Proceedings of the 17th World Congress
The International Federation of Automatic Control
Seoul, Korea, July 6-11, 2008

Model-Driven Product-Line Architectures for Mobile Devices

Jules White and Douglas C. Schmidt

Vanderbilt University
Department of Electrical Engineering and Computer Science
e-mail: {jules,schmidt}@dre.vanderbilt.edu).

Abstract: The large number of mobile device types and possible device configurations makes it possible to deliver mobile applications to user devices that are not fully compatible with device characteristics. In these situations, users must perform the tedious and error-prone tasks of altering their device configurations to meet the needs of applications. Mobile application product-lines and automated product variant selection engines are promising approaches for deriving and delivering custom tailored applications to devices, thereby eliminating the need for end-user configuration.
This paper provides three contributions to model-driven product-line variant selection for mobile devices. First, it describes an infrastructure-driven product variant configuration tool that tailors a product variant to the specific capabilities of a mobile device. Second, it shows how this tool automates the capture of device capabilities and maps them to product-line feature models. Third, it shows how a constraint solver can be used to derive a valid product variant and incorporate a device's resource constraints into the derivation process.

1. INTRODUCTION

Current trends and challenges. Mobile devices, such as PDAs and cell phones, are proliferating rapidly. Cell phone technology has improved dramatically since 2003 and most popular devices have some level of PDA capabilities. Users of mobile devices have thus created a large and growing market for third-party mobile applications, such as games and office tools. These third-party applications are often delivered via *over-the-air provisioning*, *i.e.*, installed over a cellular data connection from a remotely accessible provisioning server.

It is hard to create a single application that can handle a large number of device configurations, however, since they vary greatly in hardware capabilities, such as display size, processing power, and memory. Moreover, applications must be carefully configured to account for the unique characteristics of each device, particularly when they have limited hardware resources. For example, games are often delivered with different resolution images or numbers of levels depending on the memory available on the device. To select a correct configuration of an application that includes media, therefore, requires considering the overall resource constraints of the device.

One approach is to build a small number of variants of an application that can be used to support many devices. Although these application variants are designed to work in wide range of device configurations, many applications still require device users to manually configure their devices to accomodate variants. For example, Opera browsers run a series of tests when they are first launched to ensure that device configurations are compatible with installed browser variants. If device configurations are not compatible, users receive an error message and a link to a support forum to help diagnose the configuration mismatch.

Although the approach of supporting a large number of devices with a small set of variants is common in mobile applications, it puts undue responsibility on device users to perform device and application configuration. These users, however, rarely have the initimate understanding of either the application or device's configuration nuances. Users are thus in the tedious and error-prone process of wandering through support documentation to resolve mismatches.

Solution approach → Infrastructure-driven product variant configuration. Product-line architectures (PLAs) (Clements and Northrop [2002]) are a promising approach to help developers reduce the effort of mobile application development and configuration (Anastasopoulos [2005], Zhang et al. [2003], Muthig et al. [2004]). A PLA is comprised of a set of reusable components that can be composed in different software configurations (variants) for different device configurations. Constructing a product-line variant involves finding a way of reusing and composing the product-line's components to create an application that will function correctly on a specific device.

There has been much work Benavides et al. [2005], Lemlouma and Layaida [2004], Czarnecki et al. [2005], van der Storm [2004], Sabin and Weigel [1998] on automating product-line variant selection, This prior work, however, does not focus on techniques for incorporating resource constraints into variant selection for mobile applications. In particular, the process of deriving and delivering a fully-configured application variant for each custom device *signature* (*i.e.*, the unique capabilties of each device) presents challenges that have not been addressed by existing product variant derivation techniques Mannion and Camara [2003], Mannion [2002].

Another challenge of selecting and delivering a custom variant for mobile devices is that the configuration is highly decoupled in time and space from the place where the device capability information resides. To select a variant for a device, therefore, strategies must be developed to capture the important configuration characteristics of an individual device and deliver them to the provisioning server. Existing techniques for delivering de-

vice capabilities to a server either employ static databases capabilities associated with device model numbers or user-provided configuration values. Static database approaches cannot capture the numerous possible variations between devices of the same model, whereas users may not know how to obtain or provide incorrect configuration values.

In prior work White et al. [2007b] we defined an architecture for automated product variant selection for mobile devices that accomodates resource constraints and addresses how configuration values are migrated from a device to a provisioning server. This paper focuses on a previously unexplored dimension of work on automated variant selection: *a technique called infrastructure-driven product variant configuration that tailors product variants for their target infrastructure*. In particular, we show how combining PLAs with an automated product-line variant selection engine can deliver application variants for mobile devices that are tailored to the unique configuration of each individual device. Delivering these customized application variant to users via over-the-air provisioning helps alleviate the tedious and error-prone manual configuration traditionally required from users.

Paper organization. The remainder of the paper is organized as follows: Section 2 presents a case study used throughout the paper to motivate the need for infrastructure-driven product variant configuration; Section 3 describes the challenges of selecting a product variant for an arbitrary mobile device; Section 4 shows how we use model-driven development tools to capture the relationship between device configuration and application configuration; Section 5 discusses infrastructure-driven product variant configuration and shows how the target device's configuration can be used to drive application configuration; and Section 7 presents concluding remarks.

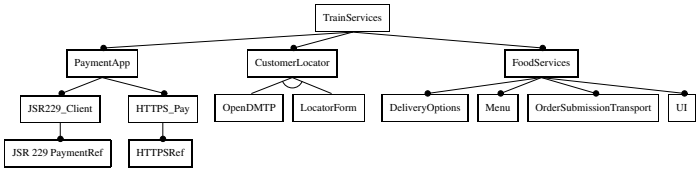

Fig. 1. Feature Model for Train Services Applications

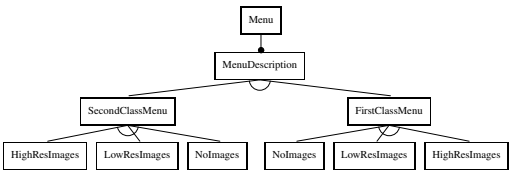

Fig. 2. Food Services Menu Feature Model

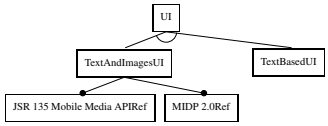

Fig. 3. Food Services UI Feature Model

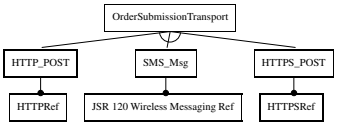

Fig. 4. Food Services Order submission Feature Model

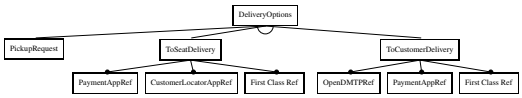

Fig. 5. Food Services Delivery Options Feature Model

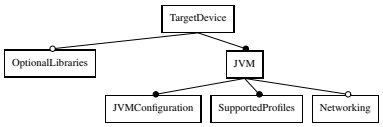

Fig. 6. Target Device Feature Model

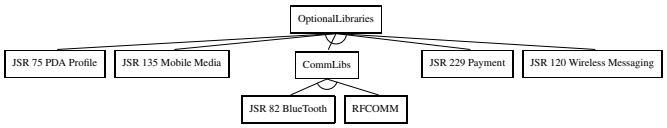

Fig. 7. Java Optional Libraries Feature Model

2. MOTIVATING CASE STUDY EXAMPLE

To motivate our work on infrastructure-driven product variant configuration, this section describes a mobile application case study for ordering food on a train to show the complexity of tailoring an application variant for an arbitrary device. In this example, when a passenger boards a train, they can download a variant of the train services application to their mobile device from the railroad company's provisioning server. First class passengers can use this application to order gourmet foods and have the food delivered directly to their seats. Second class passengers can order food via a standard (not gourmet) menu, but must go to the restaurant car to pickup the food.

The food services application is built using a product-line that provides differing image sets of menu items that can be downloaded with the application depending on the device resources. For example, Figure 2 shows how high and low resolution images can consume varying amounts of storage space. Applications can also be adapted to functional variations of different devices. For example, Figure 3 shows a *TextAndImagesUI* for devices with *JSR 135 Mobile Media API* support and a *TextBasedUI* for devices without JSR 135 support. The multiple variations in application configuration be carefully matched against the capabilities of requesting devices.

3. CHALLENGES OF OVER-THE-AIR PROVISIONING

Creating a functioning mobile application variant requires that the application's configuration matches the capabilities and configuration of the hosting device. As described in Section 1 user are often required to partially adapt device configurations to match application needs. In previous work (White et al. [2007b]), we showed that rather than partially adapting devices to applications, applications can be completely adapted to devices. This section explains why adapting the application to the device is hard to help motivate the solutions we describe in Section 4.

The key to meshing a mobile application to the device it runs on is understanding the device's unique configuration. Although documenting and reasoning about a device's configuration may initially appear easy, devices have a significant number of points of variability, including:

(1) **Hardware** - Devices have widely varying display sizes, memory capacities, storage capacities, processing capabilities, data connection throughputs, etc.
(2) **Middleware** - Java Virtual Machines (JVM) deployed on mobile devices come in different configurations and profiles that support different core libraries and functionality. Each JVM also can be deployed with further optional libraries, such as implementations of various Java Specification Requests (JSRs). JVMs may also have varying stack sizes and even bugs that can affect their functionality (Alves et al. [2005]).
(3) **User customization** - Many phones, such as the Blackberry 8100, allow users to change important platform options, such as TCP/IP settings. Moreover, users can install third-party software and media, thereby reducing storage space and available memory. Users can also upgrade firmware.
(4) **Service provider customization** - Each individual cell phone service provider often customizes each mobile phone for their service offerings. Many service providers lock phones to their networks, may restrict phone functionality to force customers to use pay services, provide special APIs that are tightly linked to their services, and/or limit the customizations that can be done to the phone to reduce support costs.
(5) **Context data** - An emerging trend is to treat devices differently depending on context data, such as the geospatial coordinates of the device. Moreover, devices may be categorized by attributes of a customer's account (*e.g.*, platinum, gold, or silver level customers) or a customer's purchase (*e.g.*, first vs. second class).

Even after a device's unique characteristics are known, a variant selection engine must determine how to assemble the various reusable components of the product-line into an application that fits the device's configuration. An analogous problem is trying to find a way to connect two semi-complete halves of a puzzle by adding a number of currently unused intermediate pieces. For example, in the train food service application from Section 2, once it is known that a device does not support the JSR 135 Mobile Media API, a component must be found that can bridge the application's need for a GUI with the requirement that the chosen GUI not use the Mobile Media API. In this example, the connecting piece is the TextBasedUI.

The connecting of two halves can be far more complicated than the direct inference from the previous example. In particular, there may be multiple sets of pieces that can bridge the two configurations; the goal is to select the set that minimizes the memory consumed by the application or the bandwidth required to transfer the application across a cellular connection to the device. Moreover, selecting one component, such as the TextBasedUI, will exclude the use of other components, such as the HighResImages and LowResImages for the menu. When all these rules and global resource constraints are considered, selecting a variant is hard.

4. USING MODEL-DRIVEN DEVELOPMENT TO REDUCE COMPLEXITY

Third-generation programming languages, such as Java, C#, and C++, are not well suited to capture the high-level rules needed to determine how device variability affects product variant construction. These languages require writing a significant amount of custom logic to correctly assemble an application variant for a given device configuration. Moreover, this complex assembly logic must be adapted and maintained as the product-line evovles and new device types emerge.

In contrast, model-driven development (MDD) is a promising technique that can be used to provide a higher-level of abstraction and capture these composition rules that are hard to encode with third-generation programming languages. Feature modeling (Kang et al. [1990], Antkiewicz and Czarnecki [2004]) is an MDD technique that can be used to describe an application in terms of functional and non-functional variations (features) and the rules for composing features. This technique provides an intuitive mechanism for describing variability and has been applied across a large number of domains, including automotive construction, boilers for nuclear reactors, and mobile devices. Formal mechanisms also exist to translate feature models and the selection of product variants into Constraint Satisfaction Problems (CSPs) (Benavides et al. [2005], White et al. [2007a]).

A CSP is a set of variables and a set of constraints over these variables. A constraint solver is used to find a set of values (a labeling) of the variables for which all the constraints hold. Often, not only is a labeling needed, but a labeling that simultaneously maximizes or minimizes the value of a function. With a CSP formulation of variant selection, a constraint solver can be used to derive a valid variant, as follows:

- The variables are the application features, *e.g.*, TextBasedUI, TextAndImagesUI, HighResImages, LowResImages, etc.
- The values of the variables determine if a particular feature is in a specific variant, *e.g.*, TextAndImagesUI = 1 implies that the variant has the TextAndImagesUI user interface.
- The constraints are the composition constraints for the features, *e.g.*, TextAndImagesUI requires the JSR 135 Mobile Media API and the SMS_Msg order submission transport requires the JSR 120 Wireless Messaging API.

A labeling of a product variant CSP is a set of features that can be enabled to create a complete and correctly constructed application variant for a device. An optimized selection can produce a variant that functions correctly within a device's constraints and minimizes a variant fitness function, such as the total memory consumed by the variant.

4.0.0.1. Benefits of using constraint-based variant configuration Using a constraint solver eliminates most of the manual variant selection complexity described in Section 3, allows for automation, and ensures solution quality. For example, when a constraint solver is used to derive a variant, the variant selected is not only correct with respect to the feature composition constraints, but it also guarantees worst case bounds on solution quality. A variant can be selected that is optimal or an approximation algorithm can be used to find a solution that is at most a guaranteed percentage away from the optimal solution.

4.0.0.2. Challenges of using a CSP configuration model Transforming a product's feature model into a CSP, however, is a tedious and error-prone process. Typically, product engineers who can easily build a feature model for an application do

not possess the mathematical background to perform a manual translation of a feature model into a CSP. Moreover, as a product-line evolves, the CSP must be updated continually to reflect the application's feature model. Expending a large amount of energy to perform this manual and non-intuitive mapping is less than ideal.

4.0.0.3. Addressing the challenges of CSP-based configuration with MDD To address the limitations with conventional CSP techniques—thereby reducing the cost of leveraging a constraint solver for product derivation—we have developed a model-driven development tool to provide an intuitive modeling language to domain experts and automatically generate the complex CSP from domain models. The graphical tool uses feature models to capture the variabilities in the application and uses code-generation to compile the feature models into a CSP that can be operated on with a constraint solver. Using a MDD approach shields domain experts from the tedious and error-prone translation from feature models to CSPs. The graphical feature modeling tool is based on the Generic Eclipse Modeling System (GEMS) and runs in Eclipse.

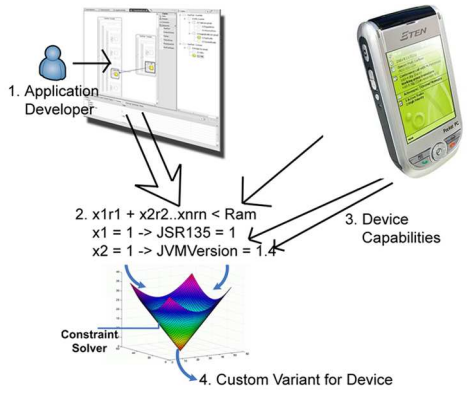

Fig. 8. The Scatter MDD Approach

Figure 8 shows an overview of our MDD approach. In step 1, the application developers describe the high-level rules of configuring the application using graphical feature models in Eclipse. In step 2, the graphical modeling tool generates a CSP representation of the problem that is in the native format (not a visual format) of the constraint solver. In step 3, when a request from a device arrives for an application variant, the capabilities of the device are plugged into variables in the CSP produced from the graphical feature models. Finally, a constraint solver is used to derive an application configuration that matches the capabilities of the device.

5. INFRASTRUCTURE-DRIVEN PRODUCT VARIANT CONFIGURATION

To derive a valid variant with a CSP, product engineers select certain features that should be enabled in the selected variant by setting their corresponding variables in the CSP. Finding a variant on-the-fly for over-the-air provisioning requires a framework for determining how to set the intial required features to guide the solver. We have developed a model for providing this initial input to the solver called *infrastructure-driven product variant configuration*, which uses two sets of feature models:

- The points of variability in the software application that will be configured, including the menu (first or second class), GUI (text only or text and images), ordering submission mechanism (http, SMS message, etc.), and images of menu items (high resolution, low resolution, etc.).
- The points of variability in the infrastructure to which the application can be deployed, including the mobile device, server, etc.

Constraints are applied to the application and infrastructure feature models that show how the selection of features in the infrastructure model restrict the features that can be selected in the application feature models.

Figure 4 from the train services example in Section 2 shows how the *SMS_Msg* feature (a feature from the application feature set) requires the *JSR 120 Wireless Messaging API* feature be enabled in the infrastructure feature set. The infrastructure features may also include context data, such as cabin class. The *ToSeatDelivery* feature requires that the *First Class* feature be enabled in the infrastructure feature set.

We term our derivation technique infrastructure-*driven* because the infrastructure features are the intial input used by the constraint solver to derive the required application features. When a request arrives for an application variant, the initial action the server takes is to select the infrastructure features corresponding to the capabilities of the requesting device. The provisioning server then asks the solver to derive an application variant that meets the constraints of the selected infrastructure features. The selection of infrastructure features therefore drives the selection of application features.

Infrastructure-driven configuration addresses the original problem motivated in Section 1, namely the need to produce an application variant that is adapted specifically to the requesting device. The infrastructure-driven model also provides a clear architecture for automating software variant derivation: requests arrive, the requestor's configuration is determined, the corresponding features in the infrastructure model are selected, and a solver is used to derive the needed application features. As shown in Section 5.2, the infrastructure-driven model can easily be extended to incorporate resource constraints for media into the derivation process.

5.1 Determining the Infrastructure Configuration

As we discussed in Section 1, adapting the application to the device at the provisioning server migrates the configuration process away from the device. Migrating the configuration off of the device, however, introduces the challenge that the configuration process does not have direct access to the device to evaluate its configuration. Instead, the device must either send its entire configuraton along with the request or the provisioning server must obtain device configuration information using the limited information available with a standard provisioning request. Several ways of addressing this problem are summarized below.

5.1.0.4. Approach 1: Send configuration values with the provisioning request In this approach users select from a number of application variants for common device types. Again, these variants are configured for the standard device configuration and often require further device configuration by the user. With this approach,the user is sending the device's configuration information to the provisioning server. The advantage of this

approach is that nearly any type of configuration information from the device, including user customizations or context data, can be provided to the provisioning server. The disadvantage of the approach, however, is that it relies on error-prone user input. A device user may not know the requested configuration values (such as JVM configuration). Context data, such as ticket number, can be validated to ensure that the user provides correct values. Device capabilities, however, cannot be validated since the provisioning server does not have the target device to check them against.

5.1.0.5. Approach 2: Query a device capabilities database for configuration values In this approach a device capabilities database (Womer and Telecom [2006]), such as the Wireless Universal Resource File (WURFL) (Luca Passani [2007]), is used to associate device capabilities with the unique *UserAgent* header sent with HTTP requests from a device. This approach has the advantage that the device capabilities that do not vary across device models can be automatically deduced without user interaction. Furthermore, the data derived from a device capabilities database is guaranteed to be correct (assuming it has been properly produced from device specifications) The downside to this approach, however, is that it cannot obtain dynamic information, such as context data or user customizations.

5.1.0.6. Approach 3: On-demand probing. In this approach a static device capabilities database is used to deduce most configuration properties of the device. If, however, values for infrastructure configuration properties are needed that are not available in the database, the provisioning server returns a probe to the device to obtain the missing information. The probe runs a check on the device and posts the results to the server to obtain the final variant for the device.

A very simple probe can be a form that is returned to the user to obtain context data, such as the traveler's ticket number. When this data is returned to the server, it can derive cabin class. A probe may also be a small executable, such as a Java Midlet, that can determine properties that the user is unlikely to know, such as JVM configuraton. For example, a probe can be downloaded that attempts to lookup a Java class by name, and if it cannot, notifies the server that a particular library is not present on the requesting device.

The advantage of the on-demand probing approach is that it minimizes user-provided configuration values while still being able to incorporate dynamic device configuration information. A drawback of the approach is that the executable probe will leave behind a software artifact that the user may need to manually remove later. Despite the possibility of left-over software artifacts, we chose this approach because it posseses the advantages of both the first two approaches.

In situations where context data and static device data (*e.g.*, display size, JVM configuration, etc.) are needed, the on-demand probing approach can capture context data with a simple HTML form and device characteristics from a device capabilities database. In this scenario, the first approach would require user to provide the device capabilities, which is tedious and error-prone. The second approach is also insufficient since it cannot capture context data effectively. The on-demand probing approach, in contrast, provides a reliable source for device capabilities and allows the capture of context data.

5.2 Incorporating Resource Constraints for Media

Media (*e.g.*, audio, video, images, etc.) must often be delivered along with a mobile application. As described in Sections 1 and 3, media consumes a large amount of a devices resources, particularly storage space. When variant selection takes place, the overall resources consumed by the features selected for a variant must not exceed those available on the requesting device. These types of resource constraints are global feature selection constraints that are extremely hard to manage manually since they are combinatorially complex.

In previous work (White et al. [2007a]), we delineated a process for incorporating global resource constraints into the feature selection process. Our formulation of global resource constraints has been incorporated into infrastructure-driven product variant configuration. The formulation of resource constraints in the CSP produces a large number of variables (resource types X total features = total variables). Without an MDD approach, producing this large CSP is extremely tedious and error-prone. With the MDD approach, the CSP is automatically generated by the modeling tool.

As part of the initial step of selecting the infrastructure features corresponding to the configuration of the requesting device, the provisioning server can also set the upperbounds used by the CSP constraints limiting resource consumption. For example, if a Blackberry 8100 phone requests a variant of the train food service application, as part of the intial infrastructure feature selection process, the bounds for the remaining storage left on the phone's media card can be set in the feature selection CSP.

A major challenge of the on-demand probing approach, however, as we have shown in (White et al. [2007b]), is that resource constraints make variant derivation time consuming. Attempting to optimize the variant with respect to a cost function also can require significant time. The time consumed by both of these activities is an exponential function of the number of unfixed variabilities (un-labeled variables in the CSP).

What we learned through our work with infrastructure-driven configuration is that the number of unfixed variabilities is typically low after the application features are filtered by the constraints on the selected infrastructure features. By carefully constraining the application features with respect to the infrastructure features, therefore, product engineers can usually produce feature models that can incorporate resource constraints or optimization and be configured in reasonable time frames.

6. RELATED WORK

Czarnecki et al. (CZARNECKI et al. [2005]) propose the process of *staged configuration* of feature models. In staged configuration, multiple parties iteratively eliminate variability to configure a product variant. The idea of infrastructure-driven software product configuration is a specialized of generic staged configuration techniques. In particular, infrastructure-driven configuration focuses on the specific challenge of creating a custom crafted product variant for a target host, whereas staged configuration does not detail the specifics of how a product variant is configured for a specific target infrastructure. Moreover, infrastructure-driven product configuration provides an architecture for performing automated product variant derivation, whereas staged configuration does not detail an architecture for automation.

The product configuration automation architecture described in this paper relies on the mapping from feature selection to a CSP provided by Benavides et al. (Benavides et al. [2005]). Our work defines a number of complementary extensions to the work of Benavides. First, we show how a CSP model can be used to automate derivation specifically for mobile devices. Second, we present infrastructure-driven product variant configuration, which solves the key challenges of determining what CSP variables to constrain when searching for a variant that conforms to a device. Finally, we present challenges and solutions specific to mobile devices that related to migrating the source of the configuration CSP's input data to a remote device.

7. CONCLUDING REMARKS

Due to the resources limitations, mobile applications must be carefully crafted to fit the unique capabilities of each device. The large number of different mobile devices and the significant variability between device models makes it hard to create a single application variant that can function correctly across all devices. Even within a single device model, there can be a significant amount of variability, as shown in Section 3. Application variants are often delivered to devices in a manner that requires users to perform some level of device configuration to allow the application to function correctly.

By combining a mobile application product-line, a constraint solver, and the infrastructure-driven product variant configuration technique presented in Section 5, mobile applications can be customized for each target device. Infrastructure-driven product variant configuration uses an infrastructure feature model and an application feature model to capture the affect of device capabilities on application features. This dual model approach allows an automated product derivation engine to select features in the infrastructure model corresponding to the target device and derive an application variant that is configured correctly for the device. This approach application configuration can alleviates tedious and error-prone end user device configuration.

A key challenge of using an infrastructure-driven configuration approach for mobile devices is that the configuration process is separated in time and space from the device, which contains the data needed to configure the infrastructure feature models. Methods are therefore needed to either migrate the required device configuration information to the configuration engine or to statically deduce the device's configuration from its model. Since both approaches have drawbacks we present a third approach in Section 5.1, called on-demand probing that is a hybrid of the static deduction and configuration sending models. On-demand probing solves some challenges of migrating device configuration information to a provisioning server but introduces other challenges, such as residual probe software artifacts that consume resources on the user devices and must be manually removed by users. In future work, we plan to evaluate different probing techniques to determine ways of avoiding residual software artifacts while still reducing user interaction.

REFERENCES

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.

M. Anastasopoulos. Software Product Lines for Pervasive Computing. *IESE-Report No. 044.04/E version*, 1, 2005.

Michal Antkiewicz and Krzysztof 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.

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.

Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, 2002.

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.

K. CZARNECKI, S. HELSEN, and U. EISENECKER. Staged configuration using feature models. *Lecture notes in computer science*, 10:266–283, 2005.

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

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.

Andrea Trasatti Luca Passani. Wireless Universal Resource File, http://wurfl.sourceforge.net/, 2007.

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.

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.

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.

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.

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

Jules White, Krzysztof Czarnecki, Douglas C. Schmidt, Gunther Lenz, Christoph Wienands, Egon Wuchner, and Ludger Fiege. Automated model-based configuration of enterprise java applications. In *EDOC 2007*, 2007a.

Jules White, Andrey Nechypurenko, Egon Wuchner, and Douglas C. Schmidt. Optimizing and Automating Product-Line Variant Selection for Mobile Devices. In *11th International Software Product Line Conference*, September 2007b.

M. Womer and F. Telecom. Device Description Landscape, 2006.

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.