ArchFeature: Integrating Features into Product Line Architecture

Gharib Gharibi and Yongjie Zheng
Department of Computer Science and Electrical Engineering
University of Missouri–Kansas City, Kansas City, Missouri, USA
ggk89@mail.umkc.edu, yzheng@umkc.edu

ABSTRACT
Product line architecture (PLA) captures the architectural commonality and differences among products of a product family. Existing PLA modeling approaches are limited in the sense that (1) it is often difficult to relate a product line feature to its implementation in the PLA, and (2) variation points in the PLA have to be manually developed and maintained. In this paper, we present a novel PLA modeling approach and toolset called ArchFeature that addresses these two challenges. ArchFeature integrates feature specification and PLA in a single monolithic architecture model. It includes a graphical modeling environment that can (1) automatically capture, maintain, and visualize the feature-PLA relationship, (2) encapsulate variability modeling from the user, and (3) support automatic derivation of architecture instances from the PLA. We implemented and integrated ArchFeature in ArchStudio, an Eclipse-based architecture development platform. We assessed its usability and effectiveness in a case study by using it to develop a full-featured architecture model for an open-source software system, Apache Solr.

CCS Concepts
Software and its engineering → Software organization and properties → Software system structures → Software architecture.

Keywords
Architecture modeling, software product line, product line architecture.

1. INTRODUCTION
A software product line [4, 20] is a family of software products that share substantial commonality and are often developed by a single organization. Product line development advocates that the differences (i.e. variability [21] or anticipated changes [19]) among the products must be explicitly represented in the artifacts such as feature models [6, 15] and product line architecture (PLA) [4] that can be reused in the process of deriving product members. A feature model captures both common and variable features in the problem space. Each feature is an end-user visible characteristic that distinguishes products of a product family [6]. PLA captures product variability in the solution space. It is commonly characterized as a monolithic architecture consisting of core elements shared by all the products, and variation points at which the products differ [7, 8, 11, 18, 21]. Each variation point is accompanied by a guard condition defined over product line features, determining when the variation occurs. By making appropriate decisions to resolve the variation points, a single product architecture can be derived from the PLA.

A number of PLA modeling approaches exist [7, 11, 12, 14, 18], supporting different modeling languages, architectural aspects, and domain-specific constructs. However, two challenges remain in this area and both are related to variability contained in the PLA. First, there is a conceptual gap between product line features and PLA. A single product line feature is often implemented by multiple scattered variation points in the PLA. As a result, the entire PLA model has to be examined to identify the variation points related to a feature. This is primarily because features and PLA are usually defined in two separate models and are developed with different tools. Second, variation points and guard conditions in the PLA have to be manually developed and maintained. This can cause significant overhead in PLA development and can prevent the user from focusing on architecture-specific design. In particular, an architecture element may contain child elements related to different features. It is even tedious and error prone under these circumstances to manually maintain PLA’s variation points (e.g. editing the accompanied guard conditions).

In this paper, we present a PLA modeling approach equipped with a graphical environment called ArchFeature. A primary difference between ArchFeature and existing PLA modeling approaches and tools is that it defines features, PLA, and their relationships in a single model. This is enabled by extending an existing XML-based architecture description language (ADL), xADL [9] that is mostly used for modeling a single system’s architecture consisting of components and connections. We developed and integrated new language constructs to model features and their relationships to the PLA. In particular, ArchFeature provides tool support for development, evolution, and customization of PLAs. It includes a graphical modeling tool to support side-by-side development and evolution of product line features and PLA. The tool can automatically capture, maintain, and visualize the feature-PLA relationships. The involved variation points in the PLA are also automatically maintained during this process. ArchFeature reduces the PLA development cost, and potentially enables architecture-centric software development [26].
We implemented and integrated ArchFeature in ArchStudio [2], an existing Eclipse-based architecture development toolset. We evaluated its usability and effectiveness in a case study with an open-source software system, Apache Solr [1]. The Solr system has around 146K SLOC. It is currently used at Cerner Corporation [5], an information technology company providing health care solutions. The case study was conducted independently by two Cerner employees. They used ArchFeature and successfully developed a full-featured model for Solr.

The rest of the paper is organized as follows. Section 2 presents two motivating scenarios to illustrate the issues that ArchFeature addresses. Section 3 introduces details of ArchFeature, including the underlying modeling approach and tool support. Section 4 describes the implementation work. Section 5 presents objectives, methodology, and results of the case study with the Apache Solr software system. Section 6 discusses and compares the related work. Finally, Section 7 concludes the paper.

2. MOTIVATING SCENARIOS

We describe two scenarios to illustrate two existing challenges involved in developing and maintaining PLAs. This highlights the tasks that ArchFeature can support. Ann in the scenarios is a software architect working in a company that follows the software product line development process. Ann is responsible for designing and modeling PLAs of the company.

Scenario 1 – Maintaining an existing PLA: Ann is asked to remove a feature from a PLA model that she created two years ago for a popular product line application. She cannot recall all the places where the corresponding feature was implemented in the PLA. Fortunately, she still has the application’s feature model – another artifact that includes the traceability links (e.g. realizedBy) from a feature to its related elements in the PLA. However, Ann finds that there are many inconsistencies between the feature model and the PLA model. For example, some PLA elements referred to in the feature model do not exist anymore. Moreover, Anna has to manually check if a PLA element is also related to other features to determine if it should be removed or updated with a new guard condition.

Scenario 2 – Developing a new PLA: Ann is working on a new PLA model. This time she decides to explicitly specify the guard condition for each variation point in the PLA model in terms of the related features. In that way, she thought she would be able to easily identify the feature-PLA relationships (e.g. using text-based searching). She soon realizes that some difficulties are involved. Not only is it time consuming to manually edit guard conditions, but also it is error prone. For example, she needs to ensure that the guard condition of a PLA component cover the guard conditions of all its child elements (e.g. interfaces), so that valid architecture instances can be derived. Manually enforcing such a restriction can be overwhelming for a complex software system.

3. ARCHFEATURE

ArchFeature is a pragmatic PLA modeling approach and toolset. Figure 1 shows an overview of ArchFeature and its underlying PLA model. The top half of the figure shows the ArchFeature interface, where a monolithic PLA and features are developed side-by-side in the same environment. The bottom half of the figure captures part of the PLA model, which includes specifications of one feature and a related PLA component. The ArchFeature toolset consists of a PLA modeling environment and an architecture derivation tool. Each is specifically introduced in the following subsections.

3.1 Underlying Modeling Approach

Extensible architecture modeling [9, 22] is a novel approach advocating that an ADL should be extensible to capture any principal design decisions (e.g. behaviors, structure, qualities) of a software system. It is different from traditional architecture modeling approaches that focus on the structure of a software system only. We exploit extensible architecture modeling in this project to address the PLA development issues described in Sections 1 and 2. We extended an existing ADL, xADL and developed new schemas for the definition of features and feature-PLA relationships. Specifically, we developed a new construct called feature. It defines a feature in terms of its type, related architecture elements, default value, display color, variants (if any), and other properties.

The code in of Figure 1 shows an example of the extended xADL specification that defines the FeatureC feature. The type element (Line 02) depicts the variability type of a feature. ArchFeature currently supports Optional, Alternative, and Optional-Alternative features. The archLinks element (Lines 06-08) includes links to the related PLA elements. Given that features and architecture elements are defined in the same xADL document, we use Xlink [10] to capture this information. The elements of defaultValue (Line 03) and displayColor (Line 05) contain the information about a feature’s default settings and the display color of its related PLA elements. They are used by the ArchFeature modeling tool to support architecture derivation and feature visualization respectively. The bindingTime element (Line 04) is included for future usage. The architecture in xADL is modeled as a configuration of components connected with each other via explicitly defined interfaces. The code in shows a definition example of a variation point related to FeatureC mentioned above. A variation point (Lines 02-11) is embedded into the specification of a component to represent an optional component. It includes a guard condition (Lines 03-10) defined as a Boolean expression containing an equality operator. The symbol element contains the name of the related product line feature. The value element indicates the value of the feature when the component should be included.

Overall, the extended xADL provides a modeling language that we can use to capture features, PLA, and their relationships in a single artifact. A primary benefit of this approach is that the user can focus on making application-specific architectural decisions, and will not be distracted by the tedious work such as editing the involved guard conditions. In addition, it also increases the quality of the developed PLA model in terms of the correctness of the defined variability information. In the following subsection, we discuss ArchFeature and its functions in more details.

3.2 Integrating Features in PLA Development

The ArchFeature modeling tool consists of two main user interface elements: a graphical editor and a feature list (see Figure 1). The graphical editor is used to visualize and edit PLAs. It supports operations such as creating components, managing connections, and adding interfaces. Architecture elements shown
in the figure are a set of components with optional ones represented by dashed lines. On the left of Figure 1 is a feature list containing features of a product line application. Each feature item corresponds to a feature definition in the ArchFeature model, and is preceded by a blue icon with an “F” letter on it, or “V” in case of variants. FeatureB in the figure is an alternative feature that includes two variants: FeatureB1 and FeatureB2. This integrated modeling environment can automatically capture, maintain, and visualize feature-PLA relationships as further explained below.

- **Developing features**: Feature development in ArchFeature is a simple and straightforward process. The user can right click the feature menu item and select one of the three feature types: optional, alternative and optional-alternative. In case of an alternative feature, the user can add new variants to the feature by right clicking the feature and selecting Add Variant. The user can set a variant as the default value of the corresponding feature by right clicking the variant and selecting Set As Default.

- **Automatic (i.e. implicit) creation of feature-PLA relationships**: ArchFeature supports automatic creation of feature-PLA relationships, including the links from a feature to its related PLA elements (e.g. Line 07 of FeatureB in Figure 1) and the variation points in all of the involved PLA elements (e.g. Lines 02-11 of FeatureB). The user can double click a feature in the feature list and enter the architecture editor to modify the PLA model. Any architecture changes made after this point will be automatically related to the selected feature.

- **Manual (i.e. explicit) creation of feature-PLA relationships**: Alternatively, the user of ArchFeature can explicitly create a relationship between a selected feature and an existing architecture element by right clicking the element and selecting Add to Current Feature. The underlying changes to the PLA model are automatically accomplished.

- **Visualizing feature-PLA relationships**: ArchFeature can automatically highlight the related PLA elements for a feature selected in the feature list. In Figure 1, the FeatureC feature is selected and the related architecture components are all highlighted in red. If the selected feature is an alternative feature with several variants, the elements related to all feature variants will be highlighted. The user can also change the feature-PLA relationship color by right clicking a feature from the feature list and selecting Change Color.

- **Removing feature-PLA relationships**: The user can easily remove an existing feature-PLA relationship by selecting the feature and then right clicking the involved architecture element and selecting Remove from Current Feature. The related links and variation points will be removed from the PLA model accordingly. Similarly, deleting a feature will automatically remove all of the related PLA elements based on the existing feature-PLA relationships.

Ann would be able to easily complete her tasks described in the first scenario in Section 2, if she created the PLA model using ArchFeature.
3.3 Managing Variation Points in the PLA

Another important function of ArchFeature is automating the process of managing (creating, updating, and removing) variation points and completely encapsulating it from the user. Specifically, a guard condition is automatically set in the background when a feature is selected in the feature list shown in Figure 1. A variation point with the preset guard condition will then be automatically created and embedded into the involved architecture elements. To support variation points related to multiple features, we use Boolean operators (e.g. OR) to connect the guard condition corresponding to each feature. For example, the user can relate a PLA element to two features consecutively. A primary benefit of automatically managing variation points in PLA modeling is that the user can focus on making application-specific architectural decisions. In particular, it increases the quality of the developed PLA model in terms of the validity of variability definition as further discussed in the following paragraph.

A main challenge involved in automatically managing variation points is handling the relationships between guard conditions of the related architecture elements. A PLA element may contain child elements corresponding to different features. In particular, these child elements cannot exist independently in a valid architecture model. Therefore, it is essential to ensure that the parent element in the PLA is always included if any of its child elements is included in the architecture of a single product. This requires that the guard condition of a variable element in the PLA model must cover all the guard conditions of its variable child elements. As described in the second scenario of Section 2, manually enforcing such a restriction can be expensive and error prone. An architecture element often contains multiple child elements and each child element can evolve in specific ways. ArchFeature integrates a special logic that can automatically enforce the rule. For example, when an interface gets related to a feature, ArchFeature will automatically add the corresponding guard condition to both the interface and the component that it belongs to (i.e. bottom-up). Similarly, when an existing core component gets related to a feature and becomes optional, the ArchFeature tool automatically extracts the guard conditions of its variable child elements and add them into the component’s guard condition using the OR operator (i.e. top-down). In this way, it ensures that an element that must not exist alone is always contained by its parent element in the derived architecture. This plays an important role in supporting architecture-centric product line development.

In addition, ArchFeature includes an architecture derivation tool called Selector. When a PLA is opened in the Selector tool, all the features defined in the PLA are automatically loaded and displayed to the user. The related information of each feature, such as feature type, current value, and description are also included. The user can easily change the value of each feature (i.e. select features) from a drop-down list. After that, a corresponding architecture instance can be automatically derived based on the selected features. Figure 2 shows a screenshot of the Selector tool. Note that the Selector tool currently does not support automatic enforcement of feature relationships. For example, selecting one feature automatically gets another related feature selected. The user has to manually check that a feature configuration fulfills the required feature relationships.

4. IMPLEMENTATION

We implemented and integrated ArchFeature in ArchStudio, an open-source Eclipse-based toolset used for developing architecture-based software systems. A video demo of the ArchFeature tool is available online [3]. ArchStudio includes a number of tools such as Archipelago, ArchEdit, and Archlight that can be used to model, visualize, analyze, and implement software architecture [2]. ArchFeature’s functions of PLA modeling and architecture derivation are based on Archipelago, an existing graphical architecture editor in ArchStudio and an architecture pruner included in ArchStudio. Our main implementation tasks included: (1) developing new xADL schemas for features specification, (2) modifying Archipelago for variability-related operations such as creating a variation point and highlighting feature-related PLA elements, (3) developing a new Eclipse View for feature-related operations, and (4) developing the Selector tool for architecture derivation.

The main functions of the ArchFeature modeling environment presented in Section 3 were developed and integrated in Archipelago. We reuse Archipelago’s existing support for the tasks such as file management, architecture visualization, and common operations of architecture modeling. A specific issue we addressed during this phase was the implementation of alternative features in PLA. Our original plan was to define an alternative variation point correspondingly and embed it in the involved PLA elements. For example, there could be an alternative architecture component including two <variant> sub-elements, which point to another two components respectively. It can be graphically represented as a component containing two inner smaller components as alternatives. A main problem of this solution was that it increases the complexity of variability modeling in PLA as different types of variation points usually have to be processed differently. Additional operations (e.g. addition of a component variant) may also need to be supported. As a result, we decided to use the expressive power of Boolean guard conditions included in the definition of a variation point to support alternative features. All variable architecture elements are marked as optional. An alternative feature is then implemented in PLA as a number of optional variation points. Each is governed by a guard condition that has the same guard symbol (i.e. feature) and different guard values that are mutually exclusive. In addition, we reused an existing architecture pruner of ArchStudio to implement the Selector tool. Based on the feature definition integrated in the PLA model, we made the entire architecture derivation process fully automated.
5. EVALUATION

A primary goal of our evaluation was to validate that the ArchFeature approach and toolset can be effectively used to develop a PLA model for a realistic software system. We have chosen to build the PLA model for Apache Solr 4.10.2 [1], a Java-based open-source enterprise search server. The Solr project has over fifty Java packages, more than a thousand classes, and approximately 146K SLOC. Solr has been through more than nine years of development. A number of features have been added to it while the system evolved over time, such as query result highlighting, spell checking, caching, NoSQL, and SolrCloud. This allows us to fully exercise ArchFeature’s functions described earlier in this paper (e.g. automatic variability modeling) and assess the effectiveness of ArchFeature in terms of helping the user understand the system and reducing the modeling workload.

The Solr architecture model developed in this case study is presented in the ArchFeature video demo [3].

The case study was independently conducted by a software architect and a software engineer from Cerner Cooptation [5]. Solr is currently used at Cerner, and this has launched a request for an explicit architecture model that can be used to describe the Solr system and associated features. Both participants were not involved in development of the ArchFeature approach and did not have the previous experience of using ArchStudio either. Their task in the case study was to use ArchFeature to develop a full-featured architecture model for Solr. Specifically, this includes: (1) analyzing Solr’s existing source code and documents to identify its main features, components, and their relationships; (2) developing an architecture model that captures the identified information using the ArchFeature modeling tool; (3) writing a report about the experience of using ArchFeature and the problems encountered. Development of the architecture model in the second step was the focus of the case study. The two participants created the core architecture structure first. After that, they added the features and related components to the architecture one by one. In particular, the relationships between features and architecture were created using both the explicit and the implicit modes introduced in Section 3.2. The architecture model of Solr developed in the case study included 183 components, 28 features, and 224 feature-architecture relationships in total. There were 27 core components representing the kernel functions of Solr such as evaluating queries, executing commands, and generating responses. Out of the 28 features, fourteen were optional features, eleven were alternative features, and three were optional-alternative features. The total number of feature variants contained in the alternative features was 143.

Several minor issues of ArchFeature were reported in the case study. One issue was that the existing feature types supported by ArchFeature are not sufficient for all the situations when modeling a real software system. Specifically, an OR feature is needed to allow more than one variants to be selected for a feature. This can be addressed by further extending xADL mentioned in Section 3.1 and defining the corresponding XML schema. It is one of our future work. A number of advantages of ArchFeature were also reported and validated. First, it is beneficial to integrate features in the development of PLA. It was easy to tell from the developed Solr model which portion of the system was relatively stable and which portion evolved frequently (e.g. involving a number of variants). The feature visualization function of the modeling environment made it straightforward to review the elements where a specific feature is implemented. Second, the modeling environment can automatically and correctly manage all the feature-architecture relationships. This was validated by utilizing the feature visualization function mentioned above and manually examining each feature. Third, the tool can automatically create, update, and remove variability definition for all the involved architecture elements despite the large size of Solr and the number of its features.

6. RELATED WORK

A number of different PLA modeling approaches and tools currently exist. In this section, we only discuss the approaches and tools that either focus on PLA modeling or explicitly address the variability mapping between software artifacts. Table 1 shows a comparison between ArchFeature and other existing approaches and tools along five criteria: Supported Features, Model, Feature-Model Mapping, Variability Definition in the Model, and Product Derivation. Most approaches in the table store features and the target model in two separate artifacts, and use traceability links to manage their relationships. Some (e.g. Feature Template, Dopler) save the trace information in one of the artifacts, and some (e.g. FeatureMapper, CIDE) save it in a third artifact. In general, they all face the challenge of automatically creating and maintaining traceability links between software artifacts that are specified in different languages and developed with different tools. EASEL, LISA, and ArchFeature are the three approaches integrating definition of features in the PLA model. A main difference is how variability is defined in their PLA models (i.e. change sets, the orthogonal model, and the monolithic model). It is important to highlight that ArchFeature is the only tool in the table supporting automatic creation and maintenance of variation points in the target model (e.g. PLA).

Among the tools listed in Table 1, FeatureMapper, LISA, and CIDE are most relevant to ArchFeature. FeatureMapper supports mapping features from a feature model to solution artifacts expressed in EMF/Ecore based languages (e.g. UML2). It is similar to ArchFeature in a number of aspects. Both offer two alternative ways (i.e. explicitly and implicitly) to relate a feature to elements in the solution space. Both support visualization of the elements in the solution space that are related to a feature. A main difference between them is that FeatureMapper stores the variability definition in a separate mapping file. In contrast, ArchFeature defines and embeds variation points in the PLA model. CIDE is a program development environment that can associate code fragments with one or more features and display them in different colors. This is similar to ArchFeature’s visualization technique. The difference is that CIDE focuses on program development and the programmer has to manually assign code to different features. LISA is one of the tools that support integrated modeling of features and architecture. It is more ambitious than ArchFeature in the sense that the architecture in LISA also integrates the information such as requirement decisions and code-level concepts (e.g. classes). In terms of variability modeling, LISA adopts the orthogonal variability modeling approach.
Table 1. Comparison of feature-model mapping approaches

<table>
<thead>
<tr>
<th>Approaches</th>
<th>Supported Features</th>
<th>Model</th>
<th>Feature-Model Mapping</th>
<th>Variability Definition in the Model</th>
<th>Product Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ménage [11]</td>
<td>Does not support feature development.</td>
<td>A monolithic PLA model with variation points included.</td>
<td>Shows optional elements in dashed lines.</td>
<td>The user manually creates and updates variation points in the PLA.</td>
<td>A prototype was built, requiring manual input of features.</td>
</tr>
<tr>
<td>Feature Template [7]</td>
<td>Feature model (i.e. features and feature relationships).</td>
<td>Model template with presence conditions and meta-expressions.</td>
<td>Defines variation points in terms of features. Does not support the mapping from feature to model.</td>
<td>The user manually creates and maintains presence conditions.</td>
<td>Fully automated with a prototype tool.</td>
</tr>
<tr>
<td>EASEL [14]</td>
<td>Integrates development of features and PLA; supports optional and alternative features, and feature dependencies and compatibilities.</td>
<td>Each change set corresponds to a feature.</td>
<td>A number of change sets, each containing a subset of architecture.</td>
<td>Composition of change sets of architecture.</td>
<td></td>
</tr>
<tr>
<td>FeatureIDE [23]</td>
<td>Feature model.</td>
<td>Feature module (i.e. a piece of source code.)</td>
<td>Supports mapping features and feature evolution to feature modules.</td>
<td>Variability implementation techniques, such as POP, AOP, and preprocessor.</td>
<td>Fully automated with tool support.</td>
</tr>
<tr>
<td>Gears [17]</td>
<td>Feature profiles.</td>
<td>A set of artifacts (e.g. requirements, design, and source code).</td>
<td>Variation points are defined in terms of features.</td>
<td>The user manually creates and maintains variation points based on Gears APIs.</td>
<td>Fully automated by the Gears product configurator.</td>
</tr>
<tr>
<td>Dopler [24]</td>
<td>Decision models that includes decision dependencies.</td>
<td>Asset models (e.g. PLA).</td>
<td>Maps the asset model to the decision model only.</td>
<td>Uses “inclusion conditions” to represent variation points.</td>
<td>Automatic generation of product configurations.</td>
</tr>
<tr>
<td>LISA [12]</td>
<td>Integrates modeling of features, requirements decisions, architecture, and code-level elements.</td>
<td>Supports mapping a variation point to all the related elements.</td>
<td>Defined in an orthogonal variability model.</td>
<td>Not supported.</td>
<td></td>
</tr>
<tr>
<td>CIDE [16]</td>
<td>Feature model.</td>
<td>Source code.</td>
<td>Uses a color mechanism to represent the feature-code fragment relationship.</td>
<td>The feature-code fragment relationship is defined in a separate file.</td>
<td>Fully automated with tool support.</td>
</tr>
<tr>
<td>ArchFeature [3]</td>
<td>Integrates modeling and development of features and PLA; supports optional, alternative, optional-alternative features.</td>
<td>Automatic/manual creation of feature-PLA mapping; visualization support.</td>
<td>Variation points in the PLA are automatically created and maintained.</td>
<td>Fully automated with tool support.</td>
<td></td>
</tr>
</tbody>
</table>

7. CONCLUSION
This paper presents a novel approach and toolset to improve and facilitate the process of developing PLAs. The scientific contribution of the approach lies in the integration of product line features and PLA in a single model, and their development in the same modeling environment. We developed an Eclipse-based PLA modeling environment that supports co-evolution of features and PLA elements. Our approach and toolset were validated in a case study and the results reveal that the approach can be effectively applied in real software development to bridge the gap between product line features and PLA, and to automate the definition of product line variability in the PLA development.

The future work of this study includes support for additional feature types and modeling of the relationships (e.g. mutual dependency) between product line features. Moreover, we are planning to extend ArchFeature to support the activities such as product line implementation in architecture-centric product line development and evolution.

8. ACKNOWLEDGMENT
This work was partially supported by the University of Missouri Research Board. We would like to thank Varun Narisetty for his help in implementing ArchFeature. We also thank Adam Carter and Jeffrey Lanning of Cerner Corporation for their help with the case study and their feedback.
9. REFERENCES


