Reducing Enterprise Product Line Architecture Deployment and Testing Costs via Model-Driven Deployment, Configuration, and Testing

Jules White, Brian Dougherty, and Douglas C. Schmidt

Vanderbilt University, Department of Electrical Engineering and Computer Science, Nashville, TN, 37212, USA
{jules, briand, schmidt}@dre.vanderbilt.edu

Abstract. *Product-line architectures (PLAs) are a paradigm for developing software families by customizing and composing reusable artifacts, rather than handcrafting software from scratch. Extensive testing is required to develop reliable PLAs, which may have scores of valid variants that can be constructed from the architecture's components. It is crucial that each variant be tested thoroughly to assure the quality of these applications on multiple platforms and hardware configurations. It is tedious and error-prone, however, to setup numerous distributed test environments manually and ensure they are deployed and configured correctly. To simplify and automate this process, we present a model-driven architecture (MDA) technique that can be used to (1) model a PLA's configuration space, (2) automatically derive configurations to test, and (3) automate the packaging, deployment, and testing of configurations. To validate this MDA process, we use a distributed constraint optimization system case study to quantify the cost savings of using an MDA approach for the deployment and testing of PLAs.*

1 Introduction

Emerging trends and challenges. *Product-line architectures (PLAs)* 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 [1]. The design of a PLA is typically guided by *scope, commonality, and variability* (SCV) analysis [2]. SCV captures key characteristics of software product-lines, including their (1) *scope*, which defines the domains and context of the PLA, (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.

Although PLAs simplify the development of new applications by reusing existing software components, they require significant testing to ensure that valid variants function properly. Not all variants that obey the compositional rules of PLA function properly, which motivates the need for powerful testing methods and tools. For example, connecting two components with compatible interfaces can produce a non-functional variant due to assumptions made by one component, such as boundary conditions, that do not hold for the component to which it is connected [3].

The numerous points of variability in PLAs also yield variant configuration spaces with hundreds, thousands, or more possible variants. It is therefore crucial that PLAs undergo intelligent testing of the variant configuration space to reduce the number of configurations that must be tested. A key challenge in performing intelligent testing of the solution space is determining which variants will yield the most valuable testing results, such as performance data.

Solution approach → Model-driven testing and domain analysis of product-line architectures. Model-driven Architectures (MDA) are a development paradigm that employs models of critical system functionality, model analysis, and code generation to reduce the cost of implementing complex systems. MDA models capture design information, such as software component response-time, that are not present in third-generation programming languages, such as Java and C++. Capturing these critical design properties in a structured model allows developers to perform analyses, such as queuing analyses of a product-line architecture, to catch design flaws early in the development cycle when they are less costly to correct.

A further benefit of MDA is that code generators and model interpreters can be used to traverse the model and automatically generate portions of the implementation or automate repetitive tasks. For example, Unified Modeling Language (UML) models of a system can be transformed via code generation into class skeletons or marshalling code to persist objects as XML. Model interpreters can be used to automatically execute tests of code using frameworks, such as Another Neat Tool (ANT) and JUnit.

MDA offers a potential solution to the challenges faced in testing large-scale PLAs. MDA can be used to model the complex configuration rules of a PLA, analyze the models to determine effective test strategies, and then automate test orchestration. Effectively leveraging MDA to improve test planning and execution, however, requires determining precisely what PLA design properties to model, how to analyze the models, and how best to leverage the results of these analyses.

This chapter focuses on techniques and tools for modeling, analyzing, and testing PLAs. First, we introduce the reader to *feature modeling* [19,20,24], which is a widely used modeling methodology for capturing PLA variability information. Second, we describe approaches for annotating feature models with probabilistic data obtained from application testing that help predict potentially flawed configurations. Next, we present numerical domain analysis techniques that can be used to help guide the production of PLA test plans. Finally, we present the structure and functionality of a FireAnt, which is an open-source Eclipse plug-in for modeling PLAs, performing PLA domain analysis to derive test plans, and automating and orchestrating PLA testing for Java applications

Paper organization. The remainder of this paper is organized as follows: *Section 2* provides a motivating case study used throughout the chapter*; Section 3* presents an overview of PLA modeling and testing challenges; *Section 4* presents an MDA approach to PLA deployment and testing; *Section 5* evaluates the MDA PLA testing approach in the context of the case study; and *Section 6* presents concluding remarks.

2 Motivating Case Study Example

To explore the characteristics of testing PLAs, we have developed an Enterprise Java Beans (EJB)-based *Constraints Optimization System (CONST)* that schedules pickup requests to vehicles. As shown in Figure 1, CONST manages a list of items that must be scheduled for pickup, a list of times that the items must arrive by, and a list of vehicles and drivers that are available to perform the pickup. CONST uses a constraint-optimization engine to find a cost effective assignment of drivers and trucks to pickups.

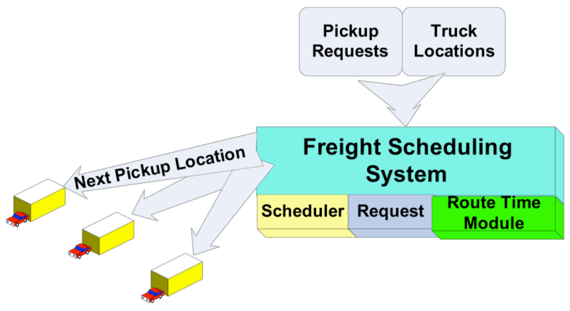

Figure 1. Highway Freight Shipment Scheduling Architecture

CONST’s optimization engine can be used to schedule a wide variety of shipment types. In one configuration, for example, the system could schedule limousines to customers requiring a ride, whereas in another configuration the system could dispatch trucks to highway freight shipments. CONST’s optimization engine must therefore be customizable at design-time to handle these various domains effectively.

CONST must also be customizable at run-time to adapt to changing operating conditions. During peak traffic times, for instance, its optimization engine may need to use traffic-aware routing algorithms, whereas during off-peak times, it may switch to faster traffic-unaware algorithms. Depending on the target domain, CONST also needs to handle failures differently. For example, when scheduling limousines to pickups a degradation of the time required to schedule a reservation below a threshold may require CONST's constraint engine to adapt to improve performance. When scheduling highway freight shipments, however, the threshold may be higher since pickup and drop-off windows are more flexible.

To support the degree of customization described above, we developed CONST as a PLA using SCV analysis, as follows:

- The **scope** is the constraint optimization system architecture and the associated components that address the domain of scheduling shipments to vehicles, e.g., computing route times between vehicles and shipments, maintaining a list of waiting shipments, and calculating the cost of assigning a vehicle to a shipment.
- The **commonality** is the set of components and their interactions that are present in all configurations of CONST, which include the scheduler updating the schedule, the route time module answering requests from the schedule, and the dispatcher sending routing orders to vehicles.
- The **variability** includes how the list of waiting shipments is prioritized, how the system calculates the cost of assigning each vehicle/driver combination to pickups, how late pickups and dropoffs are handled, and how the system handles response time degradation.

By applying the SCV analysis to CONST we designed a PLA that enables the customization of its optimization engine for various domains.

CONST variants are composed of two main assemblies of components: the *PickupList* and the *Optimizer*. The *PickupList* may be implemented as either (1) a *prioritized list* for domains, such as freight shipments, where some cargos have higher priorities, or (2) a *FIFO* list for other domains, such as taxi scheduling. The *Optimizer* is composed of a *ConstraintsOptimizationModule*, *RouteTimeModule, GeoDatabaseModule*, and *DispatchingModule*, each of which has different valid configurations. The *DispatchModule* has two valid implementations for different system to driver communication models. The *RouteTimeModule* has three different implementations. The *ConstraintsOptimizationModule* can be configured with three different algorithms. Finally, the *GeoDatabase* can use two different vendor implementations. These composition options support a total of 72 valid variants to construct from the PLA.

3 PLA Modeling, Domain Analysis, and Testing Challenges

Although PLAs can increase software reuse and amortize development costs, PLA configuration spaces are hard to analyze and test manually. Deploying, configuring, and testing a PLA in numerous configurations without intelligent modeling, domain analysis, and automation is expensive and/or infeasible. Large-scale product variants may consist of thousands of component types and instances [4] that must be tested. This large solution space presents the following key challenges to developing a PLA:

- **Challenge 1: Manually managing a PLA's configurations and constraints**. Traditional processes of identifying valid PLA variants involve software developers determining manually the software components that must be in a variant, the components that must be configured, and how the components must be composed. Such manual approaches are tedious and error-prone and are a significant source of system downtime [5]. Manual approaches also do not scale well and become impractical with the large configuration spaces typical of PLAs. In CONST, for example, there may be thousands of variations on freight types, licensing requirements, freight handling procedures, and local laws applying to transportation that require the PLA to have a substantial amount of variability.
- **Challenge 2: Determining what PLA configurations to test through domain analysis**. With hundreds or thousands of potential configurations, testing each possible configuration may not be feasible or cost effective. Developers must determine which PLA configurations will yield the most valuable information about the capabilities of different regions of the PLA configuration space. Determining how to perform this domain analysis is hard. For example, it may not be clear which freight routing algorithms in CONST yield poor performance when used together in a configuration.
- **Challenge 3: Managing the complexity of configuring, launching, and testing hundreds of valid configuration and deployment**. *Ad hoc* techniques often employ build and configuration tools, such as Make and Another Neat Tool (ANT) [6], but application developers still must manage the large number of scripts required to perform the component installations, launch tests, and report results. Developing these scripts can involve significant effort and require in-depth understanding of components. Understanding these intricacies and properly configuring applications is crucial to their providing proper functionality and quality of service (QoS) requirements [7]. Incorrect system configuration due to operator error has also been shown to be a significant contributor to downtime and recovery [5]. Developing custom deployment and configuration scripts for each variant leads to a significant amount of reinvention and rediscovery of common deployment and configuration processes. As the number of valid variants increases, there is a corresponding rise in the complexity of developing and maintaining each variant's deployment, configuration, and testing infrastructure. Automated techniques can be used to manage this complexity [8,9,10].
- **Challenge 4: Evolving deployment, configuration, and testing processes as a PLA evolves**. A viable PLA must evolve as the domain

changes, which presents significant challenges to the maintenance of configuration, deployment, and testing processes. Small modifications to composition rules can ripple through the PLA, causing widespread changes in the deployment, configuration, and testing scripts. Maintaining and validating the large configuration and deployment infrastructure is hard. Moreover, as PLA components evolve, it is essential that intelligent regression testing be performed on PLA variants to identify those that may become non-functional due to unforeseen side effects. For example, a change in a CONST component for assigning costs to shipments may have wide ranging affects on numerous configurations of the optimization engine. With a large variant solution space, it becomes even more difficult to rapidly evolve and validate the PLA.

The next section describes how we resolve these challenges via model-driven PLA testing and domain analysis techniques.

4 Model-driven Testing and Domain Analysis Techniques for Product-line Architectures

This section introduces modeling techniques for capturing PLA configuration rules and then describes how these models can be annotated with results from testing. It also presents constraint-based optimization techniques that can be used to analyze the model to derive configurations to test.

4.1 Feature Modeling

To address *Challenge 1* from *Section 3*, a modeling methodology is needed to capture the SCV of the application and reason about correct and incorrect configurations of the PLA. A widely used technique for formally representing the variability in a PLA is *feature modeling* [19,20,24]. Feature modeling describes the SCV of a PLA using a tree-like structure where each node in the tree represents a software *feature*. Features are an abstraction for an increment of functionality or a point of variability in the software architecture. For example, Figure 2 shows a partial feature model of CONST. The feature *Prioritized* represents whether or not the configuration of CONST is designed to handle prioritized shipments.

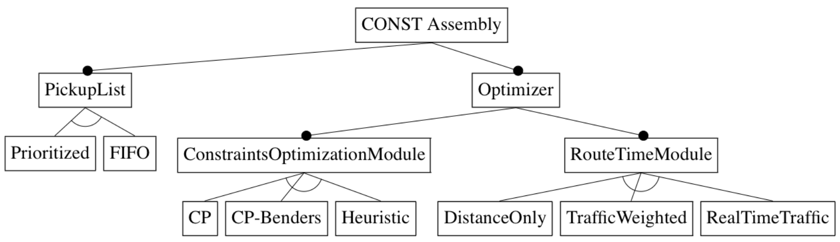

Figure 2. Segment of CONST Feature Model

Each unique configuration of a PLA is represented as a selection of features from the feature model. A unique feature selection is called a *variant*. The goal of PLA configuration with feature models is to find a variant that adheres to all of the configuration rules of the PLA and simultaneously meets the functional and non-functional requirements of the project.

As shown in Figure 2, the configuration rules of a PLA are encoded into the parent-child relationships in the feature model tree. The most basic feature modeling rule is that a feature can be selected if and only if its parent is also selected. The root feature must always be selected. In Figure 2, the *Prioritized* feature can only be selected if the parent *PickupList* feature is also selected.

Points of variability in the PLA and their constraints on configuration are specified as special types of parent-child relationships. The children of a feature can be combined into an XOR group where exactly one of the child features can be selected at a time. The children can also form a cardinality group, where the selection of the child features must conform to a cardinality expression. For example, a feature can specify that between 1…3 of its children can be selected.

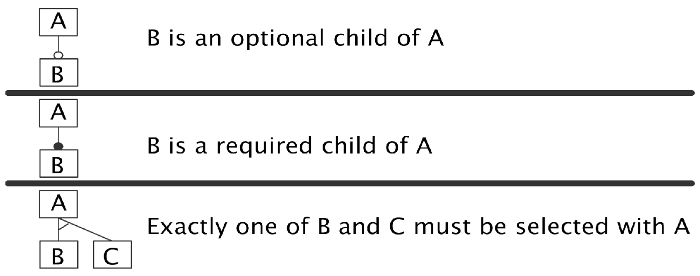
A
B
B is an optional child of A
A
B
B is a required child of A
A
B
C
Exactly one of B and C must be selected with A

Figure 3. Feature Model Notation

4.2 Mapping Test Result Data to Feature Model Quality Attributes

Addressing *Challenge 2* from *Section 3* requires developing an understanding of how test results map to points of variability in the PLA. For example, if all tests that include the features HazMat and TankerMonitoring fail, developers should be able to map this information to the feature model in order to further investigate the failure or add additional constraints to mark the feature set as invalid. To tackle this challenge, a combination of statistical analysis and feature modeling techniques can be used.

Quality Attributes [24] are an extension of feature modeling whereby attribute information is stored along with each feature. Cost information, for example, can be stored as an attribute on each feature to provide guidance on the overall configuration cost of a PLA variant. In terms of testing, a powerful approach to mapping test information to feature models is to use probabilistic data for quality attributes.

Probabilistic quality attributes are used to understand the probability that a set of feature selections will reach a desired goal. For example, given the current selection of features, what further set of features are most commonly selected to complete the feature selection. In terms of testing, probabilistic data can be used to understand the probability that the feature selection will fail one or more tests if a given set of additional features is added. If TankerMonitoring is selected, adding the HazMat feature will guarantee failure and thus its quality attribute will be set to 100%.

Values for the quality attributes are derived by evaluating the historical results of past tests. If two features have been tested together in three different variants and failed in one of them, the probability of failure for selecting both features would be 33%. Initially, no historical test data may be available, in which case, all values are set to 0%.

4.3 Feature Models and Constraint Satisfaction Programming

To help address Challenge 2, a method is needed to aid developers in deriving configurations to tests. Given a set of probabilistic quality attribute values, it may not be straightforward to understand which combination of features will yield the largest information gain for testing. The problem is that selecting a set of features to maximize a function of the features is an NP-Hard problem. To address this issue, automated methods can be used to automatically derive the set of features that will yield the best information gain.

Constraint Satisfaction Problems (CSPs) [26] are mathematical models that specify a set of variables and a set of constraints governing the values that those variables may be assigned. There are numerous automated tools, called constraint solvers, for automatically deriving solutions to CSPs that adhere to the CSP constraints and maximize or minimize a function of the variables. Feature models can be translated into CSPs where the solution to the CSP yields a valid selection of features. Moreover, the CSP can be solved using a constraint solver for a selection of features that maximizes or minimizes a function of the feature model features, which can help developers automatically derive configurations to test and address Challenge 2.

4.3.1 Mapping Feature Models to CSPs

A CSP is a problem defined by a set of variables for which values must be derived that meet a set of constraints. For example, $Y > 0$, $X + Y < 10$, $X \neq Y$, is a simple CSP over the integer variables X and Y. A valid solution, called a labeling of the variables, to the CSP is $X = 2$, $Y = 1$.

Constraint solvers offer the ability to automatically derive solutions to a CSP that maximize or minimize a given function. For example, a CSP could be asked to derive a solution to the CSP that maximizes the value of X – Y. The optimal solution in this case would be X = 8, Y = 1.

A feature model can be transformed into a CSP such that deriving a valid solution to the CSP produces a valid selection of features in the feature model. The transformation is achieved by:

A preorder traversal of the feature model is performed starting from the root feature

As the ith feature is traversed, a variable f_i is added to the CSP

a. The variable f_i has the domain [0,1]
b. After solving the CSP, a value of 1 for f_i indicates that the feature is selected

3. As the ith feature is traversed, for each quality attribute, k, of the feature, a variable a_{ik} is added to the CSP. The value of a_{ik} is set to the numerical value of that quality attribute for the ith feature.
4. The constraint, $f_0 = 1$, is added to ensure that the root feature is selected in any derived configuration
5. A second preorder traversal of the feature model is performed and at each feature, constraints encoding the parent/child relationships in the feature model are encoded as follows

    if f_k is a mandatory child of f_i, the constraint $f_k = 1 \Leftrightarrow f_i = 1$

    if f_i has children f_k and f_p in an XOR relationship, the constraint $f_i = 1 \Leftrightarrow (f_k = 1 \oplus f_p = 1)$

    if f_i has an optional child f_k, $f_k = 1 \Rightarrow f_i = 1$

Using this transformation process, a CSP is produced for which a correct labeling of the variables, $f_i \in F$, yields a feature selection that adheres to the feature model roles. For each feature, f_i, if $f_i = 1$, then the corresponding feature in the feature model is selected in the configuration. Similarly, if $f_i = 0$, the corresponding feature is not selected in the configuration.

Once the feature model is translated into a CSP, a constraint solver can be used to derive configurations that maximize or minimize functions of the features and quality attributes. For example, assume that the kth quality attribute of each feature is the cost of selecting it. The solver can be asked to derive a minimal cost configuration by asking the solver to derive a solution that minimizes the function: $Cost(F) = \Sigma f_i a_{ik}$

For each feature, f_i, if the feature is selected, its value is set to 1. In the function provided to the solver, this will cause the sum for Cost(F) to incur the cost of the ith feature a_{ik}. If f_i, is not selected, is set to 0, which causes its cost to be zeroed out and not included in the sum.

4.3.2 Deriving Test Plans that Maximize Information Gain

As discussed in *Section 4.2.2*, statistical information from test results can be encoded into a feature model's quality attributes. Automatic configuration using a CSP can be used to both help plan test processes to maximize the intake rate of statistical test data. Furthermore, a CSP-based configuration derivation process can find configurations that have the greatest or least probability of functioning correctly based on current statistical data gleaned from test results.

For example, assume that based on previous tests and statistical analysis, a failure quality attribute, a_i, has been constructed. Each quality attribute, a_i, specifies the probability that the configuration will fail one or more tests if it the feature f_i is selected. Using this failure probability attribute, configurations of the feature model can be derived that have the greatest probability of failure based on the provided feature model and failure statistics.

In the CSP, we can encode the probability of a configuration failing as $Fail(F) = Max(f_0a_{0,} f_1a_{1, \ldots} f_na_n)$, where n is the number of features. Determining a configuration that maximizes the probability of failure simply requires asking the constraint solver to derive a configuration that maximizes Fail(F). Moreover, configurations with low probabilities of failures can be generated by requesting that the solver minimize the value of Fail(F).

As *Challenge 4* from *Section 3* pointed out, the test plans must be evolved as the PLA evolves and new test results are obtained. Using a CSP-based configuration derivation process, new configurations to test can automatically be derived as the PLA changes. Moreover, the test plans can also be updated as each test results sheds more light on the PLA configuration space.

4.4 Test Automation from Feature Models.

To address Challenge 3, which is the complexity of deploying, configuring, and testing a PLA, we have developed *FireAnt*. FireAnt is an MDA tool that allows application developers to describe the components that form the common building blocks of their PLA and to construct feature models specifying how the blocks can be composed to form valid variants. FireAnt significantly reduces the cost of testing a PLA in the following key ways:

Test, Deployment, and Configuration Infrastructure Generation. FireAnt allows developers to describe the target hardware where variants will be deployed. Using a target hardware definition and the artifact mapping, FireAnt can automatically package all the archive files required to deploy each variant, as well as generate the required configuration scripts. These scripts may be in implemented in a variety of languages. Currently, FireAnt provides bindings for generating Another Neat Tool (ANT) build files.

Test Automation. FireAnt can use CSP configuration derivation techniques to generate a global configuration script that remotely deploys, configures, and tests variants automatically on each possible hardware target.

FireAnt was developed using the *Generic Eclipse Modeling System* (GEMS) [17], which is an open-source MDA environment built as an Eclipse plug-in. A GEMS-based metamodel describing the domain of PLA deployment, configuration, and testing was constructed and interpreted to create the FireAnt domain-specific modeling language (DSML) for PLAs. FireAnt's modeling environment uses GEM's support for multiple views to capture the feature model, deployment, configuration, and testing requirements of a PLA. The remainder of this section discusses how each of these views can be used to manage the complexity of testing a PLA and how the view addresses each of the challenges described in *Section 3*.

4.4.1 FireAnt Feature Modeling

To facilitate the analysis of the variant solution space and address Challenge 1 requires a formal grammar to describe the structure of the PLA and its valid configurations. This customization grammar can then be used to automatically generate and explore the variant solution space using the CSP techniques described in *Section 4.3.1*. In FireAnt, the *Logical Composition View* is a feature model for capturing the SCV of a PLA. This view allows developers to formalize what features are available in the PLA, the hierarchical relationships between features, and the rules for selecting groups of features.

To capture a formal definition of the PLA, the components on which it is based must be modeled. The *Feature* element is the basic building block in the Logical Composition View. A Feature represents an indivisible unit of functionality, such as an EJB or CORBA component. In the CONST application, the various algorithm implementations for the constraints optimization engine are represented as Features. A configuration is a valid composition of Features that produces a complete set of application functionality. Each configuration may require different source artifacts depending on the features that it contains.

The feature model rules are specified through composition predicates. FireAnt supports that standard feature modeling constraints for AND, Exclusive OR and optional features. The children of each feature are connected through a composition predicate to their parent to specify the rules governing their selection. In CONST, for example, the *ConstraintsOptimizationModule* is connected to the Exclusive OR predicate, which can be connected to each algorithm packaged with the optimizer to create a variant. This composition indicates that the *ConstraintOptimizationModule* is composed from one of the three algorithms.

To specify the compositional variability in the PLA, developers build Component, Assembly, and Predicate trees, which we call *Logical Composition* Trees. At the root of the tree is an Assembly representing the entire PLA. The root Assembly, Predicate, and children specify the modules that must be present to complete the PLA. Each level down the tree specifies the composition of smaller pieces of functionality.

In the CONST system, the root of the feature model is the CONST Assembly. The CONST Assembly is connected to an AND predicate and the predicate is in turn connected to the PickupList and Optimizer Assemblies, which specifies that both a PickupList and Optimizer must be present in CONST variants. The CONST Logical Composition tree is shown in Figure 4.

By capturing PLA compositional variability in a feature model through the Logical Composition tree, developers can formally specify how valid variants are composed. With a formal specification of the variant construction rules, FireAnt can automatically explore the variant solution space to discover all valid compositional variants of the PLA.

4.4.2 Dependency and Deployment Views

Simply capturing the configuration rules for the PLA is not sufficient to automate deployment and testing. FireAnt must have a specification of how the features in the feature model map down to individual source artifacts. For example, if the *ConstraintsOptimizationModule* is selected, what Java jar files need to be packaged into the final variant that is tested.

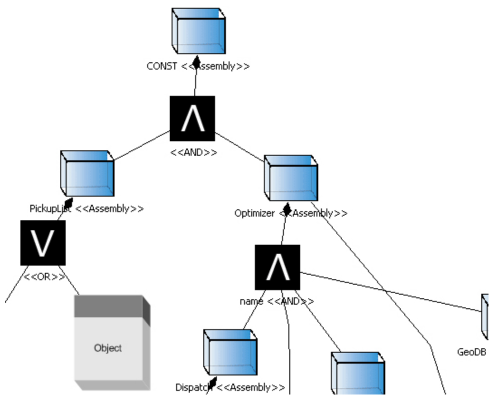

Figure 4. CONST Logical Composition Tree

To automate the packaging and configuration of variants and address Challenge 3, a dependency model must be developed to associate each feature with physical artifacts, such as JAR files, it relies on. This mapping from physical artifacts to PLA components can be used to automatically manage and package the artifacts and configuration scripts required for each variant.

The Dependency View manages the complexity of organizing and maintaining all the various physical artifacts required to deploy and configure a variant. A variant may contain hundreds of components, each with multiple physical artifacts required for their deployment. As the number of variants grows, it becomes hard to package all physical artifacts required to deploy a variant. Our CONST application, for example, has 72 unique valid package combinations that can be created for the variants. Each possible package requires a unique artifact set.

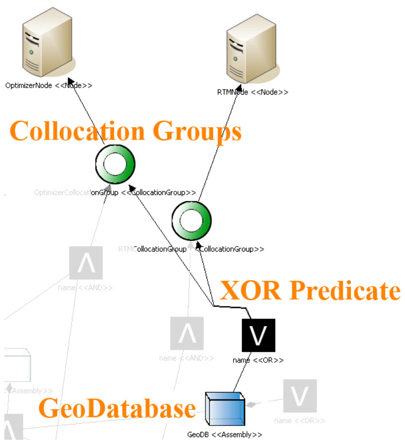

Figure 5. Logical Deployment Tree for the GeoDatabase Assembly

In distributed applications, developers may need to test the deployment of the application across different numbers and configurations of hardware. FireAnt's *Physical Deployment View* allows developers to specify rules on how features and their associated artifacts can be mapped to a series of remote hosts. FireAnt then takes each of these possible deployment variants and determines the unique packaging combinations of artifacts that are required for all possible valid deployments. Each unique package is called an *Egg*.

The *Physical Composition View* shows which physical artifacts are associated with each egg. Individual zip archives can be created for each deployment package by traversing the Physical Composition View trees. This view manages the complexity of determining what physical artifacts should be present in for the deployment of each variant's features to a host. FireAnt can automatically collect and zip all of the required artifacts for a variant's Assemblies by traversing the Physical Composition Tree.

5 Empirically Evaluating FireAnt Generative and Analytic Capabilities

A method for estimating the point at which developing a PLA becomes more cost effective than a traditional development approach is described in [2]. This paper defines the average economic or time cost of developing a variant manually without a PLA to be C_0 and the cost of the same development with automation to be C_1. To develop N variants using a manual approach, therefore, has a total cost of $N*C_0$. A is defined to be the initial overhead of performing SCV analysis and creating reusable components. C_1 is assumed to be smaller than C_0. The cost of developing the same N variants with a PLA is $A + N * C_1$.

For small numbers of variants, the initial cost *A* does not make a PLA cost effective. As the number of variants, *N,* grows, however, a PLA becomes more cost effective since $C_1 < C_0$. This section expands on this formula to estimate the cost of testing *N* variants developed manually and *N* variants developed with a PLA. We then show how FireAnt can decrease the initial cost *A* of developing a testing infrastructure for a PLA.

In the context of testing, we let T_0 be the cost of manually developing the infrastructure to test a variant and T_1 be the cost of developing the same infrastructure for a PLA variant. T_1 should be significantly smaller than T_0, since tests for determining the correctness of individual components can be reused for each variant. Moreover, any tests that check the correctness of a common element among the variants can be shared. To develop the testing infrastructure for a new variant, therefore, T_1 will only be comprised of the cost of developing tests for the unique components of each variant. With a manual approach, however, the variants do not share common components and tests cannot be shared among them making $T_0 > T_1$.

With a PLA, conversely, we incur an initial cost *A* of developing a flexible process for integrating and orchestrating the tests shared between variants. Even with the use of automation tools, such as those available for running JUnit tests, a developer must manually specify which tests to run for each variant. The total cost of testing *N* variants is $N*T_0$ for the manual approach and $A+N*T_1$ for the PLA. The goal of developing a testing infrastructure of a PLA is therefore to minimize *A* and ensure that the overhead of creating reusable tests does not make $T_1 > T_0$.

The remainder of this section reports the results from a series of experiments on our CONST case study described in *Section 2*. The goal of these experiments was to evaluate the extent to which FireAnt minimizes the initial cost *A* and does not require excess testing overhead that would increase T_1. Each experiment was repeated using several variations of the PLA to investigate how the performance of FireAnt scaled as the solution space grew. For testing, we used FireAnt 2.0 with a 2.2 Mhz AMD Athlon 3200 with 1 gigabyte of RAM running Windows XP and Eclipse 3.1.0. Our test cases were written using JUnit.

5.1 Solution Space Exploration Time

In our CONST case study, we evaluated the time required by FireAnt to discover and visualize all valid variants. Our initial implementation of CONST contained 17 EJBs, each packaged in individual Enterprise Application Resource (EAR) files with separate XML deployment descriptors to facilitate packaging. To analyze the impact of refactoring and its affect on FireAnt and the solution space, we created a new type of PickupList that was a hybrid priority/FIFO list. A waiting request's priority was determined by the time multiplied by the priority. Adding this PickupList implementation increased the number of valid variants to 108.

In our second refactorying, we provided two new graph representations for the optimization algorithm. One implementation used an in-memory graph representation. The second implementation used a disk-based graph representation scheme to reduce memory footprint. This refactoring increased the number of valid variants to 144. In the final refactorying, we combined both the PickupList and algorithm refactoring, which produced 216 valid variants. For each PLA, we calculated the time for FireAnt to generate all of the valid configurations (Eggs). The results of the tests are shown in Figure 5.

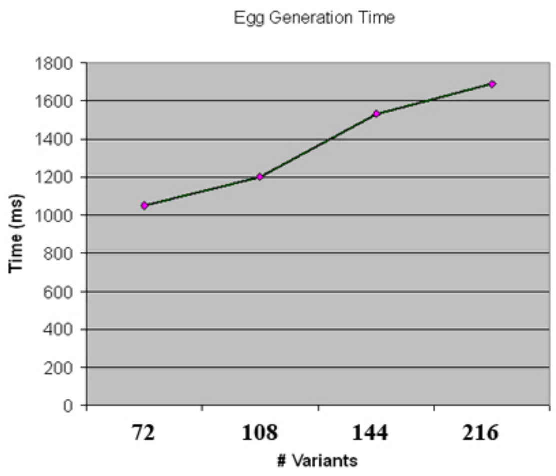

Figure 6. Solution Space Exploration Time

Figure 6 shows that the time required, D_v, to explore the solution space scaled at a rate of approximately $N * D_1 + K$, where D_1 is the time to required by FireAnt to discover a single variant and K a constant overhead. The maximum time required was less than 2 seconds. It can be seen that $D_1 = (D_V(72) - D_V(216)) / 144 < 700 / 144 = 4.8$ms. We posit that to discover the same set of variants manually, the time required would be $V(N) * N * D_0 + K$, where $D_0 > D_{1,}$ $V(N)$ is a function of N, and $V(N) > V(N-1) \geq 1$ for all N.

These results show that the discovery of a single variant is slower with a manual process and the time to discover all variants becomes increasingly worse as the number of variants grows, which stems from the inability of manual methods to scale as the complexity increases. Even without a $V(N)$ manual scaling factor and optimistically assuming $D_0 = 1000$ms, the FireAnt aided method is roughly 200 times faster. If a PLA architecture is used with a manual approach for assigning tests to variants, A varies in proportion to $V(N)$. By using FireAnt, $V(N)$ is removed and D_1 is far smaller than D_0, and thus, the cost, A, is significantly reduced for large numbers of variants.

5.2 Packaging Time

FireAnt also has the ability to collect all the resources needed to deploy a variant and package them in separate zip files for deployment across a group of nodes. FireAnt uses the Eggs and Dependency Tree to calculate the minimum physical artifacts required for each node's package. Along with the package generator, we created a translator that generates ANT build scripts for the deployment of the variant's packages. FireAnt can support generation of other scripting languages. We chose ANT, however, since it is platform-independent and well supported.

For each variant/deployment configuration, FireAnt generates local ANT scripts that are executed on each node to perform the Assembly installations. The generated ANT scripts invoke the appropriate PreDeployment, Deployment, and PostDeployment scripts required to install each component. After installing each component or assembly, the generated ANT scripts invoke any tests associated with the element in the Dependency view, which enables automated testing of each variant. FireAnt also generates a global deployment script to execute the deployment, configuration, and testing of each variant consecutively. Developers simply provide the scripts to configure and deploy the individual assemblies and / or components.

We used our AMD Athlon 3200 test platform to measure, O_v, which is how long it took FireAnt to package all of the resources and generate the ANT build scripts for each of the variants. We then measured the time required for FireAnt to collect and zip the files for each package. The results are shown in Figure 7.

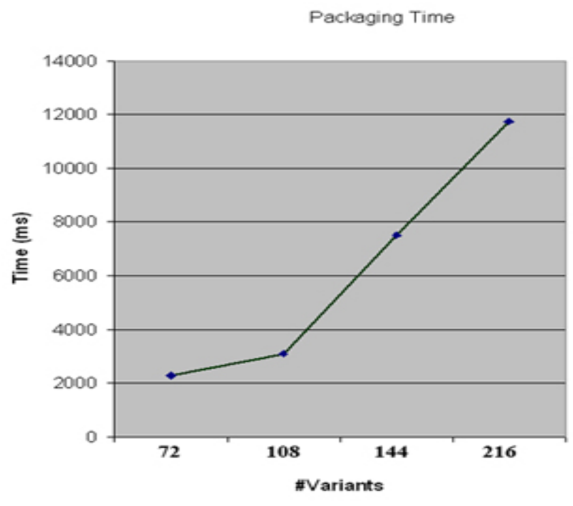

Figure 7. FireAnt Packaging Time

The results in this figure show that using FireAnt, O_v, = $N * P_1$, + K, where P_1 is the time taken to package and generate the configuration script for a single variant using FireAnt and K is a constant overhead. Again, a manual approach to accomplishing the same task would require that $O_v = V(N) * N * P_0 + K$, where P_0 is the time to manually package a variant, $P_0 > P_1$, $V(N)$ is a function of N, and $V(N) > V(N\text{-}1) \geq 1$ for all N. As can be seen from the results, $P_1 < (12000 - 2000) / 144 = 69.4$ms.

Assuming that a manual process could package all the artifacts required for a variant in 1000ms (which is extremely optimistic), the FireAnt aided method is still ~14 times faster. The FireAnt method again removes a $V(N)$ manual scaling factor, as well, from the cost A. FireAnt's packaging provides the ability to calculate and re-package all the variants automatically when new components are added to the PLA, which reduces developer effort and ensures that each variant's package footprint is always up-to-date. Thus, using FireAnt reduces the cost of R refactorings by $R * (V(N) - 1) * 14$. For large values of N, this cost savings will be significant.

5.3 Results Summary

FireAnt uses the techniques described in *Section 4* to automate (1) the generation of deployment scripts for variants, (2) the packaging of artifacts for variants, and (3) the testing of variants. These capabilities reduce the upfront cost, A, and enable rigorous testing of PLAs. They also address each of the four key challenges outlined in *Section 3*.

Due to the large number of variants it becomes costly for PLA developers to manually find and manage all possible variants without MDA tool support. This complexity increases the initial cost, A, of developing a PLA testing infrastructure since a developer must find all valid variants and determine which tests are required to ensure the proper functioning of each. In other words, $A \geq Dv + Ov$, where D_v is the time required to find each valid variant and O_v is the time required to generate an orchestration script for each variant that will execute the proper tests.

FireAnt reduces the initial cost A by automatically exploring the solution space and producing visualizations of valid variants for the developer. These capabilities significantly aid developer understanding of PLA variability and enables for the automated testing and packaging of each variant. Without automating the identification of variants of the PLA to test, it is hard to ensure that the PLA is tested properly, which is important in mission-critical domains.

6 Concluding Remarks

Product-line architectures (PLAs) can significantly improve the reuse of software components and decrease the cost of developing applications. The large number of valid variations in a PLA must be tested to ensure that only working configurations are used. Due to the large solution spaces it is infeasible or overly costly to use traditional *ad hoc* methods to test a PLA's variants.

By using MDA tools to capture the compositional and deployment variability in PLAs, we showed that much of the deployment, configuration, and testing of PLAs can be automated. This automation frees developers to focus on implementing reusable components and deployment and configuration scripts for known working units of functionality. Our experiments have shown that FireAnt can significantly reduce both the initial cost, A, of developing a PLA and the testing cost T_1 of each variant. FireAnt accomplishes this cost reduction by automating tedious and error-prone manual tasks, such as solution space exploration.

The following are our lessons learned from developing FireAnt and applying it to the EJB-based *Constraints Optimization System (CONST)* that schedules pickup requests to vehicles:

- **There is a larger up-front cost to adopt an automated test platform.** Initially, the cost of developing models for the MDA testing process increases development cost. Over time, however, this startup cost is amortized across variants of the SPL saving time and money.
- **Choosing the right statistical analysis technique for test results is an important concern**. This chapter introduces a few statistical analyses that can be used to populate quality attribute values from test results. There are a wide array of other types of analyses that can be used as well.
- **There may be unanticipated problems caused by the composition of two or more features** that may not be scriptable by FireAnt. For example, complex changes in source code may be needed. More work is needed to identify how to automate the generation of the deployment and configuration glue of PLA variants.
- **Deployment variations greatly expand the solution space** since each variant must be tested with each deployment variation. It is thus important to only model realistic deployment scenarios to restrict this space.

In future work, we are pursuing the use of FireAnt to create self-tuning installations. Many high-performance parallel computing applications, such as the Automatically Tuned Linear Algebra Software (ATLAS) [23], run performance tests in multiple configurations as part of the installation process. These applications can then interpret the performance results to optimize themselves for the given hardware.

We also plan to expand on the ATLAS approach by allowing FireAnt users to define a fitness function based on the performance metrics collected from the individual component tests. The FireAnt test automation framework will then be used to iteratively deploy variants in various configurations in an attempt to maximize this fitness function.

Developers only need to create the tests to collect the appropriate data, such as service rate, and then provide the logic to perform analyses on the results, such as throughput analysis using queuing networks, to score the configurations. FireAnt will use this cost function to automatically deploy, configure, test, and score each candidate variant in each valid component to hardware configuration. After all testing completes, FireAnt will collect the results and install the variant/component to hardware configuration with the highest score.

References

1. P. C. Clements and L. Northrop, *Software Product Lines – Practices, and Patterns*, Addison-Wesley, 2001.
2. J. Coplien, D. Hoffman, D. Weiss, “Commonality and Variability in Software Engineering,” IEEE Software, Volume 15, Issue 6, Nov.-Dec. 1998 Page(s):37-45.
3. E. J. Weyuker, “Testing Component-based Software: A Cautionary Tale,” IEEE Software, September/October 1998
4. D. Sharp, “Avionics Product Line Software Architecture Flow Policies,” Proc of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC), Oct 1999, St. Louis, MO.
5. D. Oppenheimer, A. Ganapathi, D. Patterson, “Why do Internet Services Fail, and What can be Done about It?,” Proc of the USENIX Symposium on Internet Technologies and Systems, Mar 2003, Seattle, WA.
6. Apache Foundation: Apache Ant. http://ant.apache.org.
7. A. Krishna, E. Turkay, A. Gokhale, D. Schmidt, “Model-Driven Techniques for Evaluating the QoS of Middleware Configurations for DRE Systems,” I Proc of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium, Mar 2005, San Francisco, CA.
8. A. Sloane, “Modeling Deployment and Configuration of CORBA Systems with UML,” Proc of the 22nd International Conference on Software Engineering, June 2000, Limerick, Ireland.
9. G. Edwards, G. Deng, D. Schmidt, A. Gokhale, B. Natarajan, “Model-driven Configuration and Deployment of Component Middleware Publisher/Subscriber Services,” Proc of the 3rd ACM Conference on Generative Programming and Component Engineering, Oct 2004, Vancouver, CA
10. A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. Schmidt, B. Natarajan. "Skoll: Distributed Continuous Quality Assurance," pp. 459-468, 26th International Conference on Software Engineering, May 2004, Edinburgh, Scotland.
11. A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, G. Karsai, “Composing Domain-Specific Design Environments,” I*EEE Computer*, Nov. (2001).

12. M. Harrold, D. Liang, and S. Sinha, “An Approach to Analyzing and Testing Component-Based Systems,” Proc of the ICSE'99 Workshop on Testing Distributed Component-Based Systems, May 1999, Los Angeles, CA.
13. N. Wang, C. Gill, D. Schmidt, and V. Subramonian, “Configuring Real-time Aspects in Component Middleware,” Proc of the Conference on Distributed Objects and Applications, October 2004, Cyprus, Greece.
14. V. Matena and M. Hapner, “Enterprise Java Beans Specification,” Version 1.1. Sun Microsystems (1999).
15. H. Ding, L. Kihwal, L. Sha, “Dependency Algebra: A Theoretical Framework for Dependency Management in Real-Time Control Systems,” Proc of the 12th IEEE International Conference on the Engineering of Computer-Based Systems, Apr 2005, Greenbelt, MD.
16. J. Hatcliff, W. Deng, M. Dwyer, G. Jung, V. Prasad, “Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems,” Proc of the 25th International Conference on Software Engineering, May 2003, Portland, OR.
17. J. White, D. Schmidt, "Simplifying the Development of Product-Line Customization Tools via MDA," Workshop: MDA for Software Product Lines, ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, October 2005, Montego Bay, Jamaica.
18. T. Mannisto, T. Soininen, R. Sulonen, “Product Configuration View to Software Product Families,” Proc of the 10th Int. Workshop on Software Configuration Management (SCM-10) of ICSE, May 2001, Ontario, Canada.
19. K.Kang, S.G.Cohen, J.A.Hess, W.E.Novak, and S.A.Peterson, “Feature Oriented Domain Analysis (FODA) - Feasibility Study,” Technical report CMU/SEI-90-TR-21, Carnegie-Mellon University, 1990, Pittsburg, PA.
20. T. Asikainen, T. Männistö, and T. Soininen, “Representing Feature Models of Software Product Families Using a Configuration Ontology,” Proc of the ECAI 2004 , Workshop on Configuration. Aug 23rd 2004, Valencia, Spain.
21. M. Popovic and I. Velikic, “A Generic Model-Based Test Case Generator,” Proc of the 12th IEEE International Conference on the Engineering of Computer-Based Systems (ECBS 2005), 4-7 Apr 2005, Greenbelt, MD, USA.
22. J. Guo, Y. Liao, J. Gray, B. Bryant, “Using connectors to integrate software components,” Engineering of Computer-Based Systems (ECBS 2005), 4-7 Apr 2005, Greenbelt, MD, USA.
23. R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the atlas project. Technical report, Dept. of Computer Sciences, Univ. of TN, Knoxville, March 2000.
24. K. C., Kang, J. Lee, P. Donohoe, “Feature-oriented Product Line Engineering,” *IEE Software*, Vol. 19, Issue 4, July-Aug. 2002, Pages 58-65.

25. Z. R. Dai, “Model-driven Testing with UML 2.0,” Second European Workshop on Model Driven Architecture (MDA), September 7th-8th 2004, Canterbury, England.
26. D. Benavides, P. Trinidad, and A. Ruiz-Cortes. Automated Reasoning on Feature Models.17th Conference on Advanced Information Systems Engineering (CAiSE 05, Proceedings), LNCS, 3520:491–503, 2005.