Automating Product-Line Variant Selection for Mobile Devices

Jules White and Douglas C. Schmidt
Vanderbilt University
Nashville, TN, USA
{jules, schmidt}@dre.vanderbilt.edu

Egon Wuchner and Andrey Nechypurenko
Siemens AG, Corporate Technology (SE 2)
Munich, Germany
{egon.wuchner, andrey.nechypurenko}@siemens.com

Abstract

Product-line architectures (PLAs) designed for mobile devices create a unique challenge for automated product variant selection engines since variants must be derived on-the-fly as devices are discovered. Current automation techniques do not incorporate device resource consumption constraints into variant selection and do not address how a PLA can be designed to improve automated variant selection speed. This paper presents a tool called Scatter whose input is (1) the requirements of PLA construction and (2) the resources available on a discovered mobile device and whose output is the optimal variant that can be deployed to the device. Scatter provides automatic variant selection based on configuration and resource constraints and also ensures that variant selection is optimal with regard to a configurable cost function. The paper presents our results from experiments with Scatter and how PLA design decisions affect a constraint-based variant selection engine's solving speed.

1 Introduction

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

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

Product-line architectures (PLAs) [5] are a promising approach to help developers manage the complexity of the variability between mobile devices [2, 29, 20]. PLAs [5] 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 [8]. 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.

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

Current techniques for automating variant construction from component-based models or feature models, such as those presented in [3, 15, 19, 22, 24], do not sufficiently address various challenges of designing and implementing an automated approach to selecting a product variant for a mobile device. One common capability lacking in each of these approaches is the ability to consider resource consumption constraints, such as the total available memory

consumed by the features selected for the variant must be less than 256 kilobytes. Resource constraints are important for mobile devices since resources are typically limited. Some resources, such as cellular network bandwidth, also have a measurable cost associated with them and must be conserved.

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

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

- We describe Scatter's graphical requirement and resource specification mechanisms and show how they facilitate the capture and analysis of a wide variety of requirement types
- We discuss how Scatter transforms requirement specifications into a format that can be operated on by a constraint solver and how we extend existing constraint-based automation approaches [3] to include resource constraints
- We describe the automated variant selection engine, based on a Constraint Logic Programming Finite Domain (CLP(FD)) solver [12, 25] and show how it can rapidly produce both correct and optimal variants based on the requirements
- We present data from experiments that show how PLA constraints impact variant selection time for a constraint-based variant selection engine.
- We describe PLA design rules that we have gleaned from our experiments that help to improve variant selection time when using a constraint-based approach.

The remainder of this paper is organized as follows: Section 2 describes the challenges of selecting product variants for mobile devices; Section 3 presents the problems of capturing the requirements and resources for deploying PLA variants to mobile devices and discusses how Scatter addresses them; Section 4 shows how Scatter automatically transforms PLA requirements and mobile device resources into a model that can be operated on by the CLP(FD) based variant selector; Section 5 analyzes the performance results of applying Scatter to variant selection for an example PLA; Section 6 compares our approach with related work; and Section 7 presents lessons learned and concluding remarks.

2 Challenges of Automated Variant Selection for Mobile Devices

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

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

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

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

3 Capturing PLA and Mobile Device Requirements

Traditional processes of identifying valid PLA variants involve software developers manually determining the software components that must be in a variant, the components to configure, and how to compose and deploy the components. In addition to being infeasible in a pervasive environment (where the target device signatures are not known ahead of time and variant selection must be done on demand), such manual approaches are tedious and error-prone and are a significant source of system downtime [10]. Manual approaches also do not scale well and become impractical with the large solution spaces typical of PLAs.

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

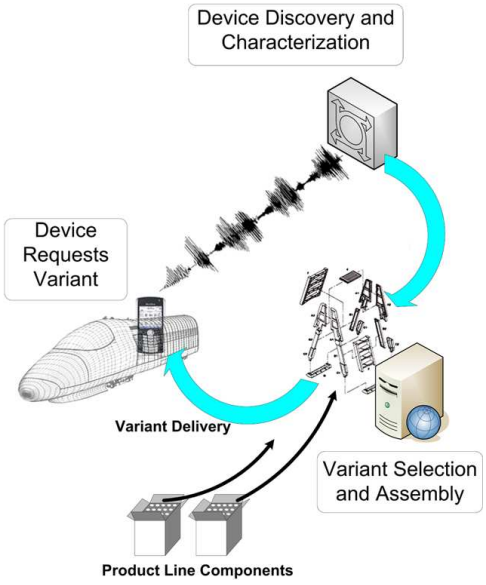

Figure 1: Selecting a Train Ticket Reservation Service for a Device

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

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

3.1 Scatter Overview

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

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

2. A compiler that converts the graphical models from the Scatter modeling tool into a both a Prolog knowledge base and a Constraint Satisfaction Problem (CSP) [12, 25] that can be operated on using a Prolog constraint solver. Scatter's formulation of the CSP is an extension of the model presented in [3], that includes resource constraints between components or features.

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

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

Scatter is implemented using the open-source Generic Eclipse Modeling System (GEMS) [27, 28], which is part of the Eclipse Generative Modeling Technologies (GMT) project. GEMS provides a convenient way to define the metamodel, *i.e.*, the visual syntax of the modeling language. Based on the metamodel, GEMS automatically generates

a graphical editor that enforces the grammar specified in the metamodel. Scatter extends our previous work using Role-based Object Constraints (ROCs) and Model Intelligence [21, 26]. Models created in Scatter are transformed via the ROCs infrastructure into formats that can be operated on by a constraint solver.

3.2 Scatter Graphical PLA Models

To facilitate the analysis of the variant solution space requires a formal grammar to describe the structure, commonality, and variability (SCV) analysis of the PLA and its valid configurations. This customization grammar can then be used to automatically generate and explore the variant solution space. Scatter provides a visual modeling tool for capturing the SCV of a PLA, as seen in Figure 2. This view allows developers to formalize which components are available in the PLA, what applications can be constructed, and how each application is composed. The components can be used as an abstraction to describe a PLA both on system structure [17] or using feature modeling [3, 13]. In our approach, configurations of components or features can be modeled as variabilities using Scatter's SCV model.

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

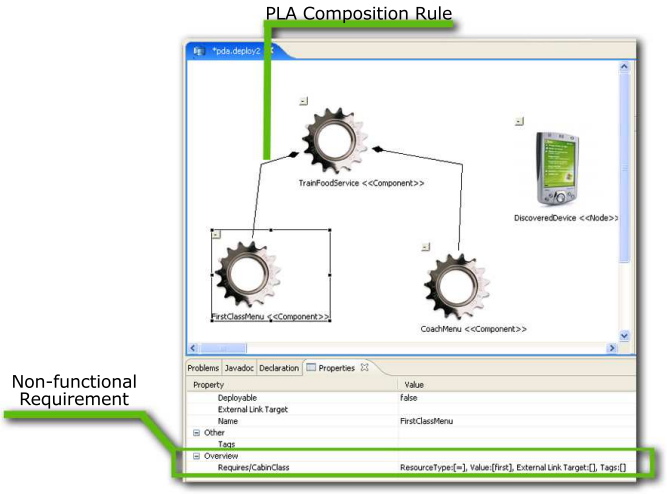

Figure 2: Scatter PLA Composition and Non-functional Requirements

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

Component dependencies can be constructed hierarchically from other components with dependencies to capture the compositional variability in a PLA. Components can also have composition rules with predicates that refer to arbitrary other components in the model. This mechanism is identical to the concept of feature references [9]. To specify the compositional variability in the PLA, developers build *Component* and *Predicate* graphs that show the dependencies and composition rules of the applications and their constituent pieces.

By capturing PLA compositional variability, developers can formally specify how valid variants are composed. With a formal specification of the variant construction rules, Scatter can then automatically explore the variant solution space to discover all valid compositional variants of the PLA for a given device, as discussed in Section 4.

3.3 Non-functional Requirements Capture

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

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

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

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

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

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

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

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

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

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

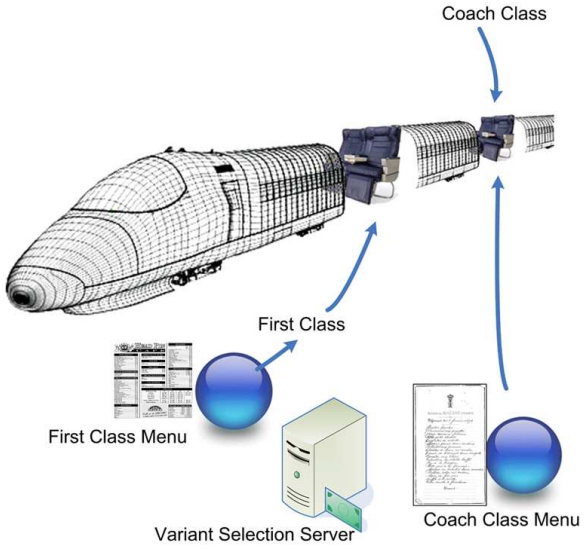

Figure 3: Cabin Class Constraints for Train Menu Variant Selection

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

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

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

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

The restrictions imposed by the specification format are only on the types of comparisons that can be done and not

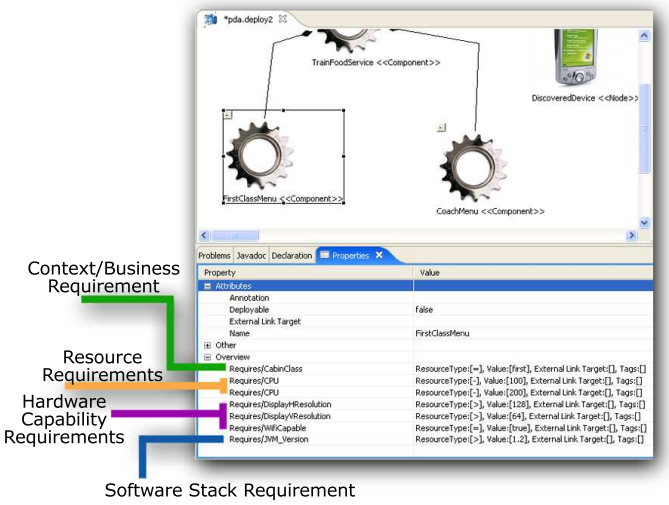

Figure 4: Capturing Mixed Non-functional Requirement Types in Scatter

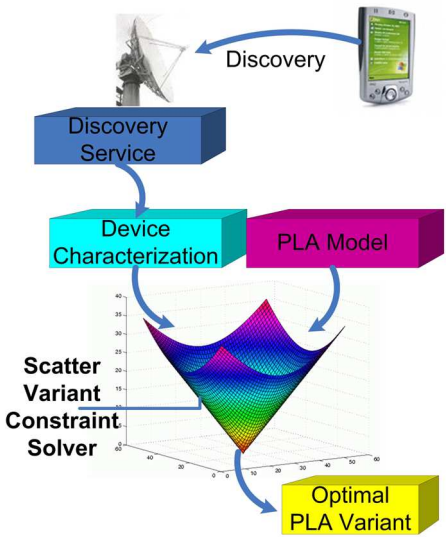

Figure 5: Scatter Integration with a Discovery Service

on the data that the comparison is based upon. This freedom in constraint specification allows Scatter's variant selection to incorporate a large array of datatypes that a device discovery service could provide. This setup allows other services to pre-process the data used by the variant selector and thus allow it to operate on very complex data sets.

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

3.4 Discovery and Device Signatures

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

Scatter exposes a SOAP-based web service and a CORBA remoting mechanism for remotely communicating device characterizations as they are discovered. The properties of a device are reported back to Scatter as key/value pairs. The keys match the names of the non-functional properties constrained by the non-functional requirements in the Scatter graphical model. As discussed in Section 4, these constraints and key/value pairs are used by the variant selection engine to filter the list of variants that can be deployed to a device.

4 Scatter Variant Selection Engine

Scatter provides an automated variant selector that leverages Prolog's inferencing engine and the Java Choco CLP(FD) constraint solver [1]. The Scatter solver uses a layered solving approach to help reduce the combinatorial complexity of satisfying the resource constraints. Scatter prunes the solution space using the PLA composition rules and the local non-functional requirements so that only variants that can run on the target infrastructure are considered. The resource constraints are a form of bin-packing an NP-Hard problem [6]. This layered pruning helps improve selection speed and enables more efficient solving. As shown in the Section 5, this layered pruning can significantly improve variant selection performance.

4.1 Layered Solution Space Pruning

Initially, the variant solution space contains many millions or more possible component compositions. Solving the resource constraints is thus time consuming. To optimize this search, Scatter first prunes the solution space by eliminating components that cannot be deployed to the device because their non-functional requirements, such a JVMVersion or CabinClass, are not met. After pruning away these components, Scatter evaluates the PLA composition rules to see if any components can no longer be deployed because one of their dependencies has been pruned in the previous step. After pruning the solution space using

the PLA composition rules, the resource requirements are considered. After solving the resource constraints, Scatter is left with a drastically reduced number of deployment solutions to select from. At this point, if there is more than one valid variant remaining, Scatter uses a branch and bound algorithm to iteratively try and optimize a developer-supplied cost function by searching the remaining valid solutions.

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

4.2 Using CLP(FD) to Solve Resource Constraints

After performing this initial pruning of the solution space, the resource and PLA composition constraints are turned into an input for a CLP(FD) solver. The transformation is an extension of the model proposed in [3] to include resource consumption constraints. The model is also extended to allow for feature references.

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

Selecting a a product variant can be reduced to a CSP. Scatter constructs a set of variables $DC_0 \ldots DC_n$, with domain $[0,1]$, to indicate whether or not the ith component is present in a variant. A variant therefore becomes a binary string where the i^{th} position represents if the i^{th} component (or feature) is present. Satisfying the CSP for variant selection is devising a labeling of $DC_0 \ldots DC_n$ such that the composition rules of the feature model are adhered to.

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

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

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

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

5 Scatter Performance Results

A key question is how fast Scatter performs and whether or not online variant selection is possible. To test Scatter's performance, we developed a series of progressively larger PLA models to evaluate solution time. The tests focused solely on the time taken by Scatter to derive a solution and did not involve deploying components. We also tested how various properties of PLA composition and local non-functional constraints affected the solution speed. Our test

were performed on an IBM T43 laptop, with an 1.86ghz Pentium M CPU and 1 gigabyte of memory.

Note that optimization and satisfaction of resource constraints is an NP-Hard problem, where it is always possible to play the role of an adversary and craft a problem instance that provides exponential performance [6]. Constraint satisfaction and optimization algorithms often perform well in practice, however, despite their theoretical worst-case performance. One challenge when developing a PLA that needs to support online variant selection is ensuring that the PLA does not induce worst-case performance of the selector. We therefore attempted to model realistic PLAs and to test Scatter's performance and better understand the effects of PLA design decisions.

5.1 Pure Resource Constraints

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

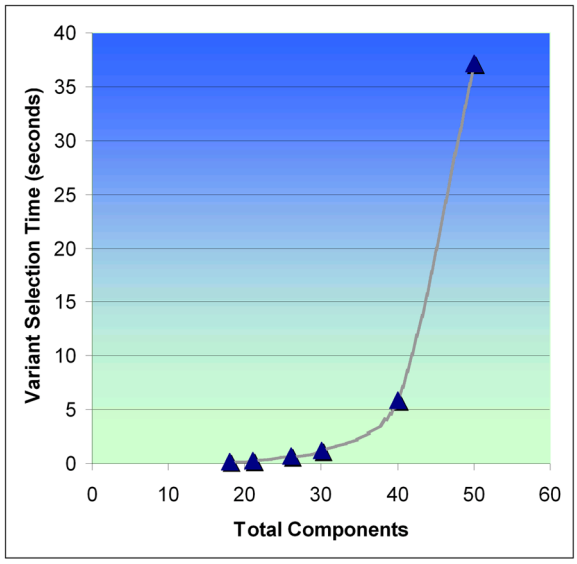

Figure 6: Scatter Performance on Pure Resource Constraints

to select a variant from 40 to 50 *Components*, solving for a variant does not scale well if resource constraints alone are considered.

5.2 Testing the Effect of Limited Resources

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

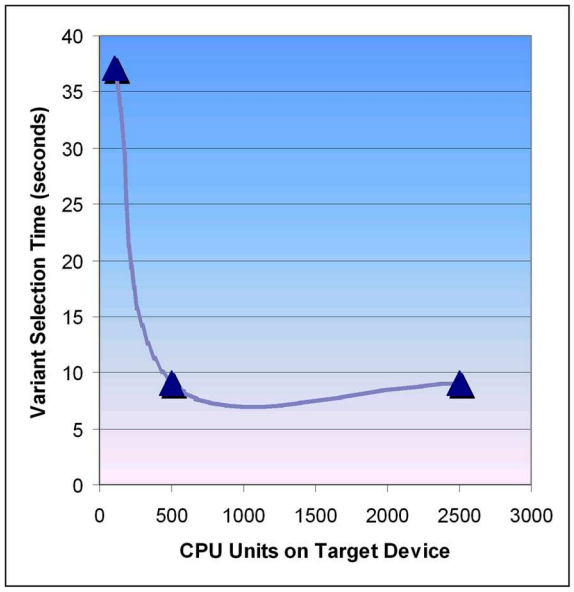

Figure 7: Scatter Performance as CPU Resources Expand on Device

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

We then proceeded to increase the memory on the device while keeping 2,500 units of CPU. The results are shown in Figure 8. Doubling the memory immediately halved the solution time. Doubling the memory again to 128 megabytes provided little benefit since the initial doubling to 64 megabytes made deployment of all of the components possible. As we had hypothesized initially, the solution speed when pure resource constraints are considered is highly dependent on how tight the resource constraints are.

5.3 Testing the Effect of PLA Composition Constraints

Our next set of experiments evaluated how well the dependency constraints within a PLA could filter the solution

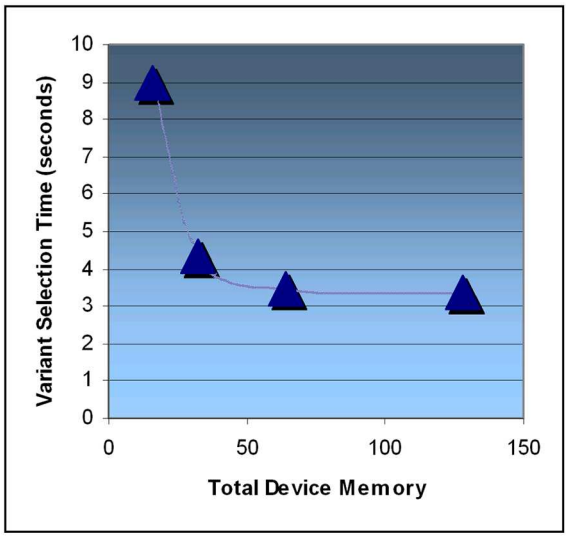

Figure 8: Scatter Performance as Memory Resources Expand on Device

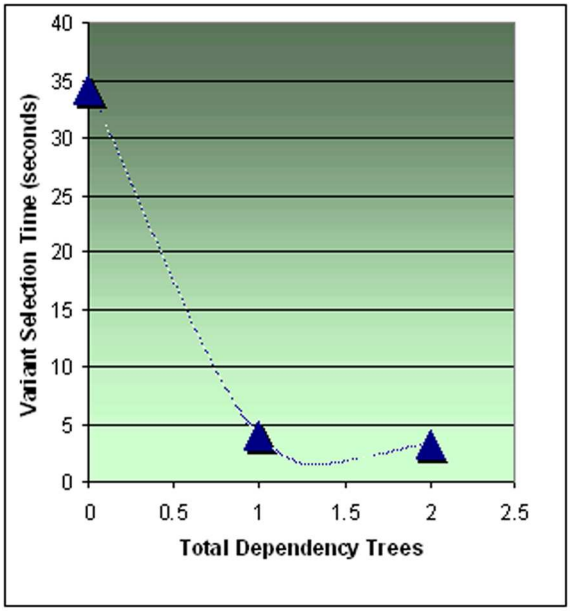

Figure 9: Scatter Performance as PLA Dependency Trees are Introduced

space and reduce solution time. We modified our models so that the *Components* composed sets of applications that should be deployed together. For example, our *TrainTicketReservationService* was paired with the *TrainScheduleService* and other complementary components.

As with the first experiment 5.1, we used our 50 component model as the initial baseline. We first constructed a tree of dependencies that tied 10 components into an application set that led the root of the tree, the reservation service, to only be deployed if all children where deployed. Each level in the tree depended on the deployment of the layer beneath it. The max depth of the tree was 5. We continued to create new dependencies between the components to produce trees and see the effect. The results are shown in Figure 9.

As can be seen from the results in Figure 9, by adding dependencies between components and creating a dependency tree, there was an immediate drop in selection time. This is presumably because it reduces the number of possible combinations of the components that must be considered for a variant. Adding more dependencies to the model to add other trees provided only a very small gain over the original large performance increase.

5.4 Results Analysis: Mobile PLA Design Strategies

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

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

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

Many variants may also require common sets of these JAR files with various footprints. These variants can be grouped based on their JAR configurations. For each group, a non-functional requirement can be added to the components to ensure that a target Device provide sufficient disk space or communication bandwidth to receive the JARs. For small devices that usually have little availabe disk space,

where resource constraints are tighter and solving takes more time, large numbers of Components can be pruned solely due to the lack of packaging variability and need for disk space. This footprint-based strategy works even if there are few functional PLA dependencies between components.

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

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

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

6 Related Work

In [15], Mannion et al present a method for specifying PLA compositional requirements using first-order logic. The validity of a variant can then be checked by determining if a PLA satisfies a logical statement. Although Scatter's approach to PLA composition also checks variant validity, it extends the work in [15] by including the evaluation of non-functional requirements not related to composition. In particular, Scatter automates the variant selection process using these boolean expressions and augments the selection process to take into account resource constraints, as well as optimization criteria. Although the idea of automated theorem proving is enhanced in [16], this approach does not provide a requirements-driven optimal variant selection engine like Scatter. Further differences between Scatter's constraint-based and Mannion's logic-based approaches is available in [3].

A mapping from feature selection to a CSP is provided by Benavides et al. [3]. Scatter uses this same reduction but also extends it with the capability to handle references and resource constraints. Resource constraints are a key requirement type in mobile devices with limited capabilities. Moreover, the approach presented by Benavides does not show how this constraint-based mechanism could utilize a mobile device discovery service as Scatter does. Finally, Benavides et al. do not address how PLA design decisions can be used to improve constraint solver performance as this paper does.

In [14], Lemlouma et. al, present a framework for adapting and customizing content before delivering it to a mobile device. Their strategy takes into account device preferences and capabilities, as does Scatter. The approaches are comparable in that each attempts to deliver customized data to a device that handles its capabilities and preferences. Resource constraints are a key difference that makes the selection of software for a device more challenging than adapting content. Unlike [14], Scatter not only provides adaptation for a device, but also optimizes adaptation of the software with respect to its provided PLA cost function.

Many complex modeling tools are available for describing and solving combinatorial constraint problems, such as those presented in [18, 7, 23, 4, 11]. These modeling tools provide mechanisms for describing domain-constraints, a set of knowledge, and finding solutions to the constraints. These tools, however, do not provide a high-level mechanism to capture non-functional requirements and PLA composition rules geared towards mobile devices. These tools also do not provide a mechanism for incorporating data from a device discovery service. Finally, these papers have not addressed how PLA design decisions influence variant selection speed.

7 Concluding Remarks

Online PLA variant selection for mobile devices is a challenging domain that can benefit from automation since

there are too many complexities and unknown device characteristics to manually account for all possibilities ahead of time. Constraint-solver based automation is a promising technique for online variant selection. This paper describes how our Scatter tool supports efficient online variant selection. Moreover, by carefully evaluating and constructing a PLA selection model based on the rules we presented, developers can alleviate the effects of worst-case solver behavior.

From our experience developing and evaluating Scatter, we learned the following lessons:

- PLA composition and non-functional requirements can be used to efficiently prune the variant selection space and provide good performance. There are many patterns of requirements specification that can be used to optimize a PLA for automated variant selection. In future work, we intend to further explore these patterns.

- Although Scatter can automate variant selection, it works best when a PLA is crafted with performance in mind. An arbitrary PLA may or may not allow for rapid variant selection. PLA's that will be used in conjunction with an automated variant selector should be carefully constructed to avoid poor performance.

- A key challenge of automating product variant selection is debugging mistakes in the product-line's specification. A simple mistake, such as a misplaced exclusion constraint between components, can cause variant selection to fail. Moreover, the failure may only appear intermittently for certain device types and be hard to identify during testing. Even once it is discerned that there is a problem, identifying the source of the problem can be extremely challenging (we have experienced this phenomenon).

- More work must be done to understand how to merge and integrate the various information sources that will provide device characterizations. Device characterizations may come from customer databases, discovery services, and location services. Finding the right transformations to correlate and utilize these diverse information streams is important to provide customized and correct variant selection.

- Developers normally focus on the functional variability in a product. Looking at other aspects of variability, such as packaging variability, is important too. As we have shown, although a product may have high functional variability, it can be significantly less variable with respect to packaging or memory footprint. These non-functional aspects can be exploited to reduce the complexity of automated variant selection.

In future work, we plan to integrate and test various discovery mechanisms and resource, context, and device characterization schemas to see how Scatter performs. We also plan to extend Scatter to interface with various types of runtime deployment middleware infrastructure.

References

[1] Choco constraint programming system. http://choco.sourceforge.net/.

[2] M. Anastasopoulos. Software Product Lines for Pervasive Computing. *IESE-Report No. 044.04/E version*, 1.

[3] D. Benavides, P. Trinidad, and A. Ruiz-Cortes. Automated Reasoning on Feature Models. *17th Conference on Advanced Information Systems Engineering (CAiSEŠ05, Proceedings), LNCS*, 3520:491–503, 2005.

[4] Y. Caseau, F.-X. Josset, and F. Laburthe. CLAIRE: Combining Sets, Search And Rules To Better Express Algorithms. *Theory and Practice of Logic Programming*, 2:2002, 2004.

[5] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, 2002.

[6] E. Coffman Jr, G. Galambos, S. Martello, and D. Vigo. Bin packing approximation algorithms: combinatorial analysis. *Handbook of Combinatorial Optimization. Kluwer Academic Publishers*, 1998.

[7] J. Cohen. Constraint Logic Programming Languages. *Commun. ACM*, 33(7):52–68, 1990.

[8] J. Coplien, D. Hoffman, and D. Weiss. Commonality and Variability in Software Engineering. *IEEE Software*, 15:37–45, Nov.-Dec. 1998.

[9] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration through specialization and multi-level configuration of feature models. *Software Process Improvement and Practice*, 10(2):143–169, 2005.

[10] D. P. D. Oppenheimer, A. Ganapathi. Why do Internet Services Fail, and What can be Done about It? *Proceedings of the USENIX Symposium on Internet Techn̆nologies and Systems*, March 2003.

[11] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, November 2002.

[12] J. Jaffar and M. Maher. Constraint Logic Programming: A Survey. *constraints*, 2(2):0, 1994.

[13] K. Kang et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Carnegie Mellon University, Software Engineering Institute, 1990.

[14] T. Lemlouma and N. Layaida. Context-aware Adaptation for Mobile Devices. *Mobile Data Management, 2004. Proceedings. 2004 IEEE International Conference on*, pages 106–111, 2004.

[15] M. Mannion. Using First-order Logic for Product Line Model Validation. *Proceedings of the Second International Conference on Software Product Lines*, 2379:176–187, 2002.

[16] M. Mannion and J. Camara. Theorem Proving for Product Line Model Verification. *Fifth International Workshop on Product Family Engineering, PFE-5, Siena*, pages 4–6, 2003.

[17] T. Männistö, T. Soininen, and R. Sulonen. Product Configuration View to Software Product Families. *10th International Workshop on Software Configuration Management (SCM-10), Toronto, Canada*, pages 14–15, 2001.

[18] L. Michel and P. V. Hentenryck. Comet in Context. In *PCK50: Proceedings of the Paris C. Kanellakis memorial workshop on Principles of computing & knowledge*, pages 95–107, New York, NY, USA, 2003. ACM Press.

[19] S. Mittal and F. Frayman. Towards a generic model of configuration tasks. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 2:1395–1401, 1989.

[20] D. Muthig, I. John, M. Anastasopoulos, T. Forster, J. Dörr, and K. Schmid. GoPhone-A Software Product Line in the Mobile Phone Domain. *IESE-Report No*, 25, 2004.

[21] A. Nechypurenko, J. White, E. Wuchner, and D. C. Schmidt. Applying Model Intelligence Frameworks to Deployment Problems in Real-time and Embedded Systems. In *Proceedings of MARTES: Modeling and Analysis of Real-Time and Embedded Systems at the 9th International Conference on Model Driven Engineering Languages and Systems, MoDELS/UML 2006*, 2006.

[22] D. Sabin and R. Weigel. Product configuration frameworks-a survey. *Intelligent Systems and Their Applications, IEEE [see also IEEE Intelligent Systems]*, 13(4):42–49, 1998.

[23] G. Smolka. The Oz Programming Model. In *JELIA '96: Proceedings of the European Workshop on Logics in Artificial Intelligence*, page 251, London, UK, 1996. Springer-Verlag.

[24] T. van der Storm. *Variability and Component Composition*. Springer, 2004.

[25] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press Cambridge, MA, USA, 1989.

[26] J. White, A. Nechypurenko, E. Wuchner, and D. C. Schmidt. Intelligence frameworks for assisting modelers in combinatorically challenging domains. In *Proceedings of the Workshop on Generative Programming and Component Engineering for QoS Provisioning in Distributed Systems at the Fifth International Conference on Generative Programming and Component Engineering (GPCE 2006)*, 2006.

[27] J. White, D. Schmidt, and A. Gokhale. The J3 Process for Building Autonomic Enterprise Java Bean Systems. *icac*, 00:363–364, 2005.

[28] J. White and D. C. Schmidt. Simplifying the Development of Product-Line Customization Tools via MDD. In *Workshop: MDD for Software Product Lines, ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, October 2005.

[29] W. Zhang, S. Jarzabek, N. Loughran, and A. Rashid. Reengineering a PC-based system into the mobile device product line. *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*, pages 149–160, 2003.