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

Jules White and Douglas C. Schmidt

Vanderbilt University, Department of Electrical Engineering and Computer Science, Nashville, TN, 37212, USA
{jules, 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 FireAnt, which is a tool for the model-driven development (MDD) of PLA deployment and testing plans. To validate FireAnt, we use a distributed constraints optimization system case study to illustrate the cost savings of using an MDD approach for the deployment and testing of PLAs.

1 Introduction

Product-line architectures (PLAs) [1] 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 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. Testing is essential since even variants that obey the compositional rules of the PLA may not function properly. 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 it is connected to [3]. It is therefore crucial that PLAs undergo significant testing to validate the correctness of all possible configurations of their components.

Deploying, configuring, and testing all valid variants of a PLA without automated methods, however, 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 – Creating a model of the PLA's variant solution space. 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 down-time [5]. Manual approaches also do not scale well and become impractical with the large solution spaces typical of PLAs.

Challenge 2 - Managing the complexity of hundreds of valid configuration and deployment options for product line variants. *Ad hoc* techniques often employ build and configuration tools, such as Make and Another Neat Tool (ANT) [6], but application deployers still must manage the large number of scripts required to perform the component installations. 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 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 down-time 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 and configuration infrastructure. Automated techniques can be used to manage this complexity [8,9,10].

Challenge 3 - Evolving deployment, configuration, and testing processes as a PLA evolves. To be viable, a 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 and configuration scripts. Maintaining and validating the large configuration and deployment infrastructure is hard. Moreover, as PLA components evolve, it is essential that regression testing be performed on all PLA variants to identify those that may become non-functional due to unforeseen side effects. With a large variant solution space, it becomes hard to rapidly evolve and validate the PLA.

Challenge 4 - Ensuring that a PLA is rigorously tested in all valid configurations. Even when *model-driven development* (MDD) [11] techniques and tools, are used to generate the customization, composition, packaging, and deployment code to implement PLA variants [9, 12] it is still impossible to ensure that all correctly constructed variants will perform as modeled. In mission-critical domains, such as avionics and automotive systems, it is essential that nonfunctional variants be discovered. The large number of valid variants, however, makes it hard to test all the valid con-

figurations and deployments. Rapid regression testing in response to component changes is even harder.

Challenge 5 – Identifying the performance characteristics of the variants. The performance characteristics of each variant must be well understood to select the appropriate variant to meet the QoS requirements. Only by performing rigorous performance tests on the entire variant solution space is it possible to choose the optimal variant for a requirement set. which is hard as the number of variants increases. Performance testing is further complicated in PLAs for distributed systems where the deployment and collocation options can have large impacts on the system performance.

Challenge 6 - Managing the packaging complexities of a product line variant. Each package that is deployed to a target must contain the minimal amount of physical artifacts, such as DLLs and Java ARchive (JAR) files, required to deploy the variant. Minimizing the footprint of the installation artifacts, ensures that the target environment does not have wasted disk space and that bandwidth and time is not used to transfer unnecessary items to the target. It is also desirable to have a configuration process that is specific to the variant to ensure that unexpected complications involving unused packages do not affect the deployment. Creating individual packages and configuration scripts is costly using traditional approaches due to the large solution space. Maintaining the packages as the PLA evolves is even more challenging.

This paper presents three contributions to the deployment, configuration, and testing of PLAs. First, we describe the structure and functionality of *FireAnt*, which is an open-source model-driven development (MDD) Eclipse plug-in for specifying the SCV of a PLA, the artifact and configuration dependencies of each component, and the intended deployment destinations of each component. Second, we describe the structure and functionality of *FireAnt's* deployment, configuration, and test-generation infrastructure, which explores the variant solution space and produces build and deployment scripts to configure each valid variant. Third, we present empirical results that show FireAnt significantly reduces the overhead of developing and maintaining a deployment and testing infrastructure for a PLA.

The remainder of this paper is organized as follows: Section 2 provides a motivating case study example for our work; Section 3 describes the FireAnt MDD tool for modeling PLAs; Section 4 quantifies the benefits of the FireAnt code generation capabilities in the context of our case study; Section 5 compares our work with related research; 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. It uses a constraint-optimization engine to find a cost effective assignment of drivers and trucks to pickups. 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.

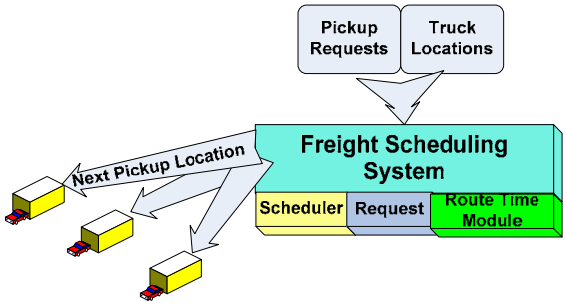

Figure 1: Highway Freight Shipment Scheduling Architecture

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. CONST also needs to handle failures differently, depending on the target domain. For scheduling limousines to pickups, for example, 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, GeoDatabase-*

Module, 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 be constructed from the PLA.

3 Modeling PLA Deployments and Configurations with FireAnt

To address the challenge of deploying, configuring, and testing a PLA, we have developed *FireAnt*. FireAnt is an MDD tool that allows application developers to describe the components that form the common building blocks of their PLA and to construct AND/OR trees 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:

1. **SCV Capture:** FireAnt models components common to the application and allows variability to be described formally using AND/OR trees, which enables developers to express rules governing a PLA. FireAnt can also capture the deployment variability in an application, e.g., to specify which components can be deployed together and which cannot.
2. **SCV to Artifact Mapping:** FireAnt provides views that allow developers to specify the physical artifacts, such as Java Archive (JAR) files, required for a common element. Variabilities can be mapped to configuration scripts that configure them properly.
3. **Solution Space Discovery:** Using model interpreters, FireAnt can automatically infer all valid variants from the commonality and variability trees. FireAnt can combine this information with artifact mapping information to show the required artifacts and configuration scripts for a variant.
4. **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.
5. **Test Automation:** FireAnt can generate a global configuration script that remotely deploys, configures, and tests each variant automatically on each possible hardware target.

FireAnt was developed using the *Generic Eclipse Modeling System* (GEMS) [17], which is an open-source MDD 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 SCV, deployment, configuration, and testing re-

quirements 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 1.

3.1 Logical Composition View

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. The *Logical Composition View* is the aperture for capturing the SCV of a PLA. This view allows developers to formalize what components are available in the PLA, what assemblies can be constructed, and how each assembly is composed. As with other approaches that capture the variants based on system structure [18] rather than feature modeling [19][20], in our approach requirements are expressed as configurations of components, i.e., features are modeled as variabilities in our SCV analysis.

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 Logical Composition View. A Component 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 EJB components. *Assemblies* are valid compositions of Components and other Assemblies that provide a higher level of functionality. Assemblies may require different source artifacts for different configurations or compositions. They can be composed by specifying a composition predicate, AND or Exclusive OR and the Components or Assemblies to which the predicate should be applied. 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. Assemblies can also be constructed hierarchically from other Assemblies to capture the compositional variability in a PLA.

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 tree 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 2.

By capturing PLA compositional variability in a 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. Section 3.4

discusses how FireAnt explores the solution space and uses it to automate the testing, deployment, and configuration of PLAs.

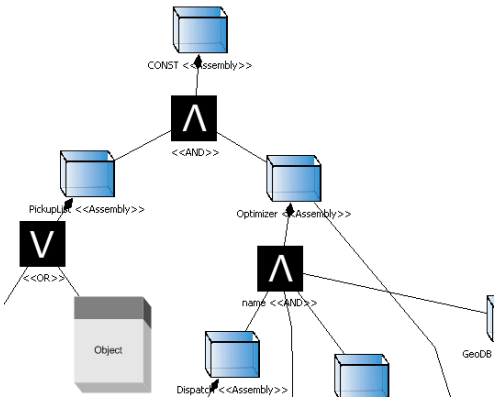

Figure 2: CONST Logical Composition Tree

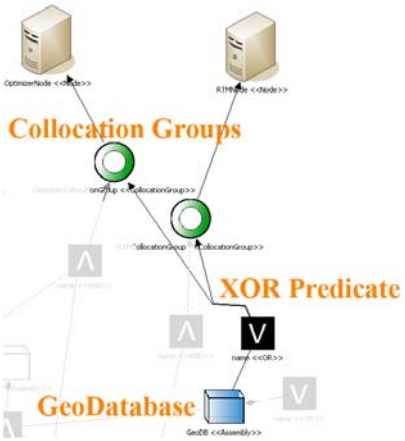

Figure 3: Logical Deployment Tree for the GeoDatabase Assembly

3.2 Logical Deployment View

It is essential to formally describe the variability in the deployment of the PLA to automate the configuration, deployment, and testing of the variants with multiple collocation strategies, hardware and OS platforms, and other performance critical variations. FireAnt's Logical Deployment View is used to specify a deployment grammar for the PLA. The Logical Deployment View addresses Challenges 2, 3, and 4. It is designed to capture and manage the complexity of the large number of deployment possibilities for a variant.

The Logical Deployment View describes which Assemblies are deployed, which Assemblies are collocated, and on what nodes they are deployed. Each top-level assembly is associated with one or more deployment predicates. These predicates are the same as in the Logical Deployment View. The predicates are then connected to one or more Collocation Groups. The mappings from Assemblies to Collocation Groups to Nodes form the *Logical Deployment Trees*. These trees specify what valid deployment variations are allowed in the PLA.

3.3 Dependency View

To automate the packaging and configuration of variants and address Challenges 2 and 6, a dependency model must be developed to associate each component 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.

3.4 Managed Views: Physical Composition and Deployment

The large size of the variant space makes it impractical to generate it manually. It is therefore essential to provide PLA developers with views of the solution space that are generated automatically from grammars describing the PLA's structure and variability. These generated views can then be used to address Challenges 1, 4, and 6.

The FireAnt managed views visualize various aspects of the variant solution spaces that are impractical to create by hand. They catalog the current possible compositional variants and deployment variations. These views are called managed views because they are generated by FireAnt and are not edited by users.

FireAnt creates the *Physical Deployment View* by traversing the Logical Composition Tree and calculating all possible combinations of Assemblies that can be deployed to each node. FireAnt then takes each of these possible variants and determines the unique packaging combinations of components 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 Assemblies. FireAnt can automatically collect and zip all of the required artifacts for a variant's Assemblies by traversing the Physical Composition Tree.

4 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 $T_1 > T_0$.

This section reports the results from a series of experiments on our CONST case study. 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.

4.1 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 re-factoring 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 re-factoring, 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 re-factoring, 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 4 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 $D_1 = (D_V(72) - D_V(216)) / 144 < 700 / 144 =$ 4.8ms. 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. That is, 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. This is a result of 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.

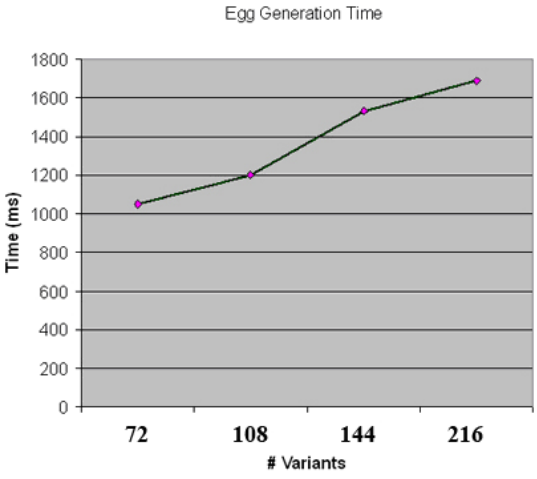

Figure 4: Solution Space Exploration Time

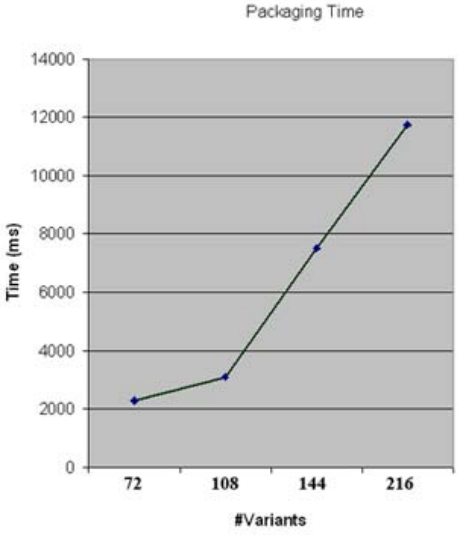

Figure 5: FireAnt Packaging Time

4.2 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 5.

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 $V(N) > V(N-1) \geq 1$ 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 repackage 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.

4.3 Results Summary

FireAnt uses the managed views described in Section 3.4 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 six key challenges outlined in Section 1.

Due to the large number of variants it becomes costly for PLA developers to manually find and manage all possible variants without MDD 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

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 identifying all possible variants of the PLA, it is hard to ensure that the PLA is tested properly, which is important in mission-critical domains.

5 Related Work

In [24], Kang et al., propose a modeling technique for PLAs called Feature Oriented Reuse Method (FORM). FORM captures the key variabilities in a PLA as features that can be present in a variant. A market analysis is first performed on the PLA to categorize and understand the important features of the variants. These features are then cataloged and mapped to the underlying object model to determine how they affect the underlying system structure. FireAnt uses a more general model of the variability in a PLA. First, FireAnt allows for variations in the configuration parameters of the underlying object model, such as thread priorities, to be modeled as variabilities. Small permutations in threading models can be applied to variants with identical features. That is, FireAnt allows for more flexibility in what is considered a variability. This can be particularly important for performance testing purposes where many configurations of a single feature variant may wish to be tested. Another key difference between FORM and FireAnt is that FireAnt provides a model-driven development tool that can automate the generation and management of an entire product line's deployment, configuration, and testing infrastructure. These powerful generative capabilities are what decrease the initial cost, *A*, and incremental costs of testing variants.

An algebra for specifying and deriving fault dependencies and propagations between components is proposed in [15], which focuses on investigating how known dependencies and assumptions of components can help predict fault propagation. FireAnt is designed not to predict fault propagation but to discover the component fault properties. FireAnt can be used to automate the discovery of faulty component compositions and configurations in PLAs, which helps identify and catalog the assumptions that were not being met by failing variants. The success of future evolutions to the PLA could then be predicted using the dependency algebra. FireAnt tests all known configurations of components and assumes that the tests provided by developers properly cover each variant. The dependency algebra in [15], could be used to help properly craft the test scripts invoked by FireAnt.

Another approach to model-driven testing is presented in [24] that uses the UML 2.0 Testing Profile and model transformations to generate tests. This work provides an effective means of relating the system design to the design of the testing process. This approach, as with other approaches, does provide a mechanism for automatically exploring a solution space and determining which tests should be applied to which variants, which is critical for PLAs. A UML-driven testing process, however, could be used in conjunction with FireAnt. FireAnt does not place restrictions on the testing framework or how tests are developed. Thus, developers could first use a UML-based test development infrastructure to produce tests that were orchestrated by FireAnt.

Other approaches to testing large-scale systems involve MDD tools that generate test cases from operational profiles [21]. FireAnt automates the testing of all variants. A complementary approach is to use statistical methods to generate test cases. Test cases created from tools, such as in [21], could be used to generate the testing orchestrated by FireAnt. Unlike FireAnt, however, this tool is not specific to PLAs.

Model-driven component design tools, such as Cadena [16], exist for Eclipse. Cadena is a model-driven development tool for designing component systems that provides the capability to package components, generate build scripts for them, and generate test infrastructure. Cadena supports development of multiple models to provide strongly typed modeling for PLAs. These tools, however, focus on the structure and implementation of the PLA. Furthermore, this approach is tailored to modeling specific variants of the PLA. Cadena is not designed to explore solution spaces and generate test infrastructure to cover all possible PLA variants. Again, this generative capability is one of the keys to rigorously testing a PLA to identify non-functional variants and unknown component dependencies, decreasing the cost of testing, and mitigating the cost of PLA evolution on the deployment, configuration, and testing infrastructure.

Techniques for improving the reuse of components by identifying and adapting components with compatible interfaces are presented in [22]. FireAnt can be applied to allow developers using these techniques to run automated tests of their assumptions about the compatibility of various components' interfaces. The results could then be used to refine the component dependency relationships. Pairing these works could provide an iterative environment for modeling and testing component compositions created through adaptive reuse.

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 MDD 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 may be unanticipated problems caused by the composition of two or more assemblies that may not be scriptable by FireAnt. 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 MDD," Workshop: MDD 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