Evaluation of Frame- and Feature-based Software Product Line Tools from the Viewpoint of Mass Customization by End Users

Gaurav Gahalaut  
*University of Jyväskylä, gaurav.gahalaut@jyu.fi*

Timo Käkölä  
*University of Jyväskylä, timokk@jyu.fi*

Follow this and additional works at: [http://aisel.aisnet.org/amcis2010](http://aisel.aisnet.org/amcis2010)

Recommended Citation
[http://aisel.aisnet.org/amcis2010/394](http://aisel.aisnet.org/amcis2010/394)
Evaluation of Frame- and Feature-based Software Product Line Tools from the Viewpoint of Mass Customization by End Users

Gaurav Gahalaut  
University of Jyväskylä  
40014 University of Jyväskylä, Finland  
gaurav.gahalaut@jyu.fi

Timo Käkölä  
University of Jyväskylä  
40014 University of Jyväskylä, Finland  
timokk@jyu.fi

ABSTRACT
Customers expect Information and Communications Technology (ICT) platforms and applications to deliver services customized to their needs. Software product line (SPL) paradigm uses platforms and variability management to develop mass-customizable software applications. The paradigm necessitates effective software tools to manage platform and application artifacts and traceability and variability information. This paper constructs a comprehensive but lightweight tool evaluation framework and uses it to evaluate two tools, XML-based variant configuration language (XVCL) and FeaturePlugin – a feature modeling plug-in for Eclipse Integrated Development Environment. The paper analyzes the capabilities of the tools for enabling the mass customization of software applications by the end users performing complex workflows. Both the XVCL and FeaturePlugin tool envisage more efficient software system development by means of reusability, support for abstraction, and configuration mechanisms. Future research is needed to refine and validate the evaluation framework in the context of other types of SPL tools.

Keywords
Software product line engineering, mass customization, frame-based tools, feature-based tools.

INTRODUCTION
Customers expect Information and Communications Technology (ICT) platforms and applications to deliver services that are customized to their needs. Customization is expensive and error-prone if it requires customer-specific systems to be built in projects paid by the customers. However, software product line engineering (SPL) helps mass-customize software products (Northrop and Clements, 2007). It is a paradigm for developing software products and software-intensive systems and services faster, at lower costs, and with better quality and higher end-user satisfaction than is possible through the engineering of single systems (Pohl et al., 2005). It differs from the engineering of single systems in two primary ways (Käkölä and Dueñas, 2006):

1. It needs two distinct development processes: domain engineering and application engineering. Domain engineering defines and realizes the commonality and variability of the software product line, thus establishing the common software platform for developing high-quality applications rapidly within the line. Application engineering derives specific applications by strategically reusing the platform and by exploiting the variability built into the platform.

2. It needs to explicitly define and manage variability. During domain engineering, variability is introduced in all domain artifacts such as domain requirements, architectural models, components, and domain test cases. It is exploited during application engineering to derive applications mass-customized to the needs of different customers and markets.

Variability models define the variability of a software product line. Variation points and variants are the central concepts of a variability model. Pohl et al. (2005, Ch. 4) present a comprehensive meta-model for orthogonal variability modeling that enables product line engineers to capture variation points, that is, the items that vary; variants defining how the variable items can vary; and constraints between variants, between variants and variation points, and between variation points. External variability is the variability of domain artifacts that is visible to customers. Internal variability is the variability of domain artifacts that is hidden from customers. Variation points corresponding to internal and external variability are called, respectively, internal and external variation points (Pohl et al., 2005).
Most software product line engineering research takes a provider viewpoint, that is, when and how the providers should build product lines using the SPL paradigm. This paper focuses on the software product line engineering paradigm also in the context of end users conducting complex workflows to mass customize software systems critical for their work systems. Some approaches provide mass customization at end user level. For example, Rabiser et al. (2009) presents an approach to mass-customize the product line at three levels, that is, providers, (ICT-organizations of) customer organizations, and end-users can all be responsible for customization. Their approach helps various stakeholders at the three levels to select features that best meet their diverse needs. For example, the stakeholders can eliminate unnecessary features from applications that otherwise would be overloaded with features, thus improving usability.

Software product line engineering involves higher levels of abstraction than single-system engineering partly because the platforms require substantial investments, have long life cycles, and have to be generally applicable to a wide range of products. Without appropriate abstractions such platforms cannot be built and variability cannot be managed effectively (Kääkölä, 2010).

A number of approaches and tools have been proposed to support software development using the software product line paradigm. The XML-based variant configuration language (XVCL) (Zhang and Jarzabek, 2003), FeaturePlugin – the feature modeling plug-in for Eclipse IDE (Antkiewicz and Czarnecki, 2004), and Feature House (Apel et al., 2009) are some of the best-known tools.

SPL tools are still under rapid development and have typically not reached maximum levels of maturity and robustness. Therefore, they should not be deployed in production use before testing and evaluating them in depth. This paper constructs an evaluation framework for SPL tools and uses it to analyze XVCL and FeaturePlugin to find out (1) the effectiveness of these tools from the viewpoints of various stakeholders and (2) specifically their ability to help end users mass-customize applications. A purpose of the framework is to help organizations screen and rank SPL tools in a lightweight manner, for example, by using published product specifications and feature lists. The highest ranking tools can then be evaluated in depth, for example, by using them in pilot projects.

To analyze the effectiveness of the tools, e-commerce systems are taken as an example domain. They can be viewed as special cases of a work system in which a purchaser can use the seller’s website to obtain product information and to perform purchase transactions (Alter, S, 2010). In this domain, an information system is an integral part of the work system and participants of the information system are participants of work systems. The work system might not be able to achieve its objectives without the information system and, at the same time, the information system will be of less importance without the work system (Alter, S, 2008).

E-commerce systems enable users to sell and purchase products online and to use payment, product information, and other services (Chua et al., 2005). The stakeholders of e-commerce systems can be categorized into direct and indirect stakeholders. Direct stakeholders include sellers, purchasers, advertisers, and the executive management personnel. Indirect stakeholders include competitors, media personnel, and researchers. E-commerce systems enable end users to perform workflows. For example, when an end user selects a payment option, the system performs a workflow, including tasks such as “obtain credit card information”, “authorize credit card”, “approve order”, “reject order”, and “ship order” (Hung and Zou, 2007). E-commerce product lines consist of optional and mandatory components such as online catalogues, payment systems, and registration components (Hernández et al., 2009). In this paper, a reference architecture is designed for the catalogue of an e-commerce product line. The support of XVCL and FMP tools for implementing the reference architecture is then analyzed.

The paper is organized as follows. Section “Evaluation Framework for Software Product Line Tools” introduces the tool evaluation characteristics. Section “A Common Example” describes the example used for tool evaluation. Section “Evaluated Tools” presents and evaluates the tools. Finally, conclusions and ideas for future research are discussed.

EVALUATION FRAMEWORK FOR SOFTWARE PRODUCT LINE TOOLS

To construct a comprehensive evaluation framework for SPL tools, relevant literature (Chmura and Crockett, 1995; Coplien, 1998; Krueger, 2002; Krueger, 2007; Ommering, 2004; Ponnekanti et al., 2003) has been analyzed to elicit relevant evaluation characteristics and to determine the underlying motivations for selecting the characteristics (Table 1). One motivation is that SPL tools should help customer organizations and end users to have a central role in mass customization but this rarely happens in practice. Our experiences during the development of a prototype also helped to determine evaluation characteristics. Enabling mass customization by all major stakeholders, including end users, is a challenging task and requires future SPL tools to have many characteristics (Table 1). In addition to the characteristics presented in Table 1,
SPL tools should provide adequate speed and be easy to use, available whenever needed, and documented comprehensively (Islam et al., 2010).

Combination of mass customization and platform-based development promotes software reuse in SPL engineering. Reusing artifacts also helps improve maintainability of the systems (Coplien, 1998): “The same design techniques that lead to good reuse also lead to extensibility and maintainability over time.”

Variability management deals with software configuration both over time and over space, whereas the traditional single systems engineering only deals with software configuration over time (Krueger, 2002). SPL tools can manage variability by providing mechanisms to specify variation points, variants, variability constraints, and to categorize variability as internal or external variability. In application engineering, variability is bound to build variants. The moment of variability resolution is called the binding time of variability. For increasing flexibility, it is often desired that variability binding should happen at late stages of realization or when end users are already using the application (Ommering, 2004).

Configuration is the instantiation of predefined variation within a development artifact during domain or application engineering. Configuration management helps manage different versions of development artifacts evolving over time. Product configuration can be manual and automated. In manual product configuration, domain artifacts composing the platform are arranged manually to derive products. In automated product configuration, products are derived automatically by arranging the domain artifacts based on preferred selection of variants at each variation point (Krueger, 2007).

<table>
<thead>
<tr>
<th>CID</th>
<th>Characteristic</th>
<th>Description / Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Support for reusability</td>
<td>Does the tool provide support for reusability?</td>
</tr>
<tr>
<td>2</td>
<td>Support for variability management</td>
<td>Does the tool provide support for variability management?</td>
</tr>
<tr>
<td>3</td>
<td>Support for variability binding</td>
<td>When is variability bound? For example, during run time, build time, or design time?</td>
</tr>
<tr>
<td>4</td>
<td>Support for programming languages</td>
<td>Is the tool independent from programming languages?</td>
</tr>
<tr>
<td>5</td>
<td>Support for automated product configuration</td>
<td>Does the tool support automated product configuration?</td>
</tr>
<tr>
<td>6</td>
<td>Support for CASE tools</td>
<td>Is the tool integrated with any UML-based CASE tools?</td>
</tr>
<tr>
<td>7</td>
<td>Support for popular IDEs</td>
<td>Does the tool work with any popular IDEs?</td>
</tr>
<tr>
<td>8</td>
<td>Support for portability</td>
<td>Can applications be ported to multiple software and hardware platforms easily?</td>
</tr>
<tr>
<td>9</td>
<td>Support for modularity</td>
<td>Can applications be developed modularly?</td>
</tr>
<tr>
<td>10</td>
<td>Support for future changes</td>
<td>Is the tool flexible enough to respond to future improvements or changes?</td>
</tr>
<tr>
<td>11</td>
<td>Support for generalizability</td>
<td>Does the tool support generic artifacts (e.g., the creation of domain artifacts that can be reused across multiple domains)?</td>
</tr>
<tr>
<td>12</td>
<td>Support for dynamic runtime evolution</td>
<td>Does the tool support dynamic runtime evolution?</td>
</tr>
<tr>
<td>13</td>
<td>Support for different levels of abstraction</td>
<td>Does the tool provide different levels of abstraction?</td>
</tr>
<tr>
<td>14</td>
<td>Support for various stakeholders</td>
<td>Does the tool support various stakeholders?</td>
</tr>
</tbody>
</table>

Table 1. Evaluation framework for SPL tools
SPL tools should be independent from programming languages to give developers flexibility to choose any programming language for development. Computer Aided Software Engineering (CASE) tools help to automate major parts of software development. Requirements management and UML modeling tools are examples of CASE tools. SPL tools also should support popular Integrated Development Environments (IDE) such as Eclipse and Netbeans to facilitate software development activities such as variability modeling or model transformation.

Platform and application portability requirements mean that platforms and applications must be portable to different hardware and operating system environments with minimal design and programming effort. To support future incremental evolution, platforms need to be extensible and designed modularly (Ponnekanti et al., 2003). Evolution of a SPL is typically a more complex problem than the evolution of a single system due to configuration requirements and possible product derivations (Loughran et al., 2004). If end users are to take more central roles in mass customization, dynamic runtime evolution becomes necessary for the SPL. Once a SPL becomes operational for end users, the SPL has to be so robust that the binding of predefined variability by the IT management and end users does not require interaction with developers to conduct business processes. Of course, end users and other stakeholders must be able to communicate to the SPL vendor for improving the reference architecture, the provided feature set, and the support for new quality requirements, so the SPL can evolve to meet stakeholder needs also on the long term.

Tools may support different levels of abstractions. Working at a higher level of abstraction increases productivity because there is less program code to write in order to provide the same functionality (Verelst, J., 2004). But from the end users’ viewpoint, SPL tools must support different levels of abstraction. Working at code level or working at design level may be two possible levels of abstraction.

Tools have different stakeholders such as end users and programmers. The number of possible stakeholders for any tool may be large. In our study, we have experimented with tools to analyze their support for six stakeholders: Customer’s IT organizations, End-users, Product line managers, Requirements engineers, Software architects, and Programmers.

A COMMON EXAMPLE

This section introduces the e-commerce system example used in the evaluation of tools. Catalogues are integral parts of any e-commerce systems. They are also common subjects for many stakeholders of e-commerce systems. For instance, buyers wanting to buy a product browse the catalogue to get the relevant product information and sellers wanting to sell the product put the relevant product information in the online catalogue. The focus of this study is to analyze the resources provided by XVCL and FMP to help sellers and buyers to mass customize the online catalogue component.

Figure 1 shows the variability model for the catalogue component. Product category, product information, and information language are external variation points and user interface is an internal variation point in the variability model. Book product category, basic information, English language, and HTML are examples of variants in the variability model. Purchaser can mass customize the catalogue by selecting variants available at the external variation points. For instance, the binding of a book product category, basic information, English language, and HTML variants from the respective variation points will derive one variant of the catalogue. Seller will be able to mass customize the catalogue in the same way. Certain constraints are also annotated in the variability model. The “requires_v-vp” constraint means that if the variant V is selected, the variation point VP must also be included. The “excludes_v_vp” constraint means that if the variant V is selected, the variation point VP cannot be selected.

EVALUATED TOOLS

This section presents the open-source based XVCL and FeaturePlugin to analyze their support for software product line engineering and, more specifically, for mass customization by the six stakeholder groups. The tools are evaluated based on the characteristics described in section “Evaluation framework for software product line tools.”

XVCL (XML-based Variant Configuration Language)

National University of Singapore developed XVCL as an open source project. XVCL is based on the frame technology developed by Netron, Inc (Bassett, 1997) that has been used in industry for a long time. XVCL technology provides a general-purpose mark-up language and the XVCL processor. XVCL helps design software in ways not possible with conventional techniques. The mark-up language helps represent programs in generic, reusable, and adaptable form. The XVCL processor works as an interpreter of the mark-up language, helping to automate the derivation of executable programs by processing the language components. XVCL can be used to configure variants for artifacts such as the software architecture, test cases, and program codes (Jarzabek and Pettersson, 2006; Jarzabek, 2007b).
Software product line engineering depends on domain specific reuse. Reusable, generic, and adaptable domain artifacts in a product line form the product line architecture. XVCL works with the principle that a meta-level program representation helps in generic design more than the conventional software components (Jarzabek, 2007b). The conventional techniques are used for the initial design and XVCL is then applied to build generic, adaptable meta-program components. XVCL can help manage variability in any domain artifacts that can be represented as text (Zhang and Jarzabek, 2003).

In XVCL, generic and adaptable meta-components are called x-frames. Each x-frame consists of textual content and XVCL commands. The content can be written in any programming language. XVCL commands are used to mark prospective variation points in the content. The XVCL <adapt> command is used to make all x-frames connected. The <insert> command enables the modification of the adapted x-frames at designated <break> points in arbitrary ways. All x-frames together form an x-framework. When XVCL is used to support SPL, the x-framework represents the product line architecture. Each specification x-frame (SPC) describes requirements for specific variants. A product line member SPC is the root of an x-framework. The XVCL processor processes the x-framework by starting from the SPC x-frame (Zhang and Jarzabek, 2003).

Figure 2 illustrates the XVCL syntax for two x-frames that are parts of the platform for the e-commerce product line described in the “Common example” section. X-frames show how (1) XVCL commands help in making components more generalizable by using generic variable names instead of hard coded ones and (2) x-frames use the <adapt> command to connect with other frames.

In Table 2, the results of evaluating XVCL are presented. The CID denotes the characteristics described in Table 1. If the tool supports a particular characteristic, then ‘Y’ is written in the respective column, otherwise “N” is assigned. CIDs 3, 13, and
14 are exceptions. CID 3 can have values ‘C’ or ‘R’ depending on whether the tool supports, respectively, compile time or run time binding. CID 13 can have values ‘C’ or ‘D’ depending on whether the tool supports, respectively, code level or design level abstractions. Numeric digits precede the ‘C’ or ‘D’, denoting the supported number of levels of abstraction. CID 14 can have up to six values ranging from 1 to 6 depending on whether the tool supports, respectively, Customer’s IT organizations (1), End-users (2), Requirements engineers (3), Product line managers (4), Software architects (5), and/or Programmers (6).

<table>
<thead>
<tr>
<th>CID</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Answer</td>
<td>Y</td>
<td>Y</td>
<td>C</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>1C</td>
<td>1,3,6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2. XVCL Characteristics

**Summary**

XVCL provides direct support for product line derivation and a general and flexible means to develop reusable domain artifacts. Its feature set is continuously enhanced. Integration of XVCL with CASE tools may be beneficial for supporting software development activities. Programmers can work with XVCL at a higher level of abstraction than with traditional Object Oriented Languages such as Java. Yet, XVCL facilitates working only at the code level and thus supports one level of abstraction. Sellers and buyers could mass customize the online catalogue component but only if they understood how to specify the generic variables and adapt x-frames at the code level.

Figure 2. XVCL syntax for specifying generic variable names and adapting x-frames
**FeaturePlugin**

In software product line engineering, feature modeling is a common technique to model commonality and variability in different artifacts. FeaturePlugin is a modeling plug-in for Eclipse Integrated Development Environment. Since variability permeates through various artifacts such as requirements documents, design documents, program code, and test documents, it becomes useful to integrate feature modeling with the development environment (Antkiewicz and Czarnecki, 2004). FeaturePlugin supports cardinality based feature modeling, the specialization of feature diagrams, and configuration based on feature diagrams. Approaches exist to map the feature models to other models such as behavioral models and data specifications (Czarnecki and Antkiewicz, 2005; Czarnecki et al., 2005). Through model transformations, product derivation is also possible (Czarnecki and Eisenecker, 2000).

Feature is a system property that is relevant to some stakeholder. Features are also used to capture commonality and variability among the applications of a software product line. Features are organized using feature diagrams. Feature models consist of feature diagrams with some additional constraints such as stakeholder information and binding time. A feature model helps represent all possible variants by describing all possible product configurations. Configuration is the process of deriving a concrete configuration conforming to a feature diagram by selecting and cloning features, and specifying attribute values (Czarnecki et al., 2005). The resulting configuration can be used as an input to code generators.

FeaturePlugin implements cardinality-based feature modeling. Feature cardinality is “an interval denoting how often a feature with subfeatures can be cloned as a child of its parent when specifying a concrete system” (Czarnecki et al., 2005). The feature cardinality \([1..n]\) denotes that at least one and at most \(n\) clones must be present in a concrete product. A mandatory feature is denoted by the feature cardinality \([1..1]\), while an optional feature is denoted by the feature cardinality \([0..1]\). The group cardinality is an interval placed on a feature group that denotes how many features can be selected from the respective feature group. Group cardinalities are denoted using angle brackets; for example, a group cardinality of \(<1..n>\) indicates that at least one and at most \(n\) features can be selected from the feature group, and that \(k\) is the total number of grouped features in the feature group. Additional constraints may be used to specify additional relationships (Antkiewicz and Czarnecki, 2004). Featureplugin is supplied with a default meta-model, which defines the structure of the feature model, along with the feature types described in the parent section. It includes definitions for feature properties, which include a name, id and description for every feature. It also supports meta-model editing to allow users to define additional feature attributes (Lau, 2006).

The example in Figure 3 shows a feature model developed by using FeaturePlugin. The model describes feature diagrams for the e-commerce product line, showing feature groups and their respective configurations when a feature is selected from each group. The model also shows how a group cardinality or feature cardinality is expressed using FeaturePlugin.

In Table 3, the results of evaluating FeaturePlugin are presented.

| CID | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Answer | Y  | Y  | R  | Y  | Y  | Y  | N  | Y  | N  | N  | N  | N  | 1D | 1,3,4,5,6 |

**Table 3. FeaturePlugin Characteristics**

**Summary**

FeaturePlugin provides direct support for product line derivation and an expressive means to develop and configure domain artifacts. It supports a higher level of abstraction than XVCL because it works at the design level. But FeaturePlugin only supports feature models, thus providing the same level of abstraction for all the stakeholders. It can be integrated with Rational Software Modeler, a CASE tool helping to instantiate products from feature models (Czarnecki et al., 2002). Sellers and buyers can mass customize the online catalogue component simply by selecting relevant features.

**CONCLUSIONS AND FUTURE RESEARCH**

This paper provided an evaluation framework, including a set of evaluation characteristics, for evaluating the capabilities of methodologies and tools to enable mass customization by a variety of stakeholders, including end users. The set of characteristics can be used to evaluate any SPL tools. We have designed the framework to be lightweight, enabling the screening and ranking of tools without the need to extensively use all the tools during evaluation. However, in this research, we could not use the framework to score the two evaluated tools by just reading their product descriptions and feature lists. We needed to analyze and use the tools in great depth to uncover the most important criteria for evaluation and create the framework accordingly.
The paper also provided an overview of the XVCL and FeaturePlugin tools and deployed the evaluation framework to analyze the features provided by these tools to support mass customization. FeaturePlugin works by modularizing features and then by composing features with the help of cardinality-based constraints. XVCL supports generic development of program components at meta-level. Besides the framework, the main finding of this paper is that if the best features of these tools are combined into a unified tool set, it is likely that end users can be enabled to mass-customize applications. XVCL and FeaturePlugin envisage high software development productivity by means of reusability, support for abstraction, and configuration mechanisms. They have similarities and differences, which in combination can provide mutual benefits.

Future research is needed to refine and validate the framework by applying it to different types of SPL tools. We will include SPL tools that may not score well using the framework in future research to provide a better understanding of the limitations and the scope of the framework. We will also study to what extent the framework can be used to screen SPL tools before using them extensively, for example, in pilot projects.

In addition, we will study the interaction of XVCL and FeaturePlugin to develop a unified tool set with the help of the two tools. When combined, these tools may be able to support mass customization by end users. We believe the discussed tools have unique strengths and weaknesses. Model transformations from feature models to product instantiations utilize a template-based approach. XVCL makes it feasible to use frames as templates in any template-based model transformation due to XVCL frames’ generic nature. Lee and Kang (2006) present an approach to dynamically reconfigure products by using a feature-based platform. Dynamical reconfiguration of features makes it easier to cope with SPL evolution.
ACKNOWLEDGMENTS

We thank the anonymous reviewers for their excellent comments that helped us improve this paper substantially.

REFERENCES


