicator of the solving complexity was the number of XOR- or cardinality-based feature groups in a model. XOR and cardinality-based feature groups are features that require the set of their selected children to satisfy a cardinality constraint (the constraint is 1..1 for XOR).

For our tests, we limited the branching factor to at most five subfeatures per feature. We also set the probability of XOR- or cardinality-based feature groups being generated to 1/3 at each feature with children. We chose 1/3 since most feature models we have encountered contain more required and optional relationships than XOR- and cardinality-based feature groups. The total number of cross-tree constraints was set at 10%. We also eliminated all diagnosis results from void feature models, since void feature models produced faster diagnostic times and would have skewed the results towards smaller solving times.

To generate feature selections with errors, we used a probability of 1/50 that any particular feature would be configured incorrectly. For each model, we bounded the total errors at 5. In our initial experiments, the solving time was not affected by the number of errors in a given feature model. Again, the prevalence of XOR- or cardinality-based feature groups was the key determiner of solving time.

Bounding Method Scalability

First, we tested the scalability of the less computationally complex bounding diagnosis method. The speed of the bounding technique allowed us to test 2,000 feature models at each data point (2,000 different variations of each size feature model) and test the bounding method's scalability for feature models up to 500 features. With models above 500 features, we had to reduce the number of samples at each size to 200 models due to time constraints. Although these samples are small, they demonstrate the general performance of our technique. Moreover, the results of our experiments with feature models up to 500 features were nearly identical with sample sizes between 100 and 2,000 models.

Figure XI.5 shows the time required to diagnose feature models ranging in size from

50 to 500 features using the bounded method. The figure captures the worst and average solving time in the experiments. As seen from the results, our technique could diagnose models with 500 features in an average of ≈300ms.

The upper bound used for this experiment was a maximum of 10% feature selection changes. When the feature bound was too tight for the diagnosis (*i.e.*, more were needed to reach a correct state) the solver quickly declared there was no valid solution. We therefore discarded all instances where the bound was too tight to avoid skewing the results towards shorter solving times.

Figure XI.5 shows the results of testing the solving time of the bounding method on feature models ranging in size from 500 to 5,000 features.

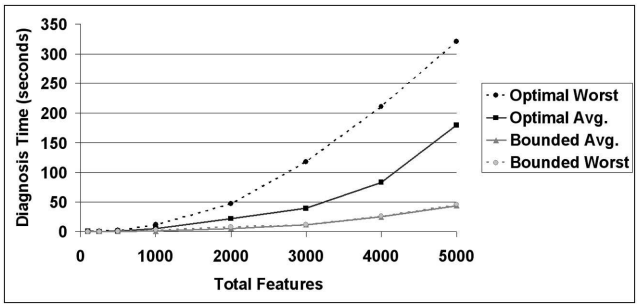


Figure XI.5: Diagnosis Time for Both Methods for Large Feature Models

Models of this size were sufficient to demonstrate scalability for common production systems. The results show that for a 5,000 feature model, the average diagnosis time was ≈ 50 seconds.

Another key variable we tested was how the tightness of the bound on the maximum number of feature changes affected the solving time of the technique. We took a set of 200 feature models and applied varying bounds to see how the bound tightness affected solution time. Figure XI.6 shows that tighter bounds produced faster solution times. These results indicate that tighter bounds allow the solver to discard infeasible solutions more quickly and thus arrive at a solution faster.

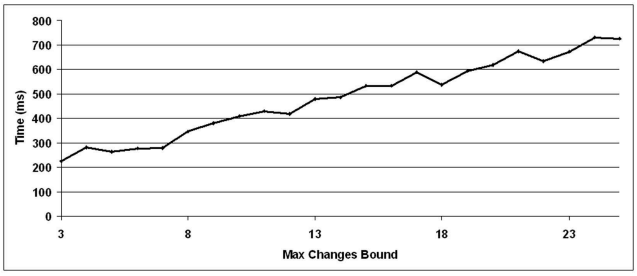


Figure XI.6: 500 Feature Diagnosis Time with Bounding Method and Varying Bounds

Optimal Method Scalability

Next, we tested the scalability of the optimal diagnosis method using 2,000 samples below 500 features and 200 samples for all larger models. Figure XI.5 shows the results from feature models up to 500 features. At 500 features, the optimal method required an average of $\sim$1.5 seconds to produce a diagnosis. Figure XI.5 also shows the tests from larger models ranging in size up to 5,000 features. For a model with 5,000 features, the solver required an average of $\sim$3 minutes per diagnosis.

Comparative Analysis of Optimal and Bounding Methods

Finally, we compared the scalability and quality of results produced with the two methods. Figure XI.5 shows the bounding method performs and scales significantly better than the optimal method. For feature models of up to 1,000 features, however, both techniques take less than 5 seconds and the optimal method is the better choice. This result raises the question of how much of a tradeoff in solution quality for speed is made when the bounding method is used over the optimal method for larger models.

The bound that is chosen determines the quality of the solution that is produced by the solver. The optimality of a diagnosis given by the bounding method is the number of changes suggested by the bounding method, $Bounded(S \cup D)$, divided by the optimal number of changes, $Opt(S \cup D)$, which yields $\frac{Bounded(S \cup D)}{Opt(S \cup D)}$. Since the bounding method uses the constraint $(S \cup D) \leq K$ to ensure that at most K changes are suggested, we can

state the worst case optimality of the bounded method as $\frac{K}{Opt(S \cup D)}$. The closer our bound, K, is to the true optimal number of changes to make, the better the diagnosis will be.

Since tighter bounds produce faster solving times *and* better results, debuggers should start with very small bounds and iteratively increase them upward as needed. One approach is to layer an adaptive algorithm on top of the diagnosis algorithm to move the bound by varying amounts each time the bound proves too tight. Another approach is to employ binary search to home in on the ideal bound. We will investigate both techniques in future work.

Debugging Scenarios

Staged configuration and viewpoint debugging (Challenges 1 & 3) are special cases of the technique where the solver is not allowed to modify the selection state of one or more features (*i.e.*, the viewpoint or the feature at time T'). Both of these special cases of debugging actually reduce the search space by fixing values for one or more of the CSP variables. For example, performing staged configuration debugging, which fixes the value for one CSP variable, on a model with 1,000 features, reduced the optimal method's average solving time by ≈ 2.5 seconds and the bounding method by $\approx .1$ seconds.

Cost-based conflict mediation (Challenge 2) performs identically to the standard diagnosis technique. Cost-based mediation merely introduces a series of coefficients, $b_i \subset B$ into the optimization goal. These coefficients do not increase solving time. Furthermore, initiating the diagnosis method with the superset of the configuration participants' desired feature selections also did not impact performance.

CHAPTER XII

CONCLUSION

There are a number of hard challenges related to the configuration of SPL variants from feature models. Previously developers optimized and constructed software with an emphasis on source code and a restricted set of requirements. In the SPL paradigm, configuration is the main mode of program construction and optimization is done by performing a discriminating selection of components. Source-code focused development is primarily a manual activity whereas configuration can be highly automated.

This dissertation has shown that CSP-based configuration techniques provide a number of promising benefits for SPL construction. CSP configuration techniques can 1) perform optimization, 2) perform automated debugging, 3) perform fast and flexibly enough to serve as a healing mechanism, and 4) can guide manual modeling steps. In the future, as SPLs become increasingly complex, CSP-based configuration techniques will provide an excellent option for reducing the complexity of SPL variant derivation.

The following is a summary of lessons learned from the research work presented in this dissertation:

1. PLA composition and non-functional requirements can be used to efficiently prune the variant selection space and provide good performance. There are many patterns of requirements specification that can be used to optimize a PLA for automated variant selection. In future work, we intend to further explore these patterns.

2. Although Scatter can automate variant selection, it works best when a PLA is crafted with performance in mind. An arbitrary PLA may or may not allow for rapid variant selection. PLA's that will be used in conjunction with an automated variant selector should be carefully constructed to avoid poor performance.

3. A key challenge of automating product variant selection is debugging mistakes in the product-line's specification. A simple mistake, such as a misplaced exclusion constraint between components, can cause variant selection to fail. Moreover, the failure may only appear intermittently for certain device types and be hard to identify during testing. Even once it is discerned that there is a problem, identifying the source of the problem can be extremely challenging (we have experienced this phenomenon).

4. More work must be done to understand how to merge and integrate the various information sources that will provide device characterizations. Device characterizations may come from customer databases, discovery services, and location services. Finding the right transformations to correlate and utilize these diverse information streams is important to provide customized and correct variant selection.

5. Developers normally focus on the functional variability in a product. Looking at other aspects of variability, such as packaging variability, is important too. As we have shown, although a product may have high functional variability, it can be significantly less variable with respect to packaging or memory footprint. These non-functional aspects can be exploited to reduce the complexity of automated variant selection.

6. Fresh alleviates the problems described earlier by executing a series of Java probes at application launch to identify constrained variabilities, formalizing and solving a constraint satisfaction problem of the configuration problem, and dynamically rewriting the application's XML configuration files. The information on functional and non-functional properties collected by automated probing can be treated as a constraint satisfaction problem and a correct application configuration derived by using a constraint solver. Moreover, the constraint solver can produce a solution that is correct with respect to both the feature model and the decisions made by the roles.

7. Probes did not add significant complexity to Fresh's automated configuration approach. An application typically requires a probe for each point of variability. In some cases, a probe may be needed for each individual feature. In other cases, a single probe can identify what features are enabled in an entire feature group. As with unit test frameworks, such as JUnit, probes are relative straightforward to write. Although unit tests can often comprise a substantial amount of code compared to the application itself, this was not the case for probes.

8. Even if a full constraint-solver based solution is not deemed needed, using a configuration probing infrastructure can be useful. Creating probes to ensure that individual points of configuration are properly fixed can help improve the guarantees that an application is installed and configured properly. Since application misconfiguration contributes to a significant portion of application failures [50], developers should consider the use of automated configuration checking.

9. Capturing and allowing the weaver to solve the global application constraints required to produce a weaving solution

10. Informing the weaver of the overall solution goals so that the weaver can derive the best overall weaving solution with respect to a cost function and

11. Encoding using model transformations to automatically generate implementations of the global weaving solution for each required weaving platform.

12. CURE can scale to feature models with several thousand features.

13. The optimality of the diagnosis provided by the bounding method is determined by how close K is set to the true minimum number of features that need to be changed to reach a valid state. Setting an accurate bound for K is not easy. In future work, we plan to investigate different methods of honing the boundary used in the bounding method.

14. The same CSP can often be stated in multiple ways. Different formulations can yield different performance characteristics. In future work, we intend to see if it is possible to vary the diagnosis CSP formulation and show that the technique can scale to even larger models while still providing reasonable runtimes.

The tools and techniques described in this dissertation are open source and available from `http://www.eclipse.org/gmt/gems`.

APPENDIX A

LIST OF PUBLICATIONS

Research on Model Intelligence, Scatter, Refresh, and CURE has led to the following referred journal, conference, and workshop publications as well as book chapters.

Journal Publications

1. Jules White, Douglas C. Schmidt, Egon Wuchner, Andrey Nechypurenko, Automatically Composing Reusable Software Components for Mobile Devices, Journal of the Brazilian Computer Society Special Issue on Software Reuse, SciELO Brasil, Volume 14, Number 1, pgs. 25-44, March, 2008

2. Jules White, Harrison Strowd, Douglas C. Schmidt, Creating Self-healing Service Compositions with Feature Models & Microrebooting, International Journal of Business Process Integration & Management (to appear)

3. Jules White, Douglas Schmidt, Aniruddha Gokhale, Simplifying Autonomic Enterprise Java Bean Applications via Model-driven Engineering & Simulation, Journal of Software & Systems Modeling, Springer, Volume 7, Number 1, pgs. 3-23, May, 2007

Conference Publications

1. Jules White, David Benavides, Douglas C. Schmidt, Antonio Ruiz-Cortez, and Pablo Trindad, Automated Diagnosis of Product-line Configuration Errors in Feature Models, Software Product Lines Conference, September, 2008, Limerick, Ireland

2. Jules White & Douglas C. Schmidt, Automated Configuration of Component-based Distributed Real-time & Embedded Systems from Feature Models, Proceedings of

the 17th Annual Conference of the International Federation of Automatic Control, Seoul, Korea, July 6-11, 2008.

3. Jules White, Douglas C. Schmidt, Egon Wuchner, Andrey Nechypurenko, Optimizing & Automating Product-Line Variant Selection for Mobile Devices, 11th Annual Software Product Line Conference (SPLC), Sept. 10-14, 2007, Kyoto, Japan

4. Jules White & Douglas C. Schmidt, Automated Configuration of Component-based Distributed Real-time & Embedded Systems from Feature Models, Proceedings of the 17th Annual Conference of the International Federation of Automatic Control, Seoul, Korea, July 6-11, 2008.

5. Jules White, Krzysztof Czarnecki, Douglas C. Schmidt, Gunther Lenz, Christoph Wienands, Egon Wuchner, & Ludger Fiege, Automated Model-based Configuration of Enterprise Java Applications, EDOC 2007, October, 2007, Annapolis, Maryland

6. Jules White, Douglas C. Schmidt, Andrey Nechypurenko, Egon Wuchner, Model Intelligence: an Approach to Modeling Guidance, UPGRADE Journal (to appear)

7. Andrey Nechypurenko, Egon Wuchner, Jules White, & Douglas C. Schmidt, Application of Aspect-based Modeling & Weaving for Complexity Reduction in the Development of Automotive Distributed Realtime Embedded Systems, Proceedings of the Sixth International Conference on Aspect-Oriented Software Development, Vancouver, British Columbia, March 12-16, 2007.

8. Jules White & Douglas C. Schmidt, Reducing Enterprise Product Line Architecture Deployment Costs via Model-Driven Deployment & Configuration Testing, Poster paper at the 13th Annual IEEE International Conference & Workshop on the Engineering of Computer Based Systems (ECBS ’06), March 27th-30th, 2006, University of Potsdam, Potsdam, Germany.

9. Jules White, Douglas Schmidt, & Aniruddha Gokhale, Simplifying Autonomic Enterprise Java Bean Applications via Model-driven Development: a Case Study, Proceedings of MODELS 2005, ACM/IEEE 8th International Conference on Model Driven Engineering Languages & Systems, Half Moon Resort, Montego Bay, Jamaica, October 5-7, 2005. (Selected as a best paper)

10. Jules White, Douglas Schmidt, & Aniruddha Gokhale, The J3 Process for Building Autonomic Enterprise Java Bean Systems, Proceedings of the International Conference on Autonomic Computing (ICAC 2005), Seattle, WA, June 2005 (short paper).

Book Chapters

1. Jules White, Andrey Nechypurenko, Egon Wuchner, & Douglas Schmidt, Reducing the Complexity of Designing & Optimizing Large-scale Systems by Integrating Constraint Solvers with Graphical Modeling Tools, Designing Software-Intensive Systems: Methods & Principles, edited by Dr. Pierre F. Tiako, Langston University, Oklahoma, USA, (to appear)

Workshop Publications

1. Jules White, Douglas C. Schmidt, Sean Mulligan, The Generic Eclipse Modeling System, Model-Driven Development Tool Implementer's Forum, TOOLS '07, June, 2007, Zurich Switzerland

2. Andrey Nechypurenko, Jules White, Egon Wuchner, & Douglas C. Schmidt, Applying Model Intelligence Frameworks for Deployment Problem in Real-time & Embedded Systems, Proceedings of MARTES: Modeling & Analysis of Real-Time & Embedded Systems to be held on October 2, 2006 in Genova, Italy in conjunction with the 9th International Conference on Model Driven Engineering Languages & Systems, MoDELS/UML 2006.

3. Jules White, Andrey Nechypurenko, Egon Wuchner, & Douglas C. Schmidt, Intelligence Frameworks for Assisting Modelers in Combinatorically Challenging Domains, Proceedings of the Workshop on Generative Programming & Component Engineering for QoS Provisioning in Distributed Systems, October 23, 2006, Portland, Oregon.

4. Jules White & Douglas Schmidt, Simplifying the Development of Product-line Customization Tools via Model Driven Development, MODELS 2005 workshop on MDD for Software Product-lines: Fact or Fiction?, October 2, 2005, Jamaica.

Submitted Papers

1. Jules White, Jeff Gray, Douglas C. Schmidt, Constraint-based Model Weaving, IEEE Transactions on Aspect-Oriented Programming

2. Jules White, Douglas C. Schmidt, Automating Deployment Planning with an Aspect Weaver, IET Software Special Issue on Domain-specific Modeling Languages for Aspect-Oriented Programming

BIBLIOGRAPHY

[1] AspectJ, http://www.eclipse.org/aspectj/.

[2] Choco Constraint Programming System. http://choco.sourceforge.net/.

[3] Esper FAQ, http://esper.codehaus.org/tutorials/faq_esper/faq.html#performance.

[4] Event Stream Intelligence with Esper and NEsper. http://esper.codehaus.org.

[5] Experimental Platform, http://www.dre.vanderbilt.edu/jules/splc08.zip.

[6] GMF, Eclipse Graphical Modeling Framework. http://www.eclipse.org/gmf.

[7] HyperJ, http://www.alphaworks.ibm.com/tech/hyperj.

[8] .NET Pet Store, http://msdn2.microsoft.com/en-us/library/ms978487.aspx.

[9] The Java Pet Store. http://java.sun.com/developer/releases/petstore/.

[10] The Spring Framework, http://www.springframework.org/about.

[11] M.M. Akbar, E.G. Manning, G.C. Shoja, and S. Khan. Heuristic Solutions for the Multiple-Choice Multi-Dimension Knapsack Problem. pages 659–668. Springer, May 2001.

[12] M.M. Akbar, E.G. Manning, G.C. Shoja, and S. Khan. Heuristic Solutions for the Multiple-Choice Multi-Dimension Knapsack Problem. *International Conference on Computational Science*, pages 659–668, May 2001.

[13] E. Alba and J. Francisco Chicano. Software project management with GAs. *Information Sciences*, 177(11):2380–2401, 2007.

[14] M. Anastasopoulos. Software Product Lines for Pervasive Computing. *IESE-Report No. 044.04/E version*, 1.

[15] E. Baniassad and S. Clarke. Theme: an Approach for Aspect-oriented Analysis and Design. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 158–167, Scotland, UK, May 2004.

[16] A. Barreto, M.O. Barros, and C.M.L. Werner. Staffing a software project: A constraint satisfaction and optimization-based approach. *Computers and Operations Research*, 35(10):3073–3089, 2008.

[17] D. Batory. Feature Models, Grammars, and Prepositional Formulas. *Software Product Lines: 9th International Conference, SPLC 2005, Rennes, France, September*

26-29, 2005: Proceedings, 2005.

[18] D. Batory, D. Benavides, and A. Ruiz-Cortés. Automated Analysis of Feature Models: Challenges Ahead. *Communications of the ACM*, December, 2006.

[19] BEA Systems, et al. *CORBA Component Model Joint Revised Submission*. Object Management Group, OMG Document orbos/99-07-01 edition, July 1999.

[20] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. Using Java CSP solvers in the automated analyses of feature models. *Post-Proceedings of The Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*.

[21] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In *Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, 2007.

[22] D. Benavides, P. Trinidad, and A. Ruiz-Cortes. Automated Reasoning on Feature Models. *17th Conference on Advanced Information Systems Engineering (CAiSEŠ05, Proceedings), LNCS*, 3520:491–503, 2005.

[23] D. Beuche. Variant management with pure:: variants. Technical report, Pure-systems GmbH, http://www. pure-systems. com, 2003.

[24] J. Bézivin. From Object Composition to Model Transformation with the MDA. In *Proceedings of TOOLS*, pages 350–354, Santa Barbara, CA, USA, August 2001.

[25] J. Bézivin, F. Jouault, and P. Valduriez. First Experiments with a ModelWeaver. In *Proceedings of OOPSLA & GPCE Workshop*, Vancouver, Canada, March 2004.

[26] S. Bistarelli, H. Fargier, U. Montanari, F. Rossi, T. Schiex, and G. Verfaillie. Semiring-based CSPs and Valued CSPs: Basic Properties and Comparison. *Over-Constrained Systems*, 1106:111–150, 1996.

[27] S. Bistarelli, Ugo Montanari, and F. Rossi. Semiring-Based Constraint Satisfaction and Optimization. *Journal of the ACM*, 44(2):201–236, 1997.

[28] J. Brittain and I.F. Darwin. *Tomcat: The Definitive Guide*. O'Reilly Media, 2003.

[29] F. Budinsky. *Eclipse Modeling Framework*. Addison-Wesley Professional, New York, NY, USA, 2003.

[30] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. Addison-Wesley, Reading, MA, 2003.

[31] R. Buhrdorf, D. Churchett, and C.W. Krueger. SalionŠs Experience with a Reactive Software Product Line Approach. *Proceeding of the 5th International Workshop on*

Product Family Engineering. Nov, 2003.

[32] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot-a technique for cheap recovery. *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 31–44, 2004.

[33] Yves Caseau, Francois-Xavier Josset, and Francois Laburthe. CLAIRE: Combining Sets, Search And Rules To Better Express Algorithms. *Theory and Practice of Logic Programming*, 2:2002, 2004. Available from: `http://www.citebase.org/abstract?id=oai:arXiv.org:cs/0405091`.

[34] PL Chiu, Y. Chen, and KH Lee. A request scheduling algorithm to support flexible resource reservations in advance. *Electrical and Computer Engineering, 2004. Canadian Conference on*, 4, 2004.

[35] A. Chun. Constraint programming in Java with JSolver. *Proc. Practical Applications of Constraint Logic Programming, PACLP99*, 1999.

[36] L. Chung. *Non-Functional Requirements in Software Engineering*. Springer, 2000.

[37] J.A. Clark and J.L. Jacob. Protocols are programs too: the meta-heuristic search for security protocols. *Information and Software Technology*, 43(14):891–904, 2001.

[38] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, 2002.

[39] EG Coffman Jr, G. Galambos, S. Martello, and D. Vigo. Bin packing approximation algorithms: combinatorial analysis. *Handbook of Combinatorial Optimization. Kluwer Academic Publishers*, 1998.

[40] Jacques Cohen. Constraint logic programming languages. *Communications of the ACM*, 33(7):52–68, 1990.

[41] J. Coplien, D. Hoffman, and D. Weiss. Commonality and Variability in Software Engineering. *IEEE Software*, 15:37–45, Nov.-Dec. 1998.

[42] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT, 1990.

[43] T. Cottenier, A. van den Berg, and T. Elrad. The Motorola WEAVR: Model Weaving in a Large Industrial Context. In *Proceedings of the International Conference on Aspect-Oriented Software Development, Industry Track*, Vancouver, Canada, March 2006.

[44] I. Crnkovic. Component-based software engineering-new challenges in software development. *Information Technology Interfaces, 2003. ITI 2003. Proceedings of the 25th International Conference on*, pages 9–18, 2003.

[45] S. Curtis. The Magnetospheric Multiscale Mission...Resolving Fundamental Processes in Space Plasmas. *NASA STI/Recon Technical Report N*, pages 48257–+, December 1999.

[46] K. Czarnecki, M. Antkiewicz, C.H.P. Kim, S. Lau, and K. Pietroszek. In *FMP and FMP2RSM: Eclipse Plug-ins for Modeling Features Using Model Templates*, pages 200–201. ACM Press New York, NY, USA, October 2005.

[47] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. Anaheim, CA, USA, October 2003.

[48] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration Using Feature Models. *Software Product Lines: Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004: Proceedings*, 2004.

[49] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration through specialization and multi-level configuration of feature models. *Software Process Improvement and Practice*, 10(2):143–169, 2005.

[50] D. Patterson D. Oppenheimer, A. Ganapathi. Why do Internet Services Fail, and What can be Done about It? *Proceedings of the USENIX Symposium on Internet Techn̆nologies and Systems*, March 2003.

[51] MD Del Fabro, J Bézivin, and P Valduriez. Weaving Models with the Eclipse AMW plugin. In *Eclipse Modeling Symposium, Eclipse Summit Europe*, Esslingen, Germany, October 2006.

[52] G. Edwards, G. Deng, D. Schmidt, A. Gokhale, and B. Natarajan. Model-Driven Configuration and Deployment of Component Middleware Publish/Subscribe Services. *Generative Programming and Component Engineering (GPCE)*, pages 337–360, 2004.

[53] T. Elrad, O. Aldawud, and A. Bader. Aspect-Oriented Modeling: Bridging the Gap between Implementation and Design. In *Generative Programming and Component Engineering (GPCE)*, pages 189–201, Pittsburgh, PA, USA, October 2005.

[54] R. Fletcher. *Practical methods of optimization*. Wiley-Interscience New York, NY, USA, 1987.

[55] M. Fleury and F. Reverbel. The JBoss Extensible Server. *International Middleware Conference*, 2003.

[56] A. Foundation. Apache JMeter, http://jmeter.apache.org.

[57] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, November 2002. Available from: `http://www.amazon.fr/exec/obidos/ASIN/`

0534388094/citeulike04-21.

[58] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional, 2004.

[59] R. France, I. Ray, G. Georg, and S. Ghosh. An Aspect-Oriented Approach to Early Design Modeling. *IEE Proceedings-Software*, 151(4):173–185, 2004.

[60] D. Frankel. *Model driven architecture*. Wiley New York, 2003.

[61] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[62] P. Goldsack, J. Guijarro, A. Lain, G. Mecheneau, P. Murray, and P. Toft. SmartFrog: Configuration and Automatic Ignition of Distributed Applications. *HP Openview University Association conference*, 2003.

[63] H. Gomaa. *Designing concurrent, distributed, and real-time applications with UML*. Addison-Wesley Reading, MA, USA, 2000.

[64] J. Gosling. *Introductory Statistics*. Pascal Press, 1995.

[65] J. Gray, T. Bapty, S. Neema, and J. Tuck. Handling crosscutting constraints in domain-specific modeling. *Communications of the ACM*, 44(10):87–93, 2001.

[66] J. Hannemann, G.C. Murphy, and G. Kiczales. Role-based Refactoring of Crosscutting Concerns. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, pages 135–146, Chicago, Illinois, USA, March 2005.

[67] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987. Available from: citeseer.ist.psu.edu/article/harel87statecharts.html.

[68] M. Harman. The Current State and Future of Search Based Software Engineering. *International Conference on Software Engineering*, pages 342–357, 2007.

[69] M. Harman and B.F. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001.

[70] James Hill, Jules White, Sean Eade, and Douglas C. Schmidt. Towards a Solution for Synchronizing Disparate Models of Ultra-Large-Scale Systems. In *Proceedings of the ULSSIS Workshop*, Leipzig, Germany, May 2008.

[71] E. Hilsdale and J. Hugunin. Advice Weaving in AspectJ. pages 26–35, Lancaster, UK, March 2004.

[72] S. Holzner. *Ant: The Definitive Guide*. O'Reilly, Sebastopol, CA, USA, 2005.

[73] P.J. Huber, J. Wiley, and W. InterScience. *Robust statistics*. Wiley New York, 1981.

[74] T. Ibaraki, T. Hasegawa, K. Teranaka, and J. Iwase. The Multiple Choice Knapsack Problem. *J. Oper. Res. Soc. Japan*, 21:59–94, 1978.

[75] O.H. Ibarra and C.E. Kim. Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems. *Journal of the ACM (JACM)*, 22(4):463–468, 1975.

[76] M. Islam, M. Akbar, H. Hossain, and EG Manning. Admission control of multimedia sessions to a set of multimedia servers connected by an enterprise network. *Communications, Computers and signal Processing, 2005. PACRIM. 2005 IEEE Pacific Rim Conference on*, pages 157–160, 2005.

[77] J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *constraints*, 2(2):0, 1994.

[78] M. Jampel, E.C. Freuder, and M.W. Maher. *Over-Constrained Systems*. Springer-Verlag London, UK, 1996.

[79] R. Johnson and J. Hoeller. *Expert one-on-one J2EE development without EJB*. Wrox, 2004.

[80] N. Jussien and P. Boizumault. Implementing Constraint Relaxation over Finite Domains Using Assumption-Based Truth Maintenance Systems. *Lecture Notes In Computer Science*, 1106:265–280, 1996.

[81] K.C. Kang et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Carnegie Mellon University, Software Engineering Institute, 1990.

[82] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A Feature-Oriented Reuse Method with Domain-specific Reference Architectures. *Annals of Software Engineering*, 5(0):143–168, January 1998.

[83] D. König, A. Glover, P. King, G. Laforge, and J. Skeet. *Groovy in Action*. Manning Publications, 2007.

[84] V. Kumar. Algorithms for Constraint-Satisfaction Problems: A Survey. *AI Magazine*, 13(1):32–44, 1992.

[85] I Kurtev, K van den Berg, and F Jouault. Rule-based Modularization in Model Transformation Languages Illustrated with ATL. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC 06)*, pages 1202–1209, Dijon, France, April 2006.

[86] A.E. Lawabni and A.H. Tewfik. Resource Management and Quality Adaptation in

Distributed Multimedia Networks. *Proceedings of the 10th IEEE Symposium on Computers and Communications (ISCC'05)-Volume 00*, pages 604–610, 2005.

[87] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *Computer*, 34(11):44–51, 2001.

[88] T. Lemlouma and N. Layaida. Context-aware Adaptation for Mobile Devices. *Mobile Data Management, 2004. Proceedings. 2004 IEEE International Conference on*, pages 106–111, 2004.

[89] G. Lenz and C. Wienands. *Practical Software Factories in .NET*. Apress, Berkeley, CA, 2006.

[90] W.S. Li, W.P. Hsiung, DV Kalshnikov, R. Sion, O. Po, D. Agrawal, and K.S. Candan. Issues and Evaluations of Caching Solutions for Web Application Acceleration. In *Proceedings of the 28th International Conference on Very Large Data Bases*, Hong Kong, China, August 2002.

[91] D.C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2001.

[92] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B.G. Lindsay, and J.F. Naughton. Middle-tier Database Caching for E-business. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 600–611, Madison, Wisconsin, June 2002.

[93] M. Mannion. Using First-order Logic for Product Line Model Validation. *Proceedings of the Second International Conference on Software Product Lines*, 2379:176–187, 2002.

[94] M. Mannion and J. Camara. Theorem Proving for Product Line Model Verification. *Fifth International Workshop on Product Family Engineering, PFE-5, Siena*, pages 4–6, 2003.

[95] T. Männistö, T. Soininen, and R. Sulonen. Product Configuration View to Software Product Families. *10th International Workshop on Software Configuration Management (SCM-10), Toronto, Canada*, pages 14–15, 2001.

[96] V. Matena, S. Krishnan, B. Stearns, and L. Demichiel. *Applying Enterprise Javabeans: Component-based Development for the J2EE Platform*. Addison-Wesley, 2003.

[97] P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification & Reliability*, 14(2):105–156, 2004.

[98] Andreas Metzger, Klaus Pohl, Patrick Heymans, Pierre-Yves Schobbens, and Germain Saval. Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis. In *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, pages 243–253, 2007. Available from: `http://ieeexplore.ieee.org/xpls/abs\_all.jsp?arnumber=4384187`.

[99] Laurent Michel and Pascal Van Hentenryck. Comet in context. In *PCK50: Proceedings of the Paris C. Kanellakis Memorial Workshop on Principles of Computing & Knowledge*, pages 95–107, San Diego, CA, USA, 2003.

[100] S. Microsystems. Java Pet Store Sample Application.

[101] S. Mittal and F. Frayman. Towards a generic model of configuration tasks. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 2:1395–1401, 1989.

[102] C. Mohan. Caching Technologies for Web Applications. In *Proceedings of the 27th International Conference on Very Large Data Bases*, page 726, Rome, Italy, September 2001.

[103] M. MOSER, D.P. JOKANOVIC, and N. SHIRATORI. An Algorithm for the Multidimensional Multiple-Choice Knapsack Problem. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 80(3):582–589, 1997.

[104] M. Mostofa Akbar, M. Sohel Rahman, M. Kaykobad, EG Manning, and GC Shoja. Solving the Multidimensional Multiple-choice Knapsack Problem by constructing convex hulls. *Computers and Operations Research*, 33(5):1259–1273, 2006.

[105] D. Muthig, I. John, M. Anastasopoulos, T. Forster, J. Dörr, and K. Schmid. GoPhone-A Software Product Line in the Mobile Phone Domain. *IESE-Report No*, 25, 2004.

[106] Andrey Nechypurenko, Jules White, Egon Wuchner, and Douglas C. Schmidt. Applying Model Intelligence Frameworks to Deployment Problems in Real-time and Embedded Systems. In *Proceedings of MARTES: Modeling and Analysis of Real-Time and Embedded Systems at the 9th International Conference on Model Driven Engineering Languages and Systems, MoDELS/UML 2006*, Genova, Italy, 2006.

[107] Andrey Nechypurenko, Egon Wuchner, Jules White, and Douglas C. Schmidt. Application of Aspect-based Modeling and Weaving for Complexity Reduction in the Development of Automotive Distributed Realtime Embedded Systems. In *Proceedings of the Sixth International Conference on AspectOriented Software Development*, Vancouver, Canada, March 2007.

[108] Andrey Nechypurenko, Egon Wuchner, Jules White, and Douglas C. Schmidt. Application of Aspect-based Modeling and Weaving for Complexity Reduction in the Development of Automotive Distributed Realtime Embedded Systems. In *Proceedings of the Sixth International Conference on AspectOriented Software Development*, Vancouver, Canada, March 2007.

[109] J.A. Nelder and R. Mead. A Simplex Method for Function Minimization. *Computer Journal*, 7(4):308–313, 1965.

[110] L. Northrop, P.H. Feiler, B. Pollak, and D. Pipitone. *Ultra-large-scale Systems: The Software Challenge of the Future*. Software Engineering Institute, Carnegie Mellon University, 2006.

[111] J. Ousterhout. Why threads are a bad idea (for most purposes). *Presentation given at the 1996 Usenix Annual Technical Conference, January*, 1996.

[112] J.R. Parker and JR Parker. *Algorithms for Image Processing and Computer Vision*. John Wiley & Sons, Inc. New York, NY, USA, 1996.

[113] P.J. Phillips, H. Moon, S.A. Rizvi, and P.J. Rauss. The FERET Evaluation Methodology for Face-Recognition Algorithms. 2000.

[114] D. Pisinger. A minimal algorithm for the multiple-choice knapsack problem. *European Journal of Operational Research*, 83(2):394–410, 1995.

[115] T. Quatrani. *Visual modeling with Rational Rose and UML*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1998.

[116] R. Rabiser, P. Grunbacher, and D. Dhungana. Supporting Product Derivation by Adapting and Augmenting Variability Models. *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 141–150, 2007.

[117] YR Reddy, S. Ghosh, R.B. France, G. Straw, J.M. Bieman, N. McEachen, E. Song, and G. Georg. Directives for Composing Aspect-Oriented Design Class Models. *Transactions on Aspect-Oriented Software Development*, 3880:75–105, 2006.

[118] C.R. Reeves. *Modern Heuristic Techniques for Combinatorial Problems*. John Wiley & Sons, Inc. New York, NY, USA, 1993.

[119] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[120] D. Sabin and E.C. Freuder. Configuration as composite constraint satisfaction. *Proceedings of the Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161, 1996.

[121] D. Sabin and R. Weigel. Product configuration frameworks-a survey. *Intelligent Systems and Their Applications, IEEE [see also IEEE Intelligent Systems]*, 13(4):42–49, 1998.

[122] T. Schiex. Possibilistic Constraint Satisfaction Problems or How to Handle Soft Constraints. pages 268–275, San Mateo, CA, USA, 1992. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA.

[123] D.C. Schmidt. Middleware for real-time and embedded systems. *Communications of the ACM*, 45(6):43–48, 2002.

[124] C. Schulte and P.J. Stuckey. Efficient constraint propagation engines. *Arxiv preprint cs.AI/0611009*, 2006.

[125] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.

[126] David C. Sharp and Wendy C. Roll. Model-Based Integration of Reusable Component-Based Avionics System. Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS 2003, May 2003.

[127] S. Shavor, J. D'Anjou, P. McCarthy, J. Kellerman, and S. Fairbrother. *The Java Developer's Guide to Eclipse*. Pearson Education, Upper Saddle River, NJ, USA, 2003.

[128] P. Sinha and A.A. Zoltners. The multiple-choice knapsack problem. *Operations Research*, 27(3):503–515, 1979.

[129] Gert Smolka. The Oz Programming Model. In *JELIA '96: Proceedings of the European Workshop on Logics in Artificial Intelligence*, page 251, London, UK, 1996. Springer-Verlag.

[130] Software Composition and Modeling (Softcom) Laboratory. Constraint-Specification Aspect Weaver (C-SAW). www.cis.uab.edu/gray/Research/C-SAW, University of Alabama at Birmingham, Birmingham, AL.

[131] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro. Automated Error Analysis for the Agilization of Feature Modeling. *Journal of Systems and Software, in press*, 2007.

[132] T.B. Valesky. *Enterprise JavaBeans*. Addison-Wesley Reading, MA, USA, 1999.

[133] T. van der Storm. *Variability and Component Composition*. Springer, 2004.

[134] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press Cambridge, MA, USA, 1989.

[135] P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. *Constraint-Based Reasoning*, 1994.

[136] M. Voelter, I. Groher, I. Consultant, and G. Heidenheim. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. pages 233–242, Kyoto, Japan, September 2007.

[137] R.J. Wallace and E.C. Freuder. Heuristic Methods for Over-constrained Constraint Satisfaction Problems. *Over-Constrained Systems*, 1106:207–216, 1996.

[138] Nanbor Wang, Douglas C. Schmidt, and David Levine. Optimizing the CORBA Component Model for High-performance and Real-time Applications. In *'Work-in-Progress' session at the Middleware 2000 Conference*. ACM/IFIP, April 2000.

[139] Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan. An Overview of the CORBA Component Model. In George Heineman and Bill Councill, editors, *Component-Based Software Engineering*. Addison-Wesley, Reading, Massachusetts, 2000.

[140] J.B. Warmer and A.G. Kleppe. *The Object Constraint Language*. Addison-Wesley, Reading, MA, USA, 2003.

[141] M. Weber and J. Weisbrod. Requirements engineering in automotive development-experiences and challenges. *Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on*, pages 331–340, 2002.

[142] J. White, B. Dougherty, and D.C. Schmidt. Filtered Cartesian Flattening. *Workshop on Analysis of Software Product-Lines at the International Conference on Software Product-lines*, October 2008.

[143] Jules White. Healing Pet Store Case Study Implementation. http://www.dre.vanderbilt.edu/ jules/petstore-casestudy-code.zip, 2007.

[144] Jules White, Krzysztof Czarnecki, Douglas C. Schmidt, Gunther Lenz, Christoph Wienands, Egon Wuchner, and Ludger Fiege. Automated Model-based Configuration of Enterprise Java Applications. In *EDOC 2007*, Annapolis, Maryland USA, 2007 (to appear).

[145] Jules White, Brian Dougherty, and Douglas C. Schmidt. Filtered Cartesian Flattening: An Approach for Optimally Selecting Features Subject to Resource Constraints. In *Proceedings of the 1st International Workshop on the Analysis of Software Product Lines*, Limerick, Ireland, August 2008.

[146] Jules White, Jeff Gray, and Douglas C. Schmidt. Constraint-based Model Weaving. *Springer Transactions on Aspect-Oriented Software Development, Special Issue on Aspects and Model Driven Engineering*, 2008.

[147] Jules White, Boris Kolpackov, Balachandran Natarajan, and Douglas C. Schmidt. Reducing Code Complexity with Vocabulary-Specific XML Language Bindings. In *Proceedings of the 43rd Annual Southeast Conference*, Atlanta, GA, March 2005. ACM.

[148] Jules White, Andrey Nechypurenko, Egon Wuchner, and Douglas C. Schmidt. Intelligence Frameworks for Assisting Modelers in Combinatorically Challenging Domains. In *Proceedings of the Workshop on Generative Programming and Component Engineering for QoS Provisioning in Distributed Systems at the Fifth International Conference on Generative Programming and Component Engineering (GPCE 2006)*, Genova, Italy, 2006.

[149] Jules White, Andrey Nechypurenko, Egon Wuchner, and Douglas C. Schmidt. Optimizing and Automating Product-Line Variant Selection for Mobile Devices. In *11th International Software Product Line Conference*, Kyoto, Japan, September 2007.

[150] Jules White, Andrey Nechypurenko, Egon Wuchner, and Douglas C. Schmidt. Reducing the Complexity of Optimizing Large-scale Systems by Integrating Constraint Solvers with Graphical Modeling Tools. In Pierre F. Tiako, editor, *Designing Software-Intensive Systems: Methods and Principles*. Idea Group, 2007.

[151] Jules White, Douglas Schmidt, and Aniruddha Gokhale. The J3 Process for Building Autonomic Enterprise Java Bean Systems. *icac*, 00:363–364, 2005.

[152] Jules White, Douglas Schmidt, and Aniruddha Gokhale. Simplifying Autonomic Enterprise Java Bean Applications via Model-driven Development: a Case Study. In *MODELS 2006: 8th International Conference on Model Driven Engineering Languages and Systems*, Montego Bay, Jamaica, October 2005. IEEE/ACM, ACM Press.

[153] Jules White and Douglas C. Schmidt. Simplifying the Development of Product-Line Customization Tools via MDD. In *Workshop: MDD for Software Product Lines, ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, Montego Bay, Jamaica, October 2005.

[154] Jules White and Douglas C. Schmidt. Reducing Enterprise Product Line Architecture Deployment Costs via Model-Driven Deployment and Configuration Testing. In *13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, University of Potsdam, Potsdam, Germany, 2006.

[155] Jules White and Douglas C. Schmidt. Automated Configuration of Component-based Distributed Real-time and Embedded Systems from Feature Models. In *Proceedings of the 17th Annual Conference of the International Federation of Automatic Control*, Seoul, Korea, July 6-11 2008.

[156] Jules White and Douglas C. Schmidt. Model-Driven Product-Line Architectures for

Mobile Devices. In *Proceedings of the 17th Annual Conference of the International Federation of Automatic Control*, Seoul, Korea, July 6-11 2008.

[157] Jules White, Douglas C. Schmidt, David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortez. Automated Diagnosis of Product-line Configuration Errors in Feature Models. In *submitted to the Software Product Lines Conference (SPLC)*, Limerick, Ireland, September 2008.

[158] Jules White, Douglas C. Schmidt, David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortez. Automated Diagnosis of Product-line Configuration Errors in Feature Models. pages 659–668, September 2008.

[159] Jules White, Douglas C. Schmidt, and Aniruddha Gokhale. Simplifying Autonomic Enterprise Java Bean Applications via Model-driven Engineering and Simulation. *Journal of Software and System Modeling*, 7(1):3–23, 2008.

[160] Jules White, Douglas C. Schmidt, and Sean Mulligan. The Generic Eclipse Modeling System. In *Model-Driven Development Tool Implementors Forum at TOOLS 2007*, Zurich, Switzerland, June 2007.

[161] Jules White, Douglas C. Schmidt, Egon Wuchner, and Andrey Nechypurenko. Automatically Composing Reusable Software Components for Mobile Devices. *Journal of the Brazilian Computer Society (JBCS), Special Issue in Software Reuse: Methods, Processes, Tools and Experiences*, 2008.

[162] Jules White, Douglas C. Schmidt, Egon Wuchner, and Andrey Nechypurenko. Model Intelligence: an Approach to Modeling Guidance. *Novática*, 8(192), March 2008.

[163] Jules White, Harrison Strowd, and Douglas C. Schmidt. Creating Self-healing Service Compositions with Feature Modeling and Microrebooting. *The International Journal of Business Process Integration and Management (IJBPIM), Special issue on Model-Driven Service-Oriented Architectures*, 2008.

[164] J. Zhang, T. Cottenier, A. van den Berg, and J. Gray. Aspect Composition in the Motorola Aspect-Oriented Modeling Weaver. *Journal of Object Technology*, 6(7).

[165] W. Zhang, S. Jarzabek, N. Loughran, and A. Rashid. Reengineering a PC-based system into the mobile device product line. *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*, pages 149–160, 2003.