Automatically Composing Reusable Software Components 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) are an effective mechanism for facilitating the reuse of software components on different mobile devices. Mobile applications are typically delivered to devices using over-the-air provisioning services that allow a mobile phone to download and install software over a cellular network connection. Current techniques for automating product-line variant selection do not address the unique requirements (such as the need to consider resource constraints) of dynamically selecting a variant for over-the-air provisioning.

This paper presents the following contributions to product-line variant selection for mobile devices: (1) it describes how a constraint solver can be used to dynamically select a product-line variant while adhering to resource constraints, (2) it presents architectures for automatically discovering device capabilities and mapping them to product-line feature models, (3) it includes results from experiments and field tests with an automated variant selector, and (4) it describes PLA design rules that can be used to increase the performance of automated constraint-based variant selection. Our empirical results show that fast automated variant selection from a feature model is possible if certain product-line design guidelines are followed.

Keywords: Methods, Processes, Tools and Experiences in Software Product Lines, Automation

1 Introduction

The increasing popularity and abundance of mobile and embedded devices is bringing the promise of pervasive computing closer to reality. At the end of 2003, it was estimated that there were over two billion consumer electronic devices, such as mobile phones and television set-top boxes, and 300 million Personal Digital Assistants (PDAs) [33]. Not only is the number of devices increasing, but the number of available mobile applications is growing as well. For example, Google delivers both web-based interfaces to google mail, news, and maps and applications that can be downloaded to mobile phones [2].

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 across a mobile network. Services that allow software to be downloaded over cellular networks are called Over The Air Provisioning (OTAP) services [19, 28, 4, 5]. For example, mobile phones can now access web-based applications, such as google mail, or download custom applications from services, such as Verizon's "Get It Now." Nokia estimates that in 2003, 10,000,000 Java 2 Micro Edition (J2ME) games *per month* were downloaded worldwide [31].

In pervasive computing environments, the ability to download software on-demand plays 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.

Despite the advances in middleware and deployment technologies, however, there are still significant variabilities between devices in terms of hardware resources (such as CPU power, RAM, and display size), middleware versions (such as Java Virtual Machine versions), hardware capabilities (such as Bluetooth support), and service provider restrictions (such as required use of provider-specific APIs). Developing software that can handle all of these diverse restrictions and be deployed on a large number of heterogeneous devices is hard [6]. In some cases, due to large differences in non-functional device properties like display size, separate variants of the same Java application must be developed for each device despite the presence of a virtual machine [3].

Product-line architectures (PLAs) [13] are a promising approach to help developers reduce the high cost of mobile application development by facilitating software reuse [7, 40, 30]. A product-line architecture (PLA) [13] leverages a set of reusable software components that can be composed in different configurations (variants) for different requirement sets. Constructing a product-line variant consists of finding a way of reusing and composing the product-line's components to create a functional application. The design of a PLA is typically guided by scope, commonality, and variability (SCV) analysis [16]. 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.

A product-line documents the rules that a developer must follow when assembling existing reusable software components into an application for a new mobile device. It hard to manually retarget mobile applications using product-line components, however, due to the large number of mobile devices, limited device capabilities, complex product-line constraints, and the rapid development rate of new devices. Moreover, in a pervasive environment, software reuse must happen on-demand. When a device enters a particular context, such as a retail store, the provisioning server must very quickly deduce and create a variant for the device, regardless of whether or not the device type and its capabilities have been previously encountered.

Current automated software reuse techniques, such as those presented in [9, 25, 29, 32, 35], 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 approach 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 64 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. Reusing software components on mobile devices requires careful consideration of both the limited and unique resources of a device and the complex constraints on the mobile application product-line.

Another missing detail of these automatic reuse approaches is the architecture for how an over-the-air provisioning server can characterize a device's functional and non-functional properties (such as OS, total RAM, etc.) so that a variant can be selected for it. A variant selection engine for mobile devices must be able to interface with a provisioning server and map device capabilities to product-line models. Finally, to provide fast feature selection engines (which aids dynamic software delivery for mobile devices), 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 software reuse for mobile devices:

- We describe Scatter's architecture for integrating with an open-source over-the-air provisioning server and capturing device capabilities

- We show how Scatter enables and disables features/components in product-line models based on the sets of device capabilities it receives from the provisioning server

- We discuss how Scatter transforms requirement specifications into a format that can be manipulated by a constraint solver and how we extend existing constraint-based automation approaches [9] to include resource constraints

- We describe the automated variant selection engine, based on a Constraint Logic Programming Finite Domain (CLP(FD)) solver [21, 36], that can dynamically derive a valid configuration of reusable software components suitable for a target device's capabilities and resource constraints

- We present data from experiments that show how PLA constraints impact variant selection time for a constraint-based variant selection engine

- We present results from field tests using both real and emulated devices to obtain applications from the Scatter-integrated provisioning server

- We describe PLA design rules gleaned from our experiments that help to improve variant selection time when using a constraint-based software reuse approach.

This paper builds on our previous work on software reuse that involved automatically deriving product-variants for mobile devices with a constraint solver [38]. In particular, this paper enhances previous work by describing the design and functionality of a Scatter-integrated server for performing over-the-air provisioning of mobile devices. We also offer new empirical results obtained from field testing the Scatter-integrated provisioning server with both real and emulated mobile devices. The new results show that despite the apparent complexity of product-line composition rules and non-functional requirements, a constraint solver can be used to derive a product variant quickly enough to support over-the-air provisioning.

The remainder of this paper is organized as follows: Section 2 presents the train food services application that we use as an example product-line throughout the paper; Section 3 describes the challenges of dynamically composing reusable software components for different mobile devices and the unresolved problems of using current techniques; Section 4 presents architectures for integrating an automated variant selection mechanism into an over-the-air provisioning server; Section 5 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 6 analyzes the results of field tests and simulations of using Scatter for over-the-air provisioning; Section 7 summarizes product-line design rules that we have learned from our results that improve the speed at which a product variant can be selected; Section 8 compares our work on Scatter with related research; and Section 9 presents lessons learned and concluding remarks.

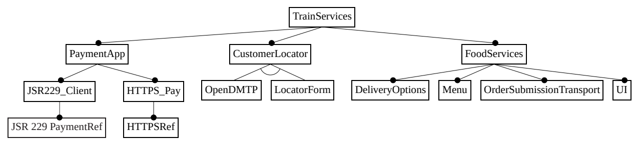


Figure 1: Feature Model for Train Services Applications

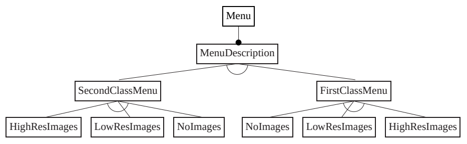


Figure 2: Food Services Menu Feature Model

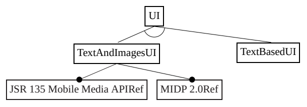


Figure 3: Food Services UI Feature Model

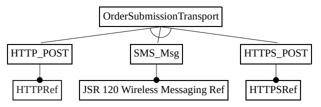


Figure 4: Food Services Order submission Feature Model

2 Motivating Example

To motivate the need for—and capabilities of—Scatter, we use an application throughout this paper that allows train passengers to order food from their mobile phones. This application is downloaded by passengers to their phones upon entering a train. The application allows passengers to choose menu items from either a first class or second class menu (depending on the traveler's ticket class).

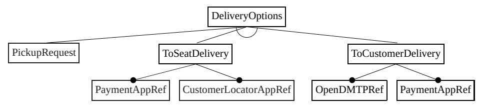


Figure 5: Food Services Delivery Options Feature Model

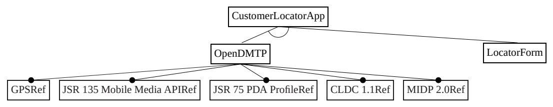


Figure 6: Customer Locator Application Feature Model

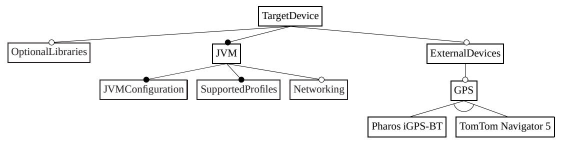


Figure 7: Target Device Feature Model

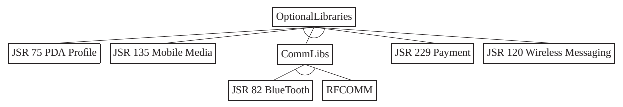


Figure 8: Java Optional Libraries Feature Model

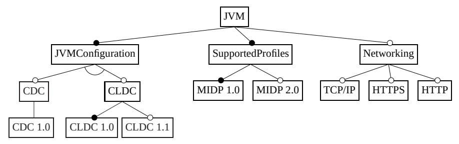


Figure 9: Java Virtual Machine Feature Model

The food services product-line has been described using feature models. Feature modeling [8, 17] characterizes and application based on function and non-functional variabilities. The feature models are designed to show the composition rules for the variable application components and how device capabilities affect what application components can be deployed.

The food services application is implemented using a variety of components, such as the Open Device Monitoring and Tracking Protocol (OpenDMTP) Java MIDlet[1]. This application can be reconfigured for devices that support different Java JVM Mobile Information Device Profile (MIDP) versions, JVM configurations (*e.g.* CDC 1.0, CLDC 1.0, and CLDC 1.1), and optional Java APIs (*e.g.* JSR 135 Mobile Media API, JSR 229 Payment API, etc.). Figures 1 through 6 show feature models capturing the SCV of the

[1]A Java application for an embedded Java 2 MicroEdition JVM.

food service application. Figures 7 through 9, show the key points of variability in the target devices that determine which food services application components are chosen when selecting a variant for a mobile device. For example, if the *TextAndImagesUI* feature from the feature model in Figure 3 is chosen, the target device must have the *JSR 135 Mobile Media API* feature (Figure 8) enabled.

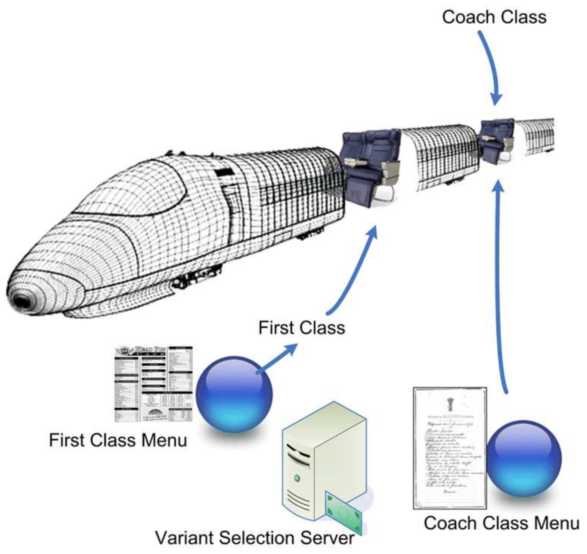


Figure 10: Alternate Variant Selection Based on Cabin Class

Context data also determines which application components can be delivered to a device, as seen in Figure 10. Second class passengers can pre-order food from a second class menu from their mobile devices but must go to the restaurant car of the train to pickup the food. First class passengers, however, order from a more extensive first class menu and can have the food delivered to either their seat or directly to their current location on the train. The food services application uses the OpenDMTP Java client implementation to report the location of a first class passenger with a Global Positioning System (GPS) capable phone. If a first class passenger does not have a phone with a connected GPS device, an application variant is delivered to the device that replaces the OpenDMTP tracking MIDlet with a form for the user to enter their current seat number.

Finally, non-functional characteristics of the device dictate certain key features of the selected variant. The food services application can be delivered with either high resolution images of the entrees (requires 64 kilobytes of storage space), low resolution images (12 kilobytes), or no entree images (0 kilobytes). The available memory and storage space on the device determines which of these image sets is appropriate. The OpenDMTP client is the largest of the other application components and requires approximately 2 kilobytes of storage space. The remaining application components consume another $\sim$2 kilobytes. The total combined resource consumption of all of the application components must be considered when choosing image sets.

For a phone with at least 66 kilobytes of remaining storage space, a number of variants are possible. If the owner of the device has a first class ticket and a GPS capable phone, a variant with the OpenDMTP library and low resolution images is suitable. If the user does not have a first class ticket or a GPS capable phone, then the high resolution images may fit. To choose an appropriate variant, therefore, the variant selection must account for the tradeoffs in resource consumption of different configurations.

3 Challenges of Automated Variant Selection for Mobile Devices

Applications for mobile devices must be carefully matched to the capabilities of each individual device due to resource constraints. Developers must therefore consider both functional capabilities (such as the optional libraries installed on the device) and non-functional capabilities (such as total memory) when reusing software components. Due to the large and highly differing array of device capabilities, however, it is difficult to determine which software components can function with each device's unique limitations and how an entire application can be assembled by reusing these viable components. For example, reusing product-line components for a mobile device involves:

1. Capturing the rules for composing the reusable product-line components or features (the application model)
2. Specifying what capabilities the target mobile device must have to support each application component or feature (the target infrastructure model)
3. Identifying the target mobile device and mapping its capabilities onto the target infrastructure model by enabling or disabling features in the model
4. Disabling application components that cannot be supported by the functional and non-functional capabilities of the device
5. Selecting and assembling a product variant from the remaining enabled components and features that adheres to the product-line's composition rules and the resource constraints of the device.

For example, with the food services application presented in Section 2, the rules for composing the application's components rules were first documented in the feature models presented in Figures 1 through 6. Next, the important features of the target infrastructure that govern which application components can be supported by a device were documented in Figures 7 through 9.

The dependencies between the application components and target device capabilities were specified with feature references. For example, the *SMS_Msg* component (Figure 4) for submitting orders contains a reference to the target infrastructure feature *JSR 120 Wireless Messaging* (Figure 8). This reference indicates that the JSR 120 Wireless Messaging feature must be enabled on the target device if the SMS_Msg component will be deployed to it.

To find a way of reusing existing software components to assemble a variant of the food services application for a Blackberry Pearl 8100 mobile phone, a developer would enable and disable the appropriate features in the target device feature model (Figures 7 through 9). The *CDC* feature of the *JVM Configuration* and the GPS features would be disabled while the *CLDC 1.1* feature would be enabled (the Blackberry 8100 supports MIDP 2.0, CLDC 1.1, and no GPS). Since the Blackberry 8100 does not support GPS in its stock configuration, this would preclude deploying the OpenDMTP feature to the phone and thus it would be disabled. Finally, an appropriate set of features, would be selected from the remaining points of variability (*e.g.*, TextUI or TextAndImagesUI, SMS_Msg order submission or HTTPS_Post, etc.).

Traditional processes of reusing software components involve developers manually evaluating a mobile device and determining the software components that must be in an application 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 thus a significant source of system downtime [18]. Manual reuse approaches also do not scale well and become impractical with the large solution spaces typical of PLAs.

Numerous tools and techniques are available that have helped improve software reuse by automating various portions of variant selection and assembly. Although these techniques are extremely valuable for product-line specification, variant selection, and variant assembly, they do not specifically address all of the issues related to online selection of variants. The remainder of this section describes the key challenges that have not been addressed when applying current approaches to dynamic variant selection for over-the-air mobile software provisioning.

1. There is no clear architecture for automatically discovering and mapping device capabilities to product-line models. Numerous tools and approaches have been developed [10, 11, 8] to capture the rules for composing a product variant. For example, *P*ure::variants [10] is a commercial tool that provides feature modeling capabilities, allows developers to specify features and feature constraints, and derives required unconfigured features for a partially configured variant. All these tools, however, are designed for *a priori* product variant selection and assume that a human modeler enables/disables features and uses the tool to derive any required additional features. To select a variant for a mobile device, therefore, developers must manually enable/disable model features to reflect the capabilities of a target device.

An over-the-air provisioning request begins by a mobile device sending a request to a provisioning server that includes a unique identifier for the device type, as seen in Figure 11. From this unique identifier, the provisioning

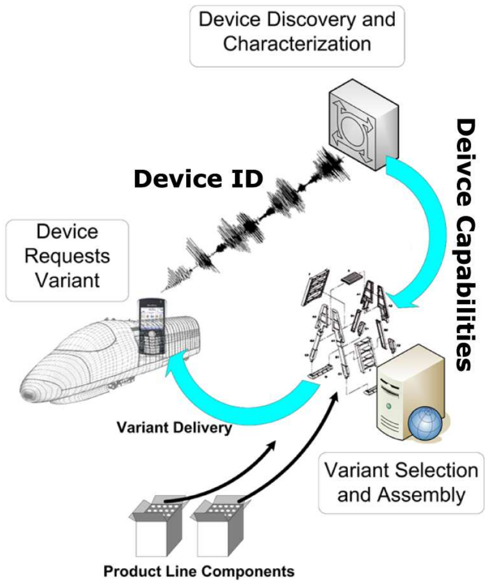


Figure 11: Selecting a Food Services Variant for a Blackberry 8100 Mobile Phone

server must be able to find the capabilities associated with the device and automatically map these capabilities into the model of the target infrastructure. Existing tools do not address how a human is removed from the modeling loop and a single device identifier is mapped into a complex set of infrastructure model capabilities. In Section 4, we present three different architectures that can be used to automatically discover device capabilities and map them to product-line models.

2. There is no documented architecture for handling incomplete context information and unknown device types. Many research efforts [29, 9, 26] have produced models for transforming a feature model or other SCV capturing mechanism into a formal model that can be reasoned with automatically. For example, [9] presents a method for transforming feature models into Constraint Satisfaction Problems (CSPs). A solver, such as a Constraint Logic Programming (CLP) solver, can then be used to automatically derive product variants for a set of device capabilities.

The key assumption with these techniques is that values for all relevant device capabilities are known. Although devices may share common communication protocols and resource description schemas, a variant selection service will not know all device signatures at design time. In many cases, information, such as the exact set of optional libraries installed on a device or ticket class of the owner may not be able to be determined based on the unique device identifier associated with the provisioning request. In other situations, a provisioning server may encounter a newly released device with completely unknown capabilities.

To address the more dynamic information needs of PLAs for mobile applications, therefore, either a strategy for selecting a variant with incomplete information or an automated method for obtaining missing capability information is needed. Current research does not address this open problem. Section 4 presents our *on-demand probing* approach that allows a provisioning server to help guarantee it has complete device information when selecting a variant.

3. There is no method for incorporating resource constraints in variant selection. Although multiple models and tools are available [29, 9, 26, 10, 11, 8] for deriving ways of reusing and assembling components for a set of device capabilities, none of these techniques or tools address how resource constraints are considered in the selection process. For mobile devices, resource constraints are a major concern and must be considered carefully. Without a mechanism for adhering to resource constraints, no reliable component selection automation can be performed. For example, deploying a set of components that requires more JVM stack capacity than is available on the target device will result in a non-functioning variant.

Different configurations of reusable components may have different costs associated with them. There may be many valid variants that can be deployed and the selector must possess the ability to choose the best configuration 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. In Section 5, we describe a modified constraint-based variant selection approach that can take resource constraints into account.

4. It is unclear if automated variant selection can be performed fast enough to support on-demand software reuse. Determing which components to reuse and how to assemble them must happen rapidly. For instance, in the train example from Section 2 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.

Many methods and tools [9, 10, 11] for automating variant selection are used for design-time selection of variants. It is still unclear, however, whether the current approaches and tools provide sufficient performance to support dynamic software reuse for over-the-air mobile software provisioning. Design-time selection with a human involves processing a single request at a time. An over-the-air provisioning server could potentially receive hundreds, thousands, or more simultaneous requests. Empirical evaluation is needed to determine if current automation techniques are sufficiently fast in practice. Section 6 presents the results from field and performance tests we performed using automated and constraint-based variant selection.

5. There are no documented design rules for facilitating variant selection automation. Although the tools and related papers cited above cover the basics of building a product-line, they do not systematically capture best design practices to facilitate automation. Many constraint solvers and theorem proving algorithms—particularly ones that incorporate resource constraints—have exponential worst case performance. For developers of product-lines that will leverage an automated variant selector, therefore, it is important to have guidelines for designing a product-line's composition rules to avoid these worst case scenarios and improve automated selection speed. Few—if any—of these types of rules have yet been documented for product-lines. Section 7, describes product-line design rules we derived from our empirical results to help improve the speed at which a variant can be automatically derived using a constraint-based approach.

4 An Architecture for Over-the-air Provisioning from a Product-line

In previous work [38], we developed Scatter, which is a graphical modeling tool for capturing the SCV of a product-line, compiling the product-line rules into a constraint satisfaction problem (CSP), and using a constraint solver to derive a valid product variant for a mobile device. This initial research began to address challenges 4 and 5 from Section 3, which involved showing that constraint-based approaches to variant selection provide good performance and deriving PLA design rules to facilitate automation. We found that model-driven development could be used to transform a high-level specification of a product-line, such as a feature model, into a constraint satisfaction problem. We also found that a constraint solver could be given a CSP and a set of device capabilities and derive an optimal variant in a reasonable time-frame.

Our initial results, however, also showed that care was needed when designing a product-line to achieve good constraint solving performance. Depending on the constraints governing the product-line, solving performance for a 50 feature model varied from a low of $\sim$1 second to a high of over 30 seconds. We found that several widely applicable rules, such as grouping components into sets based on limitations in packaging variability, could help ensure best-case solving performance.

This paper describes how we extended our prior work on Scatter to integrate it with the open-source JVending project, an implementation of the Java specification 124 for an over-the-air provisioning server [19, 28, 4, 5]. The remainder of this section presents our solutions to each open challenge outlined in Section 3. Section 6 then presents empirical results obtained from testing Scatter and JVending that show a constraint-solver based approach to deriving product-variants for over-the-air provisioning is feasible.

4.1 Obtaining the Device Information Required to Make Reuse Decisions

The first step in determining how to fulfill a provisioning request using existing software components is to characterize the unique capabilities of the requesting mobile device. After these capabilities are known, compatible components can be selected and reused in a product variant. Below, we present three different architectures for dynamically discovering device capabilities and mapping them to product-line models. These architectures can be used to help address Challenge 1 of Section 3, which is that no clear architectures have been developed for integrating an automated variant selector and an over-the-air provisioning server. Each architecture provides a differing level of device information that can be leveraged to select a product-line variant. We also evaluate each architecture in terms of Challenge 2 (incomplete device information) since different architectures provide more complete device information than others.

Over-the-air provisioning is typically initiated by a mobile user dialing a specified mobile number or sending an HTTP request to a provisioning server. In most scenarios, the provisioning request includes an identifier that the server uses to determine the type of device issuing the provisioning request and the requesting device's capabilities. The capabilities of the device are used to help determine what components are compatible with the device and should be used to assemble a variant to fulfill the request. The high-level architecture for issuing a provisioning request and deriving a variant for a mobile device with Scatter is shown in Figure 12.

Once a mobile device has initiated a provisioning request, the device's functional properties (such as optional Java libraries that are installed) and non-functional properties (such as $JVMConfiguration$, *Memory*, and *CabinClass*[2]) must be obtained and mapped to the target infrastructure model of the product-line. In our experience, we found that device capabilities can be returned as a set of name/value pairs. Each reusable component can have an expression associated with it based on these name/value pairs that determines if it can be reused in a particular device. For example, after a set of device capabilities is collected, the JSR 135 feature (Figure 9) can be enabled or disabled based on whether or not the $JSR135$ device capability variable is equal to true. If the JSR 135 feature is disabled, the TextAndImagesUI component will not be considered for reuse.

The values for these variables are typically determined using either a push or pull architecture. With a pull architecture the device sends its unique identifier and the provisioning server queries either a device description repository [39, 24] (a database of device identifiers and their associated capabilities) or the device itself for the capabilities of the device. A push model may also be used where the mobile device sends its device type information and capabilities to the server as part of its provisioning request. For example, if a user is presented with a set of HTML links to variants for a Java MIDP 1.0/CLDC 1.0 phone or an MIDP 2.0/CLDC 1.1 phone, when the user clicks on a specific link, the device is sending a request that is pushing the needed device capability information.

We next describe the push and pull models in more detail and show how neither is ideally suited for obtaining the information required for deriving a configuration of reusable software components for a product variant. We then present an alternative approach, called *on-demand probing*, that attempts to address the limitations of the push and pull models. Scatter uses this on-demand probing approach to gather missing device capability information and ensure that all needed capability values are known when reusable components are selected and assembled for a device.

[2] *CabinClass* is a boolean indicating if the traveler has a first or second class ticket. Although it would be possible to model *CabinClass* outside the target infrastructure model, we include it there for simplicity.

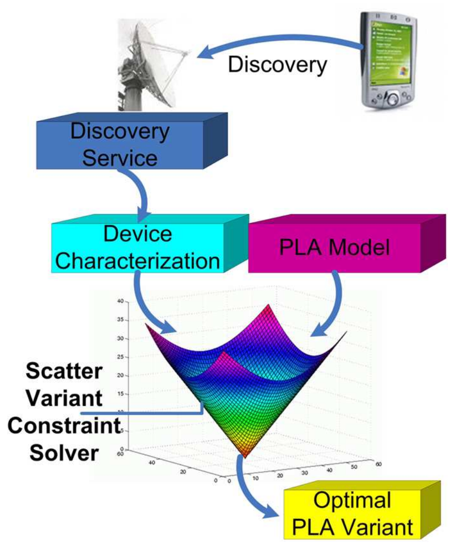


Figure 12: Scatter Integration with a Discovery Service

4.1.1 Pull Models for Discovering Device Capabilities

A pull model extracts device capabilities from a device description repository and can provide detailed information with regard to static device capabilities ranging from supported APIs to hardware specifications. A mobile device may not be able to introspectively determine all of the information available in a device description repository nor may it be efficient to send this large amount of data across a cellular network. Pull models are also desirable since they place the burden of the work on the server and decouple the device from the capability discovery mechanism. Moreover, a pull model does not require error-prone user-interaction.

Numerous open-source and commercial projects are available that offer databases of device capabilities. With a pull model, the provisioning server's main task is to identify the identifier for the type of device issuing the request and then query the appropriate device description repository for its capabilities. Although having a large database of device capabilities may appear to make it possible to build variants for devices ahead of time, a device description repository only contains *static* capability information and cannot leverage context (*e.g.* CabinClass) or dynamic information (*e.g.* remaining storage space) about a device. The database packaged with the open-source Wireless Universal Resource File (WURFL) [24] project contains 5,038 unique device signatures.

A diagram of a request for a MIDP application (MIDlet) product variant is shown in Figure 13. Initially, the device sends an HTTP request to the provisioning server for the MIDlet and includes the device's *UserAgent*, an identifier of the requesting device type or browser type, in the request headers. The provisioning server uses the User-Agent name to query a device description repository and identify the device's capabilities. Once the device's unique signature is known, Scatter is executed to determine the appropriate product variant to fulfill the provisioning request.

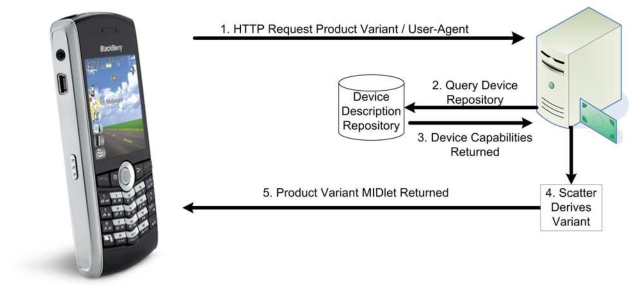

1. HTTP Request Product Variant / User-Agent
Device Description Repository
2. Query Device Repository
3. Device Capabilities Returned
4. Scatter Derives Variant
5. Product Variant MIDlet Returned

Figure 13: An HTTP Provisioning Request for a J2ME Midlet Product Variant

The key disadvantage of pull models is that they limit the information that can be used to guide variant construction since they rely on pre-compiled device information databases. New devices are released frequently and thus a repository may not know the capabilities of the latest products. Pre-compiled databases also cannot use dynamic information, such as CabinClass, specific to an individual user's device. In situations where not all required device information is available, the variant selection process faces Challenge 2 of Section 3, which involves handling missing capability information.

4.1.2 Push Models for Discovering Device Capabilities

Push models offer an apparent solution to the deficiencies of pull models. With a push model, the mobile device encodes all required capabilities and context information for deriving a product variant into its provisioning request. This architecture avoids Challenge 2 from 3 by ensuring that all needed device information is submitted with the request. For example, a device can issue an HTTP request with request parameters for the device memory, JVM stack size, display dimensions, JVM profiles/configurations, and a list of available optional Java libraries.

A push model can also incorporate context-dependent data. For example, a user can be presented with an HTML form to capture the traveler's ticket number. The form can then be sent to the provisioning server via an HTTP POST and the server can obtain the device user's cabin-class, seat assignment, name, and other reservation attributes before invoking Scatter and deriving a variant. This form-based architecture is shown in Figure 14.

The push model, however, has its own drawbacks. First, the push model relies on the user to supply critical infor-

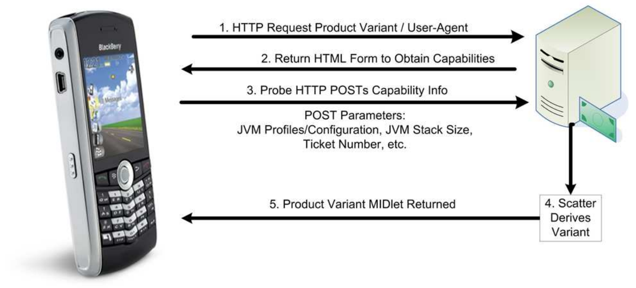

1. HTTP Request Product Variant / User-Agent
2. Return HTML Form to Obtain Capabilities
3. Probe HTTP POSTs Capability Info
POST Parameters:
JVM Profiles/Configuration, JVM Stack Size,
Ticket Number, etc.
4. Scatter Derives Variant
5. Product Variant MIDlet Returned

Figure 14: An HTTP Provisioning Request with a Push Model for a J2ME Midlet Product Variant

mation that is used to select a product variant. A user can easily make mistakes (*e.g.* provide the wrong CLDC version) and cause incorrect software variants to be delivered to the device. Users may not know all of the required platform information, such as JVM stack size, required by the provisioning server. The push model also requires sending device capabilities, such as CPU megahertz, across the network even though they do not vary across a particular device model.

4.1.3 On-demand Probing: A Hybrid Capability Discovery Model

Integrating Scatter with a provisioning server created the unique challenge that the device information required to perform variant selection could vary depending on the constraints of the product-line. For example, for some products, a pull model is appropriate since the product-line constraints only depend on device capabilities that do not vary across a model. For other product-lines, such as the train food service application, context information, such as cabin-class, is needed, motivating a push model.

The Scatter integration needed to support context information that would not be available with a pull model. Since selecting product variants using partial information is not a well-understood area of research, we decided our solution had to ensure that all required device information was available. Instead of opting for a push model and requiring error-prone interaction with the user to obtain all required capabilities, Scatter's integration with JVending uses a hybrid push/pull model, which we call *on-demand probing*.

On-demand probing uses a device description repository to obtain static capabilities. If a product-line includes constraints on capabilities that are unavailable from the repository, Scatter returns a small MIDlet to the device. The MIDlet programmatically probes the user's device for the missing capability information and may also prompt the user for context information (*e.g* ticket number). After obtaining the needed capabilities the probe sends the information back to the server to obtain the originally requested product variant. This on-demand probing architecture is shown in Figure 15.

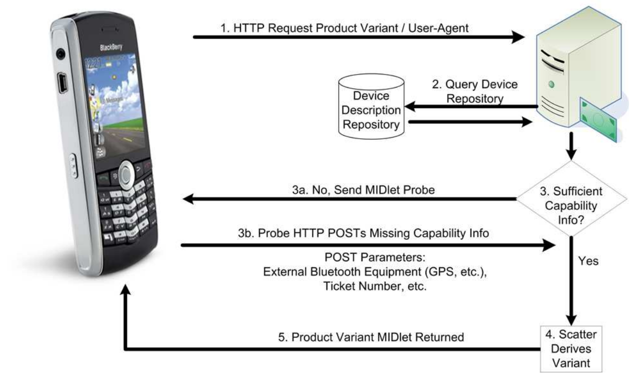

1. HTTP Request Product Variant / User-Agent
Device Description Repository
2. Query Device Repository
3a. No, Send MIDlet Probe
3. Sufficient Capability Info?
3b. Probe HTTP POSTs Missing Capability Info
POST Parameters:
External Bluetooth Equipment (GPS, etc.),
Ticket Number, etc.
Yes
4. Scatter Derives Variant
5. Product Variant MIDlet Returned

Figure 15: An HTTP Provisioning Request with a On-demand Probing Model for a J2ME Midlet Product Variant

On-demand probing combines the best attributes of both the push and pull models. When only static device capabilities are needed by the product-line constraints, on-demand probing obtains the required information from a device description repository. When context or other information that is unavailable in the repository is needed, Scatter addresses Challenge 2 by reverting to a push model. To help reduce user interaction and improve the reliability of the capability information received through a push, Scatter delivers a small executable probe to the device to obtain missing capability information.

When a new device is encountered, a probe can programmatically determine display size, JVM configuration/profile, and other information through Java APIs. This information is gleaned programmatically and can be cached for future encounters with the same device type. For context-specific information, the same probe can prompt the user for reservation numbers and other required attributes. The on-demand probing approach minimizes human interaction and can obtain dynamic context information for product variant derivation.

5 Scatter's Resource-aware Variant Selection Engine

Finding a way to configure and reuse existing software components on an arbitrary mobile device is hard. The complex requirements and composition constraints of the product-line must be used to derive a component configuration that will function properly on the limited resources of the device. Developer may therefore need to consider a combination of context, resource, software dependency, UI, and cost constraints when selecting which components to reuse and how to configure them.

It is particularly important to respect resource constraints when reusing software components on different mobile devices. As discussed in Section 3, current approaches do not account for resource constraints when deriving a product variant. Likewise, they also do not provide optimization mechanisms to selectively reuse components that consume less bandwidth and hence incur smaller cellular air time charges. To address this deficiency, this section describes how we extended the CSP approach presented in [9] to include both resource constraints and a simple variant cost optimization.

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 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 only variants that can run on the target infrastructure are considered. The resource constraints are a form of the *knapsack problem* an NP-Hard problem [14]. Scatter's layered pruning helps improve selection speed and enables more efficient solving. As shown in the Section 6, this layered pruning can significantly improve variant selection performance.

5.1 Layered Solution Space Pruning

Initially, the variant solution space may contain many millions or more possible component or feature compositions. Solving the resource constraints is thus time consuming since it is a higly combinatorial problem. To optimize this search, Scatter first prunes the solution space by eliminating components that cannot be reused on 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 reused because one of their dependencies has been pruned in the previous step. This layered pruning process is shown in Figure 16

After pruning the solution space using the PLA composition rules, Scatter considers resource requirements. After solving the resource constraints, Scatter is left with a drastically reduced number of reusable component configurations 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 Scatter's solution space pruning use a constraint solver based on Prolog inferencing. A rule is specified that only allows a component to be reused on a device, if for every local non-functional requirement on the component, a capability is present that satisfies the requirement. For example, if a component requires a JVMVersion

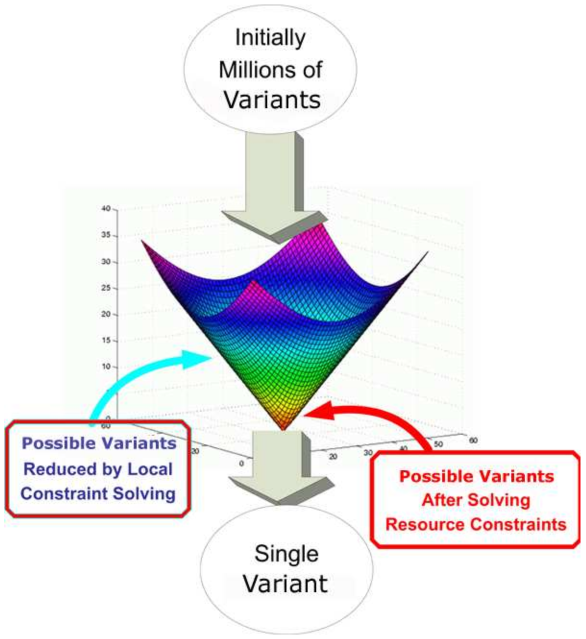

Initially Millions of Variants
Possible Variants Reduced by Local Constraint Solving
Possible Variants After Solving Resource Constraints
Single Variant

Figure 16: Scatter's Layered Deployment Solving Approach

greater than 1.2, the target device must contain a capability named JVMVersion with a value greater that 1.2 or the component is pruned from the solution space and not considered.

The simple Prolog rules for performing this pruning are listed below:

```
comparevalue(V1,V2,'>') :- V1 > V2.
comparevalue(V1,V2,'<') :- V1 < V2.
comparevalue(V1,V2,'=') :- V1 == V2.
comparevalue(V1,V2,'-') :- V1 >= V2.

matchesResource(Req,Resources) :-
     member(Res,Resources),
     self_name(Req,RName),
     self_name(Res,RName),
     self_resourcetype(Req,Type),
     self_value(Req,Rqv),
     self_value(Res,Rsv),
     comparevalue(Rsv,Rqv,Type).

canReuseOn(Componentid,Device) :-
     self_type(Componentid,component),
     self_type(Device,node),
     self_requires(Componentid,Requirements),
     self_dependencies(Componentid,Depends),
     self_provides(Device,Resources),
     forall(member(Req,Requirements),
            matchesResource(Req,Resources)),
    forall(member(D,Depends),canReuseOn(D,Device)).
```

For each component, the rule 'canReuseOn' is invoked to determine reuse feasibility. This rule also simultaneously tests the feasibility of reusing a component based on its dependencies. The last invocation in the rule checks to ensure that all of the components that the current component depends on can also be reused on the device. If any of the dependencies cannot be reused, the component cannot be reused. The rule also throws out components with a resource requirement exceeding what is available on the device, which helps to eliminate the size of the search space for the resource solver.

5.2 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 [9] 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, given the constraint "$X < Y$" then $X = 3$ and $Y = 4$ is a correct labeling of the values for X and Y. The more variables and constraints that are involved in a CSP, the more complex it typically is to find a correct labeling of the variables.

Selecting 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 $DC_0 \ldots DC_n$ that adheres to the composition rules of the feature model.

Resource consumption constraints are created by ensuring that the sum of the resource demands of a binary string representing a variant do not exceed any resource bound on the device (e.g., $\sum variant\_component\_resource\_demands < device\_resources$). For each *Component* C_i that is deployable in the 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 presence variables and summed cannot exceed the available amount of that resource, $Dvc(R)$, on the *Device*.

If the presence variable is assigned 0 (which indicates 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 $Min/Max = 1$.

The Scatter solver also 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.

After the product-line rules have been translated into CLP(FD) constraints, Scatter asks the CLP solver for a labeling of the variables that maximizes or minimizes the variable $Cost(V)$. This approach allows Scatter's variant selector to choose components that not only adhere to the compositional and resource constraints but that maximize the value of the variant. Users therefore supply a fitness criteria for selecting the best variant from the population of valid solutions.

6 Scatter Performance Results

A key question discussed in Challenge 4 of Section 3 is whether or not automated techniques for dynamically composing and reusing software components are fast enough to support over-the-air provisioning of mobile devices. To determine the feasibility of timely on-demand software reuse using a constraint solver, we devised the following series of tests of the Scatter-integrated over-the-air provisioning server:

- **Synthetic experiments**, which are simulated product-line models and device configurations designed to test specific scenarios for variant selection and product-line design hypotheses.

- **Field and stress tests**, which use actual J2ME application requirements, device identifiers, device capabilities, and HTTP provisioning requests to determine how fast variants can be derived in a realistic provisioning scenario.

6.1 Synthetic Variant Selection Experiments

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 tests were performed on an IBM T43 laptop, with a 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 [14]. 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.

6.2 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 17. This figure shows a large jump for the time to select a variant from 40 to 50 components, which indicates that solving for a variant does not scale well if resource constraints alone are considered.

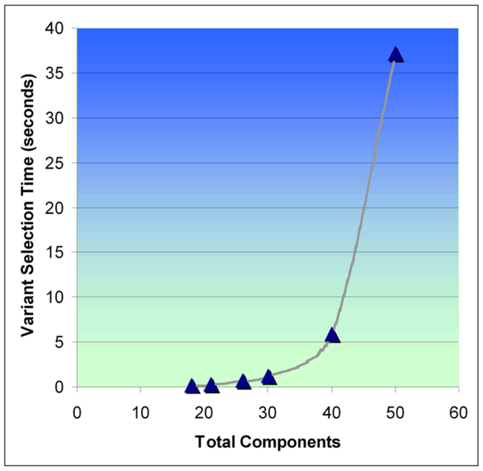


Figure 17: Scatter Performance on Pure Resource Constraints

6.3 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. We chose the 50 component model since it yielded the worst performance from Scatter. The results can be seen in Figure 18. As

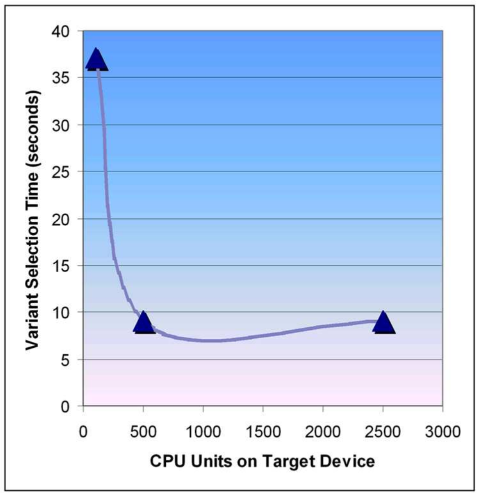


Figure 18: Scatter Performance as CPU Resources Expand on Device

shown in Figure 18, expanding the CPU units from 100 to 500 units dramatically decreased 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 19. Doubling the memory immediately halved the solution time. Doubling the memory again to 128 megabytes provided little benefit since the initial doubling

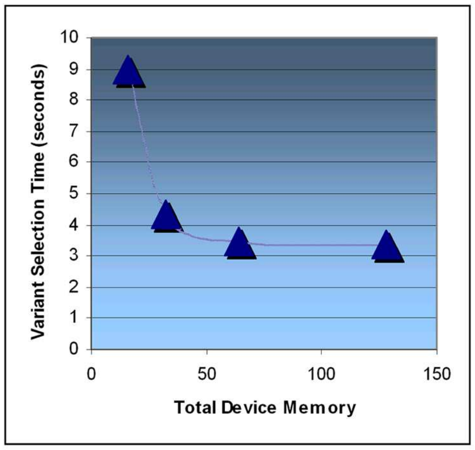


Figure 19: Scatter Performance as Memory Resources Expand on Device

to 64 megabytes deployed all of the components possible. As we hypothesized initially, the solution speed when pure resource constraints are considered is highly dependent on the tightness of the resource constraints.

6.4 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 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 from Section 6.2, 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 train 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 20.

As shown in Figure 20, adding dependencies between components and creating a dependency tree decreased selection time. This decrease occurs because the tree 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.

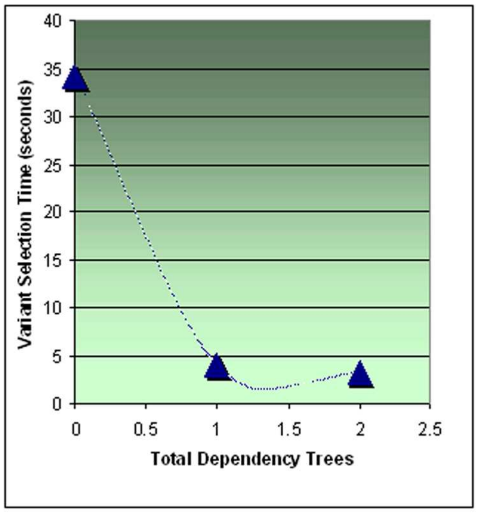


Figure 20: Scatter Performance as PLA Dependency Trees are Introduced

6.5 Field and Stress Testing Scatter

We conducted a series of field tests with real mobile phones and a series of stress tests to determine how on-demand variant selection would scale with Scatter. We integrated Scatter with an open-source over-the-air provisioning server called JVending. JVending delivers mobile applications to devices via HTTP.

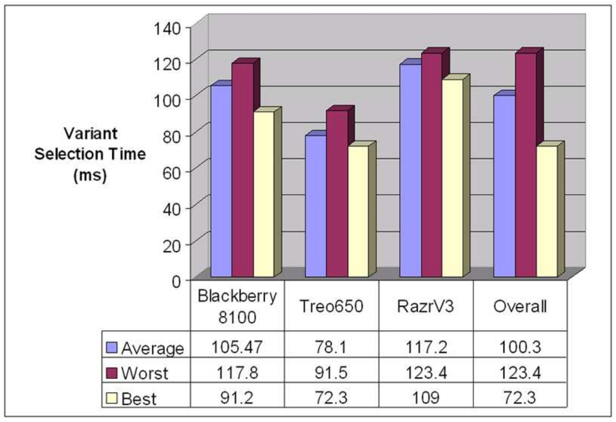
Our tests used a mix of real hardware and synthetically created requests. The actual hardware used was a Blackberry 8100, Motorola Razr V3, and Treo 650 mobile phone. The stress tests were performed using Apache JMeter to send high numbers of synthetic mobile phone provisioning requests. JMeter is an application for stress testing web applications by sending varying numbers, types, and configurations of HTTP requests. We used JMeter to simulate requests since it was infeasible to manually produce large numbers and rates of requests using real mobile phone hardware. The goals of these tests was to (1) ensure that real hardware could be provisioned correctly by Scatter and (2) determine the number of provisioning requests per second that could be handled by Scatter.

The product-line used for testing was the train food services application presented in Section 2. The product-line's feature models comprise a total of 56 features. For the field tests, we selected hardware for a commodity x86 server. The testbed was a Windows XP machine with a 2.6 gigahertz Intel Core DUO CPU, 3 gigabytes of DDR2 6400 RAM, a 10,000 rpm SATA harddrive, and dual gigabit-ethernet network cards. The JVending provisioning web-application was run in Apache Tomcat 6.1.0 using a Java 1.5 JVM in server mode. The Tomcat server and JVending application were configured with all logging disabled.

We used the Wireless Universal Resource File version 2.1.0.1 and its associated Java querying libraries to match static device capabilities to device types using the *UserAgent* header parameter included with requests. The WURFL database contains information on roughly 400 capabilities for approximately 5,000 devices. We do not include the WURFL querying time in our results (although it was typically no more than 3-4ms).

Typical web servers may receive hundreds, thousands, or more requests per second. Although we do not expect a typical provisioning server to receive such high request rates, constraint-solver based software reuse must still provide relatively high performance. To test Scatter's variant selection throughput, we used JMeter to generate a 1,000 synthetic provisioning requests from 3 different mobile phone types. The synthetic request formats were derived by sending real HTTP provisioning requests from the phones to the provisioning server and capturing the included request headers. From the point of view of the provisioning server, there was no difference between the requests produced by JMeter and the actual device.

We measured the average variant selection time for both each individual mobile phone type and overall for all phone types. The results shown in Figure 21 present the time required by Scatter to derive a first class food services variant for each device. The times shown in this figure do not include the time to send the requests across a cellular network or download the selected variant since these attributes of provisioning are outside the scope of this paper.



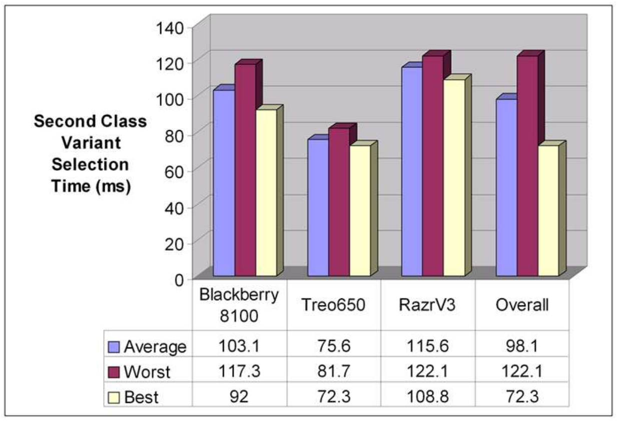
	Blackberry 8100	Treo650	RazrV3	Overall
Average	105.47	78.1	117.2	100.3
Worst	117.8	91.5	123.4	123.4
Best	91.2	72.3	109	72.3

Figure 21: First Class Food Services Variant Selection Time Over 1,000 Provisioning Requests

As shown in Figure 21, Scatter averaged under ∼120ms for all device types. Scatter could reasonably support approximately 9 requests per second. One interesting observation from this data is that the selection times for our 56 feature train food service application models were significantly faster than those of our 50 component model in the synthetic experiments.

When comparing the synthetic and train food service feature models, we found that properly specified real feature models tend to have large numbers of constraints between features. Our synthetic feature models were significantly less constrained (had a higher degree of variability) than our food services application. Less constrained models typically have far more features/components that are not disabled by target device characteristics and must be included in the resource constraint solving. We expect that this result will apply to other mobile applications since they are often carefully matched for the features of the target device.

We repeated the same test with Scatter to select a second class variant for each device. The results from the second test are shown in Figure 22. There was little difference in selection time for a second class versus a first class variant. If Figures 21 and 22 are compared, the average selection time differs by approximately ∼2ms less per device for second class variants. We attribute this difference to the slightly higher variability of first class variants. First class variants can select between two different customer locators whereas the second class variants cannot.



	Blackberry 8100	Treo650	RazrV3	Overall
Average	103.1	75.6	115.6	98.1
Worst	117.3	81.7	122.1	122.1
Best	92	72.3	108.8	72.3

Figure 22: Second Class Food Services Variant Selection Time Over 1,000 Provisioning Requests

7 Results Analysis: Mobile PLA Design Strategies

Although Scatter achieved a throughput of ∼9 requests a second, Product-line designers still must be careful when building a PLA for automated software reuse and real world dynamic over-the-air provisioning. Clustering, hardware, and constraint solver improvements can increase variant selection throughput. Product-line designers can also help increase performance by designing their product-line models for automated software reuse. Based on the results we collected from the experiments, we devised a set of mobile PLA design rules to help improve variant selection performance and address Challenge 5 of Section 3. The remainder of this section presents the design rules we gleaned from our results.

Maximize variant selection result caching. If a product-line is designed carefully, a provisioning server can cache the results of variant selection requests to greatly improve the performance of provisioning. Scatter need only be invoked when a variant must be found for a new device/-context/capabilities signature. For example, two identical Blackberry 8100 mobile phones in first class can reuse the same application components in the same configuration. The majority of requests will be for previously encountered device/context combinations, so previous component reuse decisions will still apply.

Context dependent decisions make caching harder. Product-lines can limit the number of contexts that a provisioning server is interested in. For example, the train food services application is interested in differentiating devices owned by first and second class passengers. The CabinClass context effectively doubles the number of device/-capability/context signatures that the server must cache. The number of unique values for CabinClass acts as a multiplier for the number of configurations that the provisioning server may need to cache. In this example, the provisioning server needs to cache separate variant selection decisions for devices in first class and second class cabins. Designers should attempt to use as coarse-grained context information as possible to limit this multiplier effect.

Limit the situations where resource constraints must be considered. Resource constraints also can limit what the server can cache and are the most time consuming component of variant selection. For example, if two identical Blackberry 8100 devices are encountered in first class, one device having 72K of remaining storage capacity and the other with 2mb of remaining storage capacity, the selection results from the first device will not be applicable to the second device. Either Scatter must be reinvoked for each new storage space value or a method is needed to identify when differing storage values will still produce identical results and thus can be cached.

One strategy is to broadly categorize devices based on remaining storage capacity. For example, all devices can be placed into one of three groups, devices with more than 70K of remaining storage capacity, devices with 14K to 70K, and devices with less than 12K. Any device with 70K can host any combination of the components and features of the food services application, and thus resource constraints do not need to be considered. For devices with 12K to 70K, constraint solving is necessary since multiple but not all configurations are valid. Finally, with less than 12K, no menu images can be deployed to the device but any combination of the application components are possible.

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.

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, there are relatively few *valid* combinations of the JAR files, even though there are a large number of *possible* ways the PLA can be composed.

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.

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. As we pointed out earlier, however, it is important that service classes are course-grained since they can adversely affect caching.

8 Related Work

This section compares our research on Scatter with other tools and techniques that can be used to help automate the selection of reusable software components for a mobile device. We first compare our work to other theoretical techniques for using product-line models to derive which components should be reused for a device. Next we compare Scatter to frameworks for adapting applications and content to the capabilities of a mobile device. Finally, we evaluate Scatter against other tools that allow developers to build product-line models and derive valid variant configurations.

Product-variant derivation techniques. In [25], 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 [25] 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 [26], this approach does not provide a requirements-driven optimal variant selection engine like Scatter. Additional discussion of the differences between constraint-based variant selection and Mannion's logic-based approach is available in [9].

A mapping from feature selection to a CSP is provided by Benavides et al. [9]. 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, unlike this paper, Benavides et al. do not address how PLA design decisions can be used to improve constraint solver performance.

Many complex modeling tools are available for describing and solving combinatorial constraint problems, such as those presented in [27, 15, 34, 12, 20]. 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. These papers also have not addressed how PLA design decisions influence variant selection speed.

Adaptation frameworks for mobile devices. Chisel [22] provides an adaptive application framework for mobile devices based on policy-driven context aware adaptation. This framework allows a running application to adapt to handle resource and other context-based changes in its environment. Although Chisel allows an application to adapt to a particular device's characteristics, it is not sufficient for PLA variant selection for the following reasons

- Chisel assumes that the core functionality of the application does not change via adaptation, which is not the case in the scenarios we describe, where PLA variants may share components but function very differently.

- Chisel is based on explicit developer-provided policies that describe how to adapt to changing conditions. These policies are produced manually and thus may not provide optimal or even good adaptation procedures to handle variant selection based on the environment. In contrast, Scatter automates and optimizes component selection. Automating component selection is key when hard constraints, such as resource consumption, are present.

- Scatter's optimization algorithms provide guarantees on solution quality, whereas Chisel's manually produced policies give no guarantee of solution quality.

- Scatter does not assume that the functionality of the variants is identical and can thus handle the selection of multiple variants to deploy.

In [23], 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 [23], Scatter not only provides adaptation for a device, but also optimizes adaptation of the software with respect to its provided PLA cost function.

Product-line modeling and variant derivation tools. The Eclipse Feature Modeling Plug-in (FMP) [8] provides feature modeling capabilities for the Eclipse platform. FMP allows developers to build feature models to capture the rules governing product-line configuration. FMP can also enforce product-line constraints as developers build variants. Although FMP can automatically map from Java code to feature models, FMP does not provide a mechanism for discovering and mapping mobile device capabilities to product-line models or observing resource constraints. Scatter provides both of these missing critical capabilities. We are collaborating with the FMP research group to apply Scatter's on-demand probing techniques to other domains [37].

Pure::variants [10] is a commercial tool for modeling product-lines using feature models. Developers use Pure::variants to describe a product-line and the constraints between features. Given a feature model, Pure::variants can derive values for any remaining unconfigured features that are mandated by the product-line. Unlike Scatter, however, Pure::variants does not take into account resource constraints. Moreover, Pure::variants is designed to be used at design-time by a modeler and does not provide support for automated target discovery and variant selection.

Big Lever Software Gears [11] is another widely used commercial product-line modeling tool. Software Gears posesses similar capabilities to Pure::variants. Developers describe the rules governing the variable parts of their product-line and Software Gears can derive values for reqiured but unconfigured variabilities. Software Gears does not consider resource constraints or have a mechanism for performing automated autonomous variant selection as Scatter does.

9 Concluding Remarks

Product-line architectures (PLAs) can be used to describe the rules for reusing software components on different mobile devices. Each time a new device is encountered and an application must be assembled from existing software components, a new application variant can be derived from a product-line for the device's capabilities. Mobile software is often deployed using over-the-air provisioning, which requires online selection of reusable components for an application variant. As discussed in Section 3, existing reuse approaches do not address the unique challenges of dynamic software reuse for mobile devices.

Dynamically assembling reusable software components into an application for a mobile device is a challenging domain that can benefit from automation since there are too many complexities and unknown device characteristics to account for all possibilities manually *a priori*. Constraint-solver based automation is a promising technique for online variant selection. This paper describes how our Scatter tool supports efficient online variant selection. By carefully evaluating and constructing a PLA selection model based on the design rules presented in Section 7, developers can alleviate the effects of worst-case solver behavior. As shown in Section 6, a constraint-based variant selection approach that includes resource constraint considerations can provide sufficient performance to dynamically select variants for over-the-air provisioning of mobile software.

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

- Although push and pull capability gathering models are commonly used for over-the-air provisioning, neither is ideal for automated software reuse from product-lines that leverage context information. On-demand probing—which is a hybrid of the push and pull models described in Section 4.1.3—can be used to obtain the information completeness that a push model provides while simultaneously minimizing human interaction.

- Resource constraints can be incorporated into an, constraint-based approach to reusing software, but are time consuming to solve as we showed in Section 6. Product-line designers should therefore attempt to design their constraints to only consider resource constraints when absolutely necessary, such as when a device could support multiple possible component configurations but not all components can fit in the device's memory.

- Although Scatter can automate variant selection, it works best when the constraints on a PLA's reusable software components are 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 therefore be constructed carefully to avoid performance problems. As described in Section 7, the most valuable strategies involve exploiting points of course-grained variability, such as packaging variability or service classes, to allow the variant selector to prune away large numbers of possible variants.

- Dynamically packaging reusable software components into an application is not easy. Other research [6] has evaluated different mechanisms to manage packaging and compilation variation, however, building a dynamic packaging mechanism still takes work. In future work, we plan to evaluate different strategies of dynamically packaging application variants derived by Scatter.

- When a PLA for a mobile device is properly specified with good constraints, Scatter can solve models involving 50 components in $\sim$100ms, as shown in our experiments in Section 6.5. This performance should be 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.

- Developers normally focus on the functional variability in a product. It is also important to evaluate non-functional variability, such as packaging variability. As shown in Section 7, although a product may have high functional variability, it can be significantly less variable with respect to the number of ways it can be packaged and deployed or in terms of its memory footprint. These non-functional aspects can be exploited to reduce the complexity of automated variant selection.

Scatter is available in open-source form from `www.sf.net/projects/gems`.

References

[1] Choco Constraint Programming System, http://choco.sourceforge.net/.

[2] Google Mobile Products, http://www.google.com/mobile/.

[3] Develop/Optimize Case Study: Macrospace's Dragon Island. *Nokia Forum, http://www.forum.nokia.com/main/*, 2004.

[4] O. M. Alliance. OMA Client Provisioning V1.1 Candidate Enabler, http://www.openmobilealliance.org/release_program/cp_v1_1.htm.

[5] O. M. Alliance. OMA Download over the Air V2.0 Candidate Enabler, http://www.openmobilealliance.org/release_program/dlota_v2_0.html.

[6] V. Alves, I. Cardim, H. Vital, P. Sampaio, A. Damasceno, P. Borba, and G. Ramalho. Comparative Analysis of Porting Strategies in J2ME Games. *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 123–132, 2005.

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

[8] M. Antkiewicz and K. Czarnecki. FeaturePlugin: feature modeling plug-in for Eclipse. In *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 67–72, New York, NY, USA, 2004. ACM Press.

[9] 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.

[10] D. Beuche. Variant management with pure:: variants. Technical report, Pure-systems GmbH, http://www. pure-systems. com, 2003.

[11] R. Buhrdorf, D. Churchett, and C. Krueger. SalionŠs Experience with a Reactive Software Product Line Approach. *Proceeding of the 5th International Workshop on Product Family Engineering. Nov*, 2003.

[12] 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.

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

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

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

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

[17] 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.

[18] 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 Techňnologies and Systems*, March 2003.

[19] M. Dillinger, R. Becher, and A. Siemens. Decentralized software distribution for SDR terminals. *Wireless Communications, IEEE [see also IEEE Personal Communications]*, 9(2):20–25, 2002.

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

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

[22] J. Keeney and V. Cahill. Chisel: a Policy-driven, Context-aware, Dynamic Adaptation Framework. *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, pages 3–14, 2003.

[23] 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.

[24] A. T. Luca Passani. Wireless Universal Resource File, http://wurfl.sourceforge.net/.

[25] 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.

[26] 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.

[27] 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.

[28] S. Microsystems. Over The Air User Inititated Provisioning Recommended Practice. May 2001.

[29] 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.

[30] 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.

[31] Nokia. Java Technology Enables Exciting Downloading Services for Mobile Users. October 2003.

[32] 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.

[33] D. Saha and A. Mukherjee. Pervasive computing: a paradigm for the 21st century. *Computer*, 36(3):25–31, 2003.

[34] 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.

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

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

[37] J. White, K. Czarnecki, D. C. Schmidt, G. Lenz, C. Wienands, E. Wuchner, and L. Fiege. Automated model-based configuration of enterprise java applications. In *EDOC 2007*, 2007 (to appear).

[38] J. White, A. Nechypurenko, E. Wuchner, and D. C. Schmidt. Optimizing and Automating Product-Line Variant Selection for Mobile Devices. In *11th International Software Product Line Conference*, September 2007.

[39] M. Womer and F. Telecom. Device Description Landscape.

[40] 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.