# Optimizing and Automating Product-Line Variant Selection for Mobile Devices

Jules White,
Douglas C. Schmidt
Vanderbilt University,
Department of Electrical Engineering and
Computer Science
Box 1679 Station B
Nashville, TN, 37235, USA
{jules, schmidt}@dre.vanderbilt.edu

Andrey Nechypurenko,
Egon Wuchner
Siemens AG,
Corporate Technology (SE 2)
Otto-Hahn-Ring 6
81739 Munich, Germany
{andrey.nechypurenko,
egon.wuchner}@siemens.com

## Abstract

*Product-line architectures (PLAs) designed for mobile devices create a unique challenge for deployment and configuration (D&C) planning engines. A D&C planning engine captures the unique OS, middleware, and hardware signature of the device and rapidly finds a valid variant of the PLA that can be supported by the configuration in terms of OS, middleware, and available resources, such as memory. It is hard to develop a D&C engine for a PLA that can both capture the numerous nuanced requirements of a PLA's variant construction rules and the unique characteristics of a large number of mobile devices. It is even harder to fin a valid variant to deploy to the target device fast enough to support automatic deployment in an ad-hoc environment. This paper presents a tool called Scatter whose input is (1) the requirements of PLA construction and (2) the resources available on a discovered mobile device and whose output is the optimal variant that can be deployed to the device. Scatter provides automatic variant selection based on configuration and resource constraints and also ensures that variant selection is optimal with regard to a configurable cost function.*

## 1. INTRODUCTION

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

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

Middleware, such as the Java Micro Edition, will play an increasingly important role developing custom and context-based application delivery. These platforms reduce the burden of handling the nuances between device and OS APIs placed on developers and allow more effective reuse of application software. Common deployment infrastructure [16] and device capability and preference schemas, such as RDF [22] and CC/PP [21, 17], will also make on-demand application deployment more feasible.

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 and RAM), middleware versions (such as Java Virtual Machine versions), hardware capabilities (such as communication protocols like General Packet Radio Service), and service provider restrictions (such as bandwidth usage limits). Handling all these diverse restrictions and producing an application that can be deployed on a large number of heterogeneous devices is hard.

Product-line architectures (PLAs) [6] are a promising approach to help developers manage the complexity of the variability between mobile devices [2, 35, 28]. PLAs [7] enable the development of a group of software packages that can be retargeted for different requirement sets by leveraging common capabilities, patterns, and architectural styles. The design of a PLA is typically guided by scope, commonality, and variability (SCV) analysis [10]. 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 differ-

*MOBISYS* '07 Puerto Rico

ent members of the family of products.

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

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

- We describe Scatter's requirement and resource specification mechanisms and show how they facilitate the capture and analysis of a wide variety of requirement types
- We discuss how Scatter transforms requirement specifications into a format that can be operated on by a constraint solver
- We describe the automated variant selection solver, based on a Constraint Logic Programming Finite Domain (CLP(FD)) solver [18, 31] and show how it can rapidly produce both correct and optimal variants based on the requirements
- We describe an architecture based on repair operations for negotiating with mobile devices when no suitable variant can be found and suggesting ways that the device can make itself a suitable host for a variant
- We describe Scatter's remoting mechanisms that notify it when new devices are discovered and communicate variant selections and
- We present data from performance tests that show how PLA constraints impact resource constraint satisfaction, which is the computationally-intensive component of variant selection.

The remainder of this paper is organized as follows: Section 2 presents the challenges of capturing the requirements and resources for deploying PLA variants to mobile devices and discusses how Scatter addresses them; Section 3 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 4 describes Scatter's repair operator framework and shows how it can be used to deduce modifications to a device, such as increasing memory allocations or downloading additional third-party software, to make variant selection possible; Section 5 analyzes the performance results of applying Scatter to variant selection for an example PLA; Section 6 compares our approach with related work; and Section 7 presents lessons learned and concluding remarks.

# 2. CAPTURING PLA AND MOBILE DEVICE REQUIREMENTS

Traditional processes of identifying valid PLA variants involve software developers manually determining the software components that must be in a variant, the components to configure, and how to compose and deploy the components. In addition to being infeasible in a pervasive environment (where the target device signatures are not known ahead of time and variant selection must be done on demand), such manual approaches are tedious and error-prone and are a significant source of system downtime [11]. Manual approaches also do not scale well and become impractical with the large solution spaces typical of PLAs.

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

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

Capturing and relating composition and non-functional requirements to a mobile device is hard. The remainder of this section describes key challenges of building a compositional and non-functional requirements model of a PLA and outlines how Scatter addresses them. The section also illustrates how Scatter transforms requirements captured in its graphical modeling tool into a format that can be operated on by its variant selection solvers.

## 2.1 PLA Composition Rules

Scatter is a graphical tool that provides a domain-specific modeling language (DSML) for specifying variant selection and provides a visio-like interface, as shown in Figure 2. Scatter allows developers to visually model (1) the components of their PLA, (2) the dependencies and composition rules of components, and (3) the non-functional requirements of each component. The components in the Scatter DSML represent the software components that compose the PLA. PLA developers simply drag-and-drop these compo-

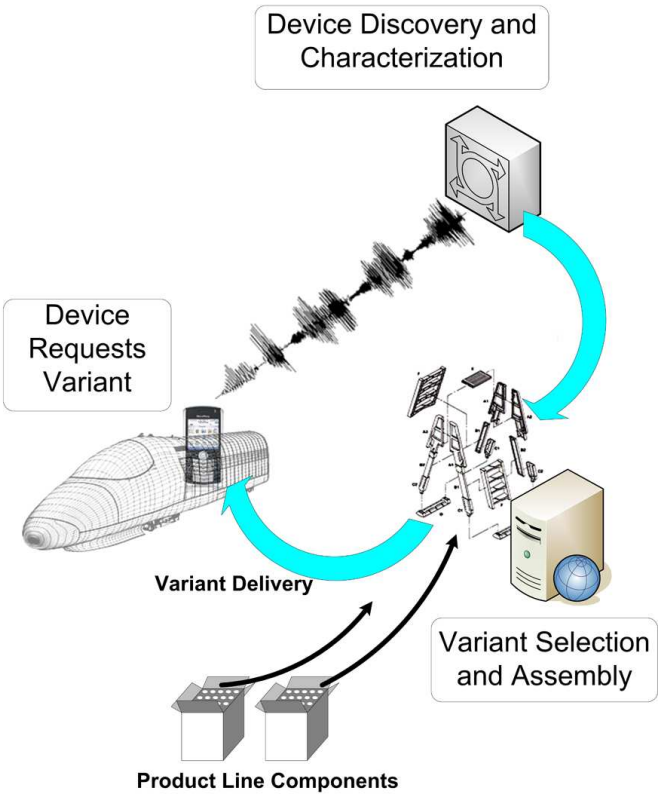


**Figure 1: Selecting a Train Ticket Reservation Service for a Device**

nents from a palette into a model.

To facilitate the analysis of the variant solution space requires a formal grammar to describe the structure, commonality, and variability (SCV) analysis of the PLA and its valid configurations. This customization grammar can then be used to automatically generate and explore the variant solution space. Scatter provides a visual model for capturing the SCV of a PLA, as seen in Figure 2. This view allows developers to formalize which components are available in the PLA, what applications can be constructed, and how each application is composed. The components can be used as an abstraction to describe a PLA both on system structure [26] or using feature modeling [3, 19]. In our approach, configurations of components or features can be modeled as variabilities using Scatter's SCV model.

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

*Applications* are valid compositions of *Components* that provide a higher level of functionality. *Applications* can be composed by specifying a composition predicate (AND or Exclusive OR) and the *Components* to which the predicate should be applied. For our train example, the *FoodService* component is connected to the Exclusive OR predicate, which can

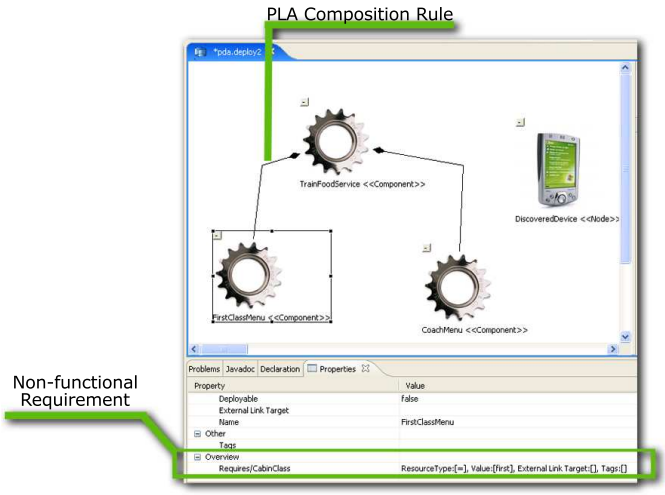


**Figure 2: Scatter PLA Composition and Non-functional Requirements**

be connected to the *first class* and *coach class menu* components. This composition indicates that the *FoodService* component can be deployed with exactly one of these menus.

*Component* dependencies can be constructed hierarchically from other components with dependencies to capture the compositional variability in a PLA. To specify the compositional variability in the PLA, developers build *Component* and *Predicate* trees, which we call Logical Composition Trees. The root *Components* of these trees represent deployable *Applications*. Each level down the tree specifies the *Components* that are required for the level above.

The graphical specification of Logical Composition Trees provides the primary means of specifying PLA composition rules in Scatter. For some situations, however, graphical mechanisms based on AND and XOR may not provide the most concise or intuitive rules. For these cases, explicit composition rules based on a domain-specific predicate logic can be used. These predicate logic constructs are provided by developers and can be incorporated into and used by the Scatter solver. The predicate logic specification for PLA composition is discussed in Section 2.3.

By capturing PLA compositional variability in Logical Composition trees, developers can formally specify how valid variants are composed. With a formal specification of the variant construction rules, Scatter can automatically explore the variant solution space to discover all valid compositional variants of the PLA for a given device, as discussed in Section 3.

## 2.2 Non-functional Requirements Capture

One challenge when building a tool to model a PLA's non-functional requirements is providing a mechanism that can not only allow modelers to express a wide variety of constraint types, but also capture them in a form that can be operated on by a constraint solver. At one end of the spectrum, are textual specifications, such as "this component should only be deployed to devices located in the first-class

cabin running Palm OS," that tend to be intuitive to produce and understand, but are imprecise in meaning and require manual translation to the format expected by a constraint solver.

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

To deduce a variant, a modeling tool must not only be able to capture the high-level picture of the constraints that developers understand, but be able to map this requirement model to one or more constraint solvers. With Scatter, we faced an added challenge that the tool would be used for variant selection across a large number of devices in a large number of contexts. It therefore needed to provide a sufficient general interface to capture a wide variety of constraint types, while at the same time provide enough specificity that developers and requirements engineers could grasp how to use the tool. Striking this balance was not easy.

Scatter provides a graphical modeling tool to address this challenge and allow developers to express requirements. To specify non-functional requirements, users drag-and-drop requirements from the palette onto components. The child requirement elements of a component specify the non-functional requirements that must be satisfied by a device's resources. Each requirement has a *Name*, *Type*, and *Value* attribute associated with it:

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

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

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

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

A key challenge in a pervasive environment is that variant selection must take into account business and context-based. For example, on a train, the first-class and coach-class cabins may offer different meal services. In coach, travelers may be able to pre-order food via a mobile phone application, but still must physically go and pickup the food. In first-class, however, train staff may be required to deliver food orders to a traveler's seat. For first class, therefore, a variant that provides a component for notifying the ordering system of where the traveler is sitting may be required while it would not be required in coach. Cabins may also offer different meal selections or meal prices, in which case the variant selection must account for the location-based rules when selecting which menu to deliver with the ordering service. This train variant selection scenario can be seen in Figure 3.

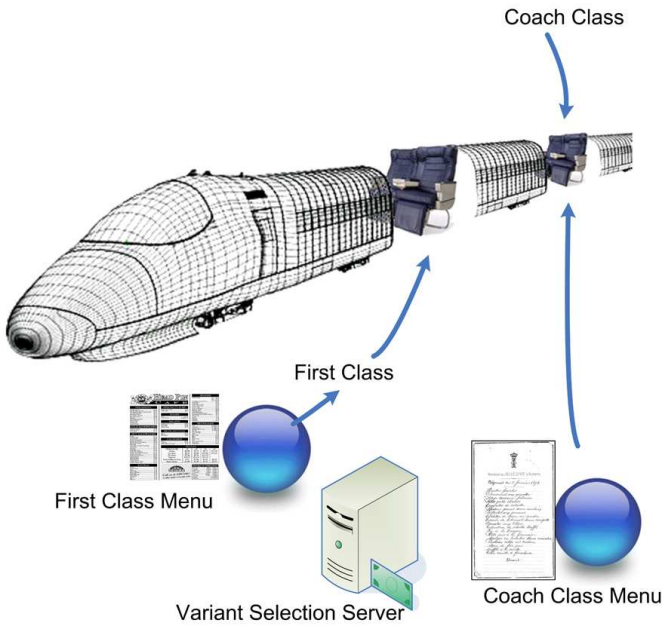


**Figure 3: Cabin Class Constraints for Train Menu Variant Selection**

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

Scatter employs a strategy, similar to the Bridge pattern [14],

that allows it to capture and solve a wide variety of constraint types. The interface and semantics for constraints, remains constant, while the datasources and datatypes over which the constraints operate is variable. For example, a modeler could specify the constraints:

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

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

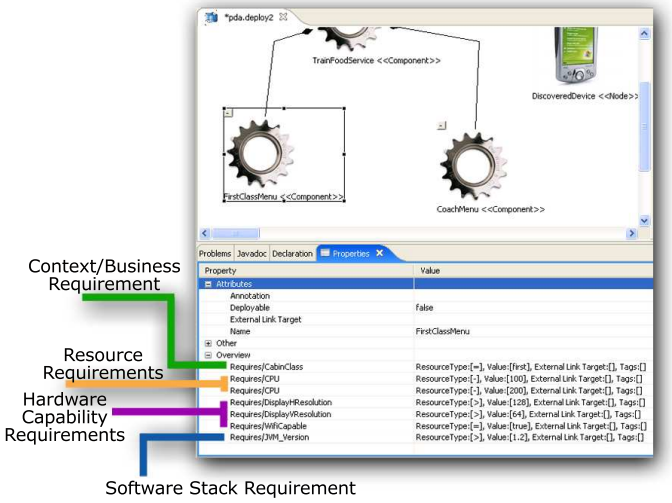


**Figure 4: Capturing Mixed Non-functional Requirement Types in Scatter**

The restrictions imposed by the specification format are only on the types of comparisons that can be done and not on the data that the comparison is based upon. This freedom in constraint specification allows Scatter's variant selection to incorporate a large array of datatypes that a device discovery service could provide. This setup allows other services to pre-process the data used by the variant selector and thus allow it to operate on very complex data sets.

For example, context processors based on GPS or RFID can calculate a device's position or type and correlate cabin class. Business-rule engines can calculate customer priorities and provide business analysis. For non-functional requirements, we have found that these six types of primitive constraint types ($>$, $-$, et al) when combined with front-end data preprocessing, cover most situations. Scatter's architecture thus allows the complex portions of variant selection, the constraint solvers to remain constant, while still allowing new datatypes to be incorporated. For scenarios where other types of constraints are needed, Scatter provides mechanisms for plugging in new types and solvers, although we have found that this capability is rarely needed in practice.

## 2.3 Transforming Requirments into Native Constraint Solver Formats

Scatter is based on the open-source Generic Eclipse Modeling System (GEMS) [33, 34], which is a part of the Eclipse Generative Modeling Technologies (GMT) project. GEMS provides a convenient way to define the metamodel, *i.e.*, the visual syntax of the modeling language. Based on the metamodel, GEMS automatically generates a graphical editor that enforces the grammar specified in the metamodel. To facilitate code generation, GEMS also provides an infrastructure for model traversal and event listening that can be used in conjunction with other MDD tools, such as ATL [12] or open Architecture Ware (oAW) [1].

Scatter extends our previous work using Role-based Object Constraints (ROCs) and Model Intelligence [29, 32]. ROCs uses Subject-oriented design [5] as the abstraction for viewing the PLA model. Subject-oriented design provides an abstraction that decomposes designs into design subjects, which are units that encapsulate a single coherent piece of functionality. The design subject abstraction helps to more tightly align system design and requirements. The PLA metamodel is viewed as a set of model entites or design subjects and the role-based relationships between them.

In Scatter, *Device*, *Component*, *Requirement*, and *Resource* are entities. Each entity may participate in multiple role-based relationships. For example, a *Requirement* may play the 'requirements' role for a *Component*. A role-based relationship may also represent properties or attributes of an entity, such as the name of a *Component*. The GEMS ROCs infrastructure automatically transforms instances of models created in Scatter into Prolog knowledge bases that use predicate names based on these relationship roles.

After a developer creates a model of a PLA in Scatter, the underlying GEMS ROCs infrastructure transforms the model into a Prolog knowledge base, which KB is based on domain-specific predicates, derived from the subject-oriented design view of the Scatter metamodel. For example, the predicate statements:

```
self_type(1, component).
self_name(1, 'FirstClassFoodReservationService').
self_requires(1, [2]).
self_type(2,requirement).
self_name(2,'CabinClass').
self_type(2,'=').
self_value(2,'first').
```

would be generated from a PLA component named *FirstClassFoodReservationService* that had a non-functional requirement that it only be placed on devices with *CabinClass* = *first*. This transformation from the visual modeling domain to the Prolog knowledge base happens behind the

scenes and does not require user intervention. This Prolog knowledge base is then leveraged by the Prolog inference engine and the Constraint Logic Programming Finite Domain (CLP(FD)) solver [18, 31] based variant selection engine.

As we discussed in Section 2.1, users can augment the basic AND and OR graphical PLA composition rules with complex rules using the Prolog domain-specific predicates. Developers add invocations of their custom composition rules in special rules for calculating the dependencies of a component and the feasibility of deploying a specific component to a device. For example, assume that a developer wants to ensure that a Wifi API component is deployed with all lower level Wifi Drivers. The developer knows that Wifi drivers have a *Requirement* named '802.11b'. A developer could specify the rule:

```
get_dependencies(Component,Dependencies) :-
     self_name(Component,'WifiAPI'),
     findall(Dependency,
           (self_requires(Dependency,Reqs),
            member(Req,Reqs),
            self_name(Req,'802.11b'))
            Dependencies)).
```

which would ensure that all other Components that with a 802.11b *Requirement* were listed as the Wifi API's dependencies. This predicate logic composition language gives developers the full power of the Prolog language for specifying rules. Although Scatter translates PLA models and device signatures into a Prolog knowledge base, developers need not expose themselves to it unless needed.

## 2.4 Discovery and Device Signatures

Another challenge to implementing an automatic variant selection engine is decoupling device discovery and variant deployment from the discovery and deployment engines. The variant selector should be independent of the mechanism used for discovery so that it can be reused in different contexts. Decoupling the variant selector from the discovery mechanism requires providing a method for outside services to remotely update the knowledge base utilized by the selector. Moreover, the discovery service may not be implemented in the same programming language or understand the specifics of the knowledge base format.

Scatter focuses on providing the engine for deducing a valid variant given a model of a PLA and a target infrastructure. Scatter allows the configuration of discovery and device profiling mechanisms into it to provide the infrastructure descriptions over which it operates. The characterization schemas that are used are also configurable. Transformations can be plugged-into Scatter to interpret new schema types. This architecture can be seen in Figure 5.

Scatter exposes a SOAP-based web service and a CORBA remoting mechanism for remotely communicating device characterizations as they are discovered. These remoting services provide a high-level API that allows the discovery service to report back the device name and the resources available on the device. The service then transforms these reports into

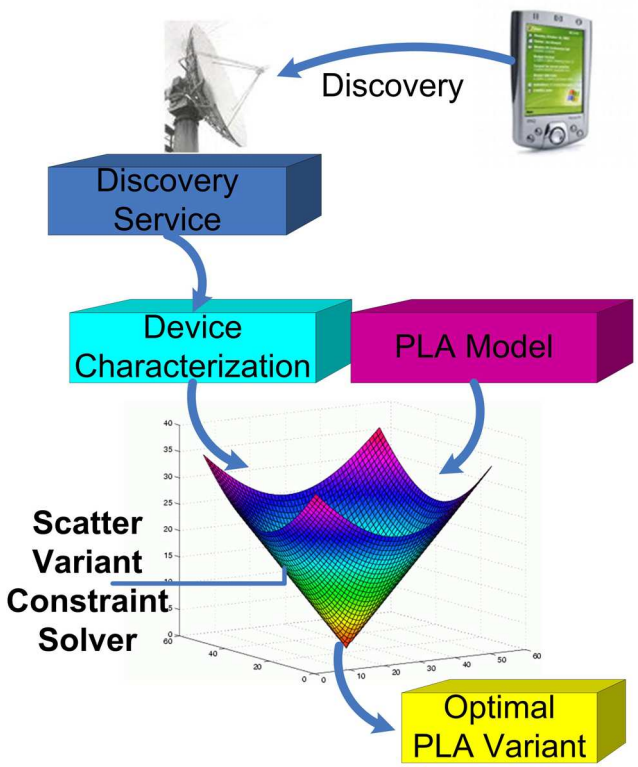


**Figure 5: Scatter Integration with a Discovery Service**

the native knowledge base format so that they can be combined with the PLA model and operated on by the Scatter variant selection solver. This transformation on the variant selector side from the high-level remote updates to the specific format of the knowledge base prevents the discovery mechanisms from becoming tightly coupled to the knowledge base format. As Scatter's knowledge base format evolves, existing mechanisms for discovery can be reused.

Scatter's remoting services also provide the ability to invoke the variant selection solver. A discovery service can remotely update the Scatter model and then invoke its variant selector to find a software configuration to push to the device. After a particular device signature has had a variant found for it, it can be cached and reused when devices with identical or compatible signatures are discovered. This capability significantly reduces time and space overhead by only invoking the Scatter solver when new device signatures are encountered.

# 3. THE SCATTER VARIANT SELECTOR

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

• **Unknown device signatures.** Although devices may share common communication protocols and resource description schemas, a variant selection service will not know all device signatures ahead of time. This ensure that variant selection must be able to run efficient and online to a valid variant when a new device is encountered. Moreover, devices

may possess wildly differing signatures. On the one extreme, a laptop may be carried onto a train with a relatively powerful Intel Core Duo processor and a gigabyte or more of RAM. On the other extreme, a Treo mobile phone may be discovered with a 312mhz XScale processor and 64mb of RAM. A variant selector must be able to handle these diverse device descriptions.

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

• **Limited selection time.** A variant may need to be selected very rapidly. On a train, 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, however, 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 devices signatures *a priori* and the need for optimization, achieving quick selection times becomes even more difficult.

To address these challenges, Scatter provides an automated variant selector that leverages Prolog's inferencing engine and a CLP(FD) constraint solver. The Scatter solver uses a layered solving approach to solving to help reduce the combinatorial complexity of satisfying the resource constraints, a form of bin-packing an NP-Hard problem, by pruning the solution space using the PLA composition rules and the local non-functional requirements. This layered pruning helps address the speed challenge outlined above and enables more efficient solving. As shown in the Section 5, this layered pruning can significantly improve variant selection performance.

## 3.1 Layered Solution Space Pruning

Initially, the variant solution space contains many millions or more possible component compositions, as seen in step 1 of Figure 6. Solving the resource constraints is thus time consuming. To optimize this search, Scatter first prunes the solution space by eliminating components that cannot be deployed to the device because their non-functional requirements, such a JVMVersion or CabinClass, are not met. After pruning away these components, Scatter evaluates the PLA composition rules to see if any components can no longer be deployed because one of their dependencies has been pruned in the previous step. After pruning the solution space using the PLA composition rules, the resource requirements are considered, as shown in step 2 of Figure 6. After solving the resource constraints, Scatter is left with a drastically reduced number of deployment solutions to select from. At this point, if there is more than one valid

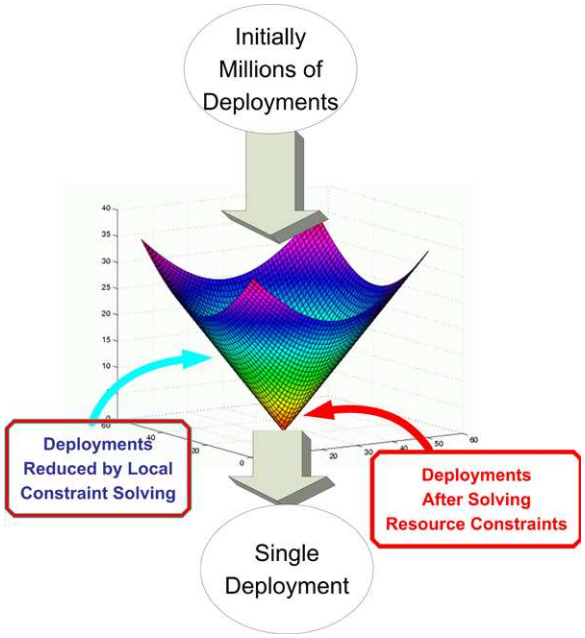


**Figure 6: Scatter's Layered Deployment Solving Approach**

variant remaining, Scatter uses a branch and bound algorithm to iteratively try and optimize a developer-supplied cost function by searching the remaining valid solutions.

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

canDeployTo(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),canDeployTo(D,Device)).
```

For each *Component*, the rule 'canDeployTo' is invoked to determine deployment feasibility. This rule also simultaneously tests the feasibility of deploying a component based on its dependencies. The last invocation in the rule checks to ensure that all of the components that current component depends on can also be deployed to the *Device*. If any of the dependencies cannot be deployed, the component cannot be deployed. 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.

## 3.2 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. 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* requirements, $C_i(R)$, when multiplied by their prescence variables and summed cannot exceed the available amount of that resource, $Dvc(R)$, on the *Device*.

If the presence variable is assigned 0, indicating the component is not in the variant, the resource demand by that component falls to zero. The constraint $\sum C_i(R) * DC_i < Dvc(R)$ is created to enforce this rule. 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 selection composition constraint that allows exactly $N$ components from the dependencies to be present. The selection operator creates the constraint: $C_i > 0 \rightarrow \sum C_j = N$. The standard XOR dependencies from the metamodel are modeled as a special case of selection where $N = 1$. Finally, the solver supports component exclusion. For each *Component* $C_n$ that cannot be present with $C_i$, the constraint $C_i > 0 \rightarrow C_n = 0$ is created.

To support optimization, a variable $Cost(V)$ is defined using the user supplied cost function. For example, $Cost(V) = 1000 - C_1 + C_2 + C_3 \ldots C_n$ could be used to specify the cost of a variant as being lower when more Components are contained within it. This cost function would attempt to maximize the number of components deployed within the resource and PLA composition limits. Once the requirements have been translated into CLP(FD) constraints, Scatter asks the CLP solver for a labeling of the variables that maximizes or minimizes the variable $Cost(V)$, which allows the variant selector to choose components that not only adhere to the compositional and resource constraints but that maximize the value of the variant. The user therefore supplies a fitness criteria for selecting the best variant from the population of valid solutions.

# 4. AUTOMATED REPAIR NEGOTIATION WITH MOBILE DEVICES

When devices enter a particular context and wish to receive a customized variant, the device may be in a state such that no valid variant can be found. For example, if a device is running several other applications and cannot provide enough memory to support even the most minimal of variants from the PLA, the solver will not be able to make a selection. In these cases, it is crucial that the variant selection mechanism provide a method of diagnosing the failure and negotiating resource or configuration changes on the device to make variant selection possible. We call this process of negotiating with a device to alter its configuration, *repair negotiation*.

The first challenge of providing automated repair negotiation is creating a way of diagnosing the cause of a variant selection failure. With numerous complex composition rules guiding the selection process, it is extremely hard to figure out *why* there is no valid variant of the PLA and *how* to repair the device or relax the PLA's non-functional requirements to overcome the problem. Simply failing to select a variant and not providing an explanation would leave the reasoning of the underlying cause to the device user, without any hints on possible modifications (such as resource expansions) to make it work. In these situations, deducing the cause of selection failure could be as hard as finding a valid variant manually.

A key question was what type of feedback should be provided to device users. One approach we evaluated was marking non-functional requirements, such as CPU demand, that could not be satisfied and then returning a list of failed requirements as an error message to the device. We found this approach unsatisfactory for the following reasons:

- For global constraints, such as resource constraints, the overall state of the device determines whether or not the constraint succeeds. In the variant selection, if the device does not provide sufficient resources to host all of the components required for a variant, it is not necessarily a single requirement that is causing the problem. Marking the first requirement that could not be met would not make sense since different packing or selection search orders could result in different requirements marked as the cause of failure.
- Even if the cause of the failure was marked in some manner, users would still need to manually determine how to modify the device from its present state to make it compliant with the failed constraints. Although fixing the problem, by taking an action such as shutting down other applications, might appear trivial when the failing constraint was identified, changing the device state could have unforseen affects on the other domain constraints. Again, manual approaches do not scale for these types of constraint satisfaction problems.

To overcome these problems, Scatter adopts a strategy of allowing a device to express a series of state modification operations that it is willing to undertake as part of the discovery process. We call these modifications that could be performed *repair operations*. For example, a repair operator

*IncreaseCPUPower* could be exposed by the device, indicating it is willing to shutdown other applications to free up CPU cycles. These repair operations can then be taken into account by the variant selection solver to find a variant if no suitable one is available in the device's current state.

Repair operations can include an amount by which a resource can be expanded or they can be specified without amounts indicating that it is the responsibility of the variant selector to deduce the minimal amount that the resource should be expanded when selecting a variant. For example, if no variant could be found and the device had published the repair operations *IncreaseCPUPower* and *IncreaseRAM*, with no expansion amounts, the constraint solver could first try to find a variant normally and if none was found to calculate the smallest variant that could be deployed to the device based on the other non-functional requirements. After finding the minimal variant, the variant, along with the required CPU and Memory could be reported back to the device, as seen in Figure 7.


**Figure 7: Negotiating Repair Operations with a Device**

To achieve this repair functionality, for each modifiable resource Rm, that reports a value, $Up(R_m)$, by which it can be expanded, the constraint $\sum C_i(R_m) * DC_i < Dvc(R_m)$ is replaced by $\sum C_i(R_m) * DC_i < Dvc(R_m) + Up(R_m)$. The solver then simply reruns the variant selection process with the expanded resource values.

Repairs without values specified are more complex. For repair operations on resources that do not express expansion values, the constraint $\sum C_i(R_m) * DC_i < Dvc(R_m)$ is replaced by $\sum C_i(R_m) * DC_i = D_m(R_m)$. A further constraint, $\sum D_m(R_m) = UpTotal$ is also added to capture the total value of the modifiable resources consumed by a variant. The cost function is modified to prioritize variants that consume less of the modifiable resources. This prioritization is acheived by changing the cost function, $Cost(V)$ to $CostM(V)$, where $CostM(V) = Cost(V) - W_m * UpTotal$ (assuming that we are attempting to maximize cost). The term $W_m$ is a configurable constant used to weight the importance of minimizing the size of the changes that the device will have to make.

Finally, since we have added the terms $Wm*UpTotal$, which causes the solver to prefer smaller solutions (and possibly no solution), we add a constraint to guarantee that the solver selects a variant, if it is possible. The new constraint is $\sum DC_i > 0$. This constraint guarantees that at least one component is present, *i.e.*, that a variant is selected if possible.

Scatter's repair operation mechanism provides the following characteristics that make automated device negotiation possible:

- diagnosing and providing a meaningful explanation of variant selection failure to a user is not required
- devices and users control repair by expressing only modifications that they are willing or capable of making
- repair is automated and does not require error-prone user intervention
- repair can be optimized to according to a cost function

Scatter's repair mechanism removes any necessity for the user to take part in diagnosing a failure. This automation is particularly important when device users may have varying skill levels and wind up requiring human assistance from an employee, which is expensive. Even though repair is automated, device users can still control modifications to their device by setting allowed modification preferences, such as whether or not applications can be closed, third-party software downloaded, or memory reservations increased. Finally, this mechanism allows developers of PLAs to provide criteria for choosing the best variant in the face of failures, which is important when costs are associated with variants.

## 5. SCATTER PERFORMANCE RESULTS

A key question is how fast Scatter performs and whether or not online variant selection is possible. To test Scatter's performance, we developed a series of increasing larger and larger PLA models to evaluate solution time. We also tested how various properties of PLA composition and local non-functional constraints affected the solution speed. Our test were performed on an IBM T43 laptop, with an 1.86ghz Pentium M CPU and 1 gigabyte of memory.

Before continuing further into the results, it is worth noting that optimization and satisfaction of resource constraints is an NP-Hard problem. It is always possible to play the role of an adversary and craft a problem instance that provides horrendous performance [8]. Constraint satisfaction and optimization algorithms, however, often perform well in practice despite their theoretical worst case performance. One challenge when developing a PLA that needs to support online variant selection is ensuring that the PLA does not induce worst-case performance of the selector. We therefore attempted to model realistic PLAs and to test Scatter's performance and better understand the effects of PLA design decisions.

## 5.1 Pure Resource Constraints

First, we 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 and thus 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. Initially, our device was allocated 100 units of CPU and 16 megabytes of memory. Scatter's performance results on this model can be seen in Figure 8. As can be seen from the large jump in time from the time

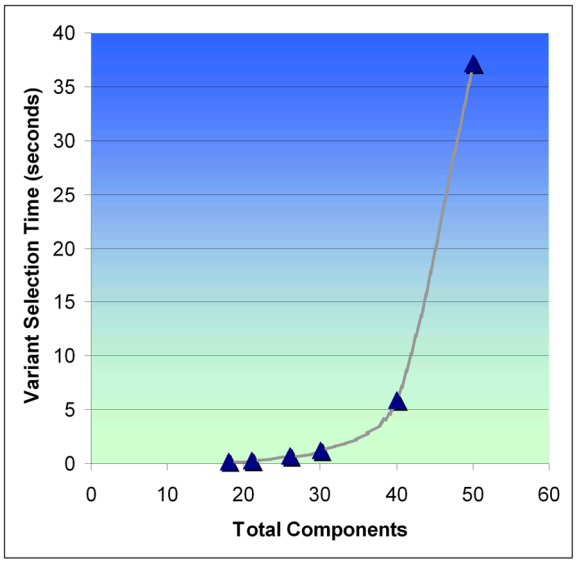


**Figure 8: Scatter Performance on Pure Resource Constraints**

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

## 5.2 Testing the Effect of Limited Resources

Next, we investigated how the tightness of the resource constraints affected solution time. We incrementally increased the available CPU on the modeled device from 100 to 2500 units for the 50 Component model. The results can be seen in Figure 9.

As shown in Figure 9, expanding the CPU units from 100 to 500 units dramatically dropped the time required to solve for a variant. Moreover, after increasing the CPU units to 2,500, there was no increase in performance indicating that the tightness of the CPU resource constraints were no longer the limiting bottleneck.

We next proceeded to increase the memory on the device while keeping 2,500 units of CPU. The results are shown in Figure 10. Doubling the memory immediately halved the solution time. Doubling the memory again to 128 megabytes provided little benefit since the initial doubling to 64 megabytes made deployment of all of the components possible. What we had hypothesized initially and was shown is that the so-

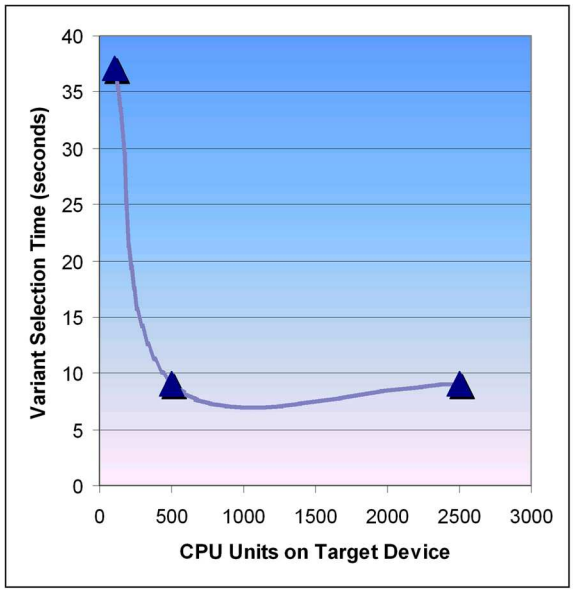


**Figure 9: Scatter Performance as CPU Resources Expand on Device**

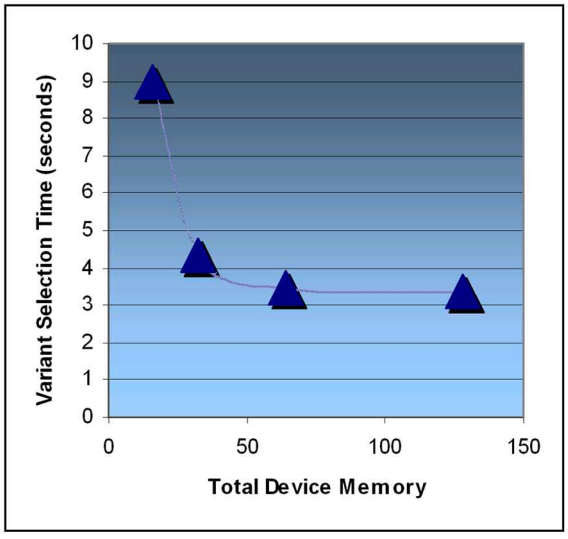


**Figure 10: Scatter Performance as Memory Resources Expand on Device**

lution speed when pure resource constraints are considered is highly dependent on how tight the resource constraints are.

## 5.3 Testing the Effect of PLA Composition Constraints

Our next 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 5.1, we used our 50 component model as the initial baseline. First, we began by constructing a tree of dependencies that tied 10 components into an application set that led the root of the tree, the reservation service, to only be deployed if all children where deployed. Each level in the tree depended on the deployment of the layer beneath it. The max depth of the tree was 5. We continued to add trees to the model to see the effect. The results are shown in Figure 11.

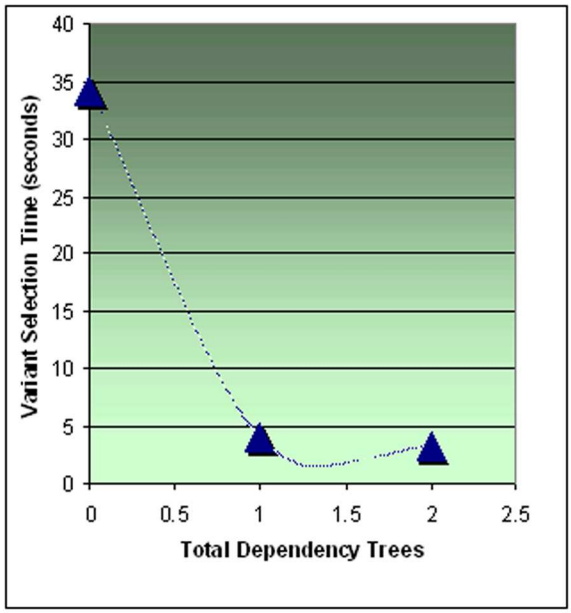


**Figure 11: Scatter Performance as PLA Dependency Trees are Introduced**

## 5.4 Results Analysis: Mobile PLA Design Strategies

As can be seen from the results, PLA dependencies can significantly reduce the time taken by Scatter to find a variant. Based on the results we collected, we devised a set of mobile PLA design rules to help improve variant selection performance. The remainder of this section presents the lessons we have learned from our results.

#### 5.4.0.1 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 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 exploited the relative lack of variation in packaging of a PLA variant.

#### 5.4.0.2 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 changes input parameters, the disk footprint of various classes of variants can be used to greatly prune the solution space. If a PLA with 50 components is composed of 5 Java Archive Resource (JAR) files, although there are a large number of ways that the PLA can be composed, there are relatively few valid combinations of the JAR files.

More than likely many variants will require common sets of these JAR files with various footprints. Variants can then 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.

#### 5.4.0.3 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.

#### 5.4.0.4 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, 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 likely hood of unanticipated variants that fail.

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

## 6. RELATED WORK

In [24], Mannion et al presents a method for specifying PLA compositional requirements using first-order logic. The validity of a variant can then be validated by determining if a PLA satisfies a logical statement. Scatter's approach to PLA composition rule specification expands on this idea by specifying PLAs as compositions using AND and XOR of components. Scatter also extends the work in [24] by including the evaluation of non-functional requirements not related to composition in validation. 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 thereom proving is enhanced in [25], this approach does not provide a requirements-driven optimal variant selection engine like Scatter.

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 is 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.

Many complex modeling tools are available for describing and solving combinatorial constraint problems, such as those presented in [27, 9, 30, 4, 13]. These 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. With these tools, developers must also re-invent the repair architecture and remoting mechansims provided by Scatter. Finally, unlike in Scatter, complex transformations must be created by the PLA developer to map the output of these tools back to the PLA.

Chisel [20] 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 two key reasons. First, 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. Second, 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. Furthermore, Scatter's optimization algorithms provide guaranteed results while Chisel's manually produced policies give no guarantee of solution quality. Finally, Scatter does not assume that the functionality of the variants is identical and can thus handle the selection of multiple variants to deploy.

## 7. CONCLUDING REMARKS

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

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

- PLA composition and non-functional requirements can be used to efficiently prune the variant selection space and provide good performance. There are many patterns of requirements specification that can be used to optimize a PLA for automated variant selection. In future work, we intend to further explore these patterns.
- Although Scatter can automate variant selection, it works best when a PLA is crafted with performance in mind. An arbitrary PLA may or may not allow for rapid variant selection. PLA's that will be used in conjunction with an automated variant selector should be carefully constructed to avoid poor performance.
- By using the Bridge pattern [15], the Scatter tool described in Section 2.1 can handle a wide variety of constraint types, such as context or business based constraints, while still providing a concrete method of mapping constraints to a solver.
- A key challenge of providing online selection is obtaining and characterizing device and context information. These challenges are handled by other efforts, such as CC/PP described in Section 1.
- More work must be done to understand how to merge and integrate the various information sources that will provide device characterizations. Device characterizations may come from customer databases, discovery

services, and location services. Finding the right transformations to correlate and utilize these diverse information streams is important to provide customized and correct variant selection.

- Resource and configuration negotiation with a device can be viewed as a repair operation and automated and optimized by a constraint solver. Automating repair, helps to eliminate manual intervention when a suitable variant cannot be found.

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

# 8. REFERENCES


[1] The openarchitectureware. http://www.eclipse.org/gmt/oaw.

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

[3] D. Benavides, P. Trinidad, and A. Ruiz-Cortes. Automated Reasoning on Feature Models. *17th Conference on Advanced Information Systems Engineering (CAiSE05, Proceedings), LNCS*, 3520:491–503, 2005.

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

[5] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Subject-Oriented Design: Towards Improved Alignment of Requirements, Design, and Code. *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 325–339, 1999.

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

[7] P. C. Clements and L. Northrop. *Software Product Lines Practices, and Patterns*. Addison-Wesley, 2001.

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

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

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

[11] 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 Technologies and Systems*, March 2003.

[12] M. Del Fabro, J. Bzivin, and P. Valduriez. Weaving models with the eclipse amw plugin. In *Eclipse Modeling Symposium, Eclipse Summit Europe 2006*, 2006.

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

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[16] R. Hall, D. Heimbigner, and A. Wolf. A Cooperative Approach to Support Software Deployment Using the Software Dock. *Proceedings of the 21st International Conference on Software Engineering*, pages 174–183, 1999.

[17] J. Indulska, R. Robinson, A. Rakotonirainy, and K. Henricksen. Experiences in Using CC/PP in Context-Aware Systems. *LNCS*, 2893:224–235.

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

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

[20] 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.

[21] G. Klyne, F. Reynolds, C. Woodrow, H. Ohto, J. Hjelm, M. Butler, L. Tran, et al. Composite capability/preference profiles (cc/pp): Structure and vocabularies. *W3C Working Draft*, 15, 2001.

[22] O. Lassila, R. Swick, et al. Resource Description Framework (RDF) Model and Syntax Specification. *W3C Recommendation*, 22:2004–03, 1999.

[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] 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.

[25] 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.

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

[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] 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.

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

[30] 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.

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

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

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

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

[35] 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.