Abstract—Product line architecture (PLA) is an important application of software architecture in software product line engineering. This paper presents an approach that addresses two existing issues in PLA development: 1) the difficulty of relating a product line feature to PLA elements, and 2) the overhead of manually developing variation points included in a PLA. The approach integrates features specification into the PLA model, and includes a PLA modeling tool called ArchFeature that supports side-by-side development of features, PLA, and their relationships. ArchFeature can automatically establish the feature-PLA relationship, create and maintain variation points in PLA, and highlight the PLA elements related to a feature. It was used in a case study to develop a full-featured architecture model for the Apache Solr software system.

Keywords—software architecture; software product line; architecture modeling

I. INTRODUCTION

Software product line engineering (SPLE) is focused on development and evolution of a family of related products that have substantial commonality - a software product line [6, 26]. It addresses the problems (e.g. architecture mismatch and maintenance of redundant code) of traditional software reuse by promoting planned reuse. Specifically, SPL is required that the differences (i.e. variability [29], anticipated changes [25]) among the products of a product line must be explicitly represented in the artifacts such as feature models [8, 21] and product line architecture (PLA) [6] that can be customized and reused in development of single products.

A feature model captures variability in the problem space and identifies the product line scope. It includes a collection of product line features and their relationships (e.g. mutual dependency). Each feature is an end-user visible characteristic used to capture commonalities or discriminate among systems in a product family [8]. PLA is the first artifact that places variability into the solution space. Software architecture is a set of principal design decisions of a software system [32]. It is commonly characterized as a configuration of components connected via explicitly defined interfaces. A PLA captures simultaneously the principal design decisions of many related products. Some of these design decisions are common among all the products, some are common among a subset of the products, and some are unique to individual products. As PLA plays an increasingly important role in SPL [31, 36], it is essential to make the PLA model relatively easy to develop, maintain, and understand.

A common approach to modeling a PLA is using a single monolithic architecture including both core elements and variation points [9, 10, 15, 24]. Variation points in the PLA identify the places (i.e. architecture elements) where the products differ [29]. Each variation point is accompanied by a guard condition determining when the variation occurs. The guard condition is usually defined as a Boolean formula over product line features. By making appropriate decisions to resolve the variation points, a single architecture describing a single product can be derived from the PLA. Compared with other PLA modeling approaches such as change sets [19] and the orthogonal model [26], the monolithic approach maintains integrity of the architecture model so that the PLA can be used to facilitate system comprehension and communication like a regular software architecture. Boolean operators (e.g. AND, OR) also enable representation of the abstract existence logic, such as one variation point related to multiple features. The monolithic modeling approach currently faces two challenges. Both are related to variability defined in the PLA and are further discussed below.

- There is a sizable mismatch or a conceptual gap between product line features and PLA. A single product line feature may translate to multiple scattered variation points in the PLA. As a result, the entire PLA often has to be examined to identify the variation points that are related to a feature. This is primarily because features and PLA usually are specified in separate models and are developed with different tools. One can evolve independently of the other. Many existing approaches [9, 13, 18, 29] address the issue by defining traceability links (e.g. realizedBy) from features to the related PLA elements at the metamodel level. However, automatically establishing and maintaining traceability links across different models and development environments is still a research challenge [1, 4].

- The variation points and guard conditions in PLA have to be manually developed. This can cause significant overhead in PLA development and prevent the user from focusing on application-specific design. In particular, an architecture element (e.g. component) may contain child elements (e.g. interfaces) related to different features. A
single element may also be related to multiple 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 pragmatic approach to developing PLAs. We extended an existing XML-based architecture description language (ADL), xADL [11], and integrated definition of product line features into the language. The extended xADL captures features and their relationships to PLA elements in a single PLA model. Based on it, we built a PLA modeling environment called ArchFeature. It supports side-by-side development of features and PLA. The user can add/remove/edit a product line feature, and modify the PLA model for the corresponding feature in ArchFeature. The user can also explicitly assign an existing PLA element to a selected feature. In both cases, the links mapping a feature to its related PLA elements are automatically created and saved in the model, so are the variation points in all the involved PLA elements. If a feature is selected, the related PLA elements are highlighted in a user-specified color. Similarly, removing a feature automatically removes the corresponding variation points from the PLA model. In addition, ArchFeature includes a Selector tool that can be used to derive an architecture instance from the PLA model. The Selector tool automatically loads the feature information (e.g. name, default value) defined in the PLA model. The user can easily select features to be included in an architecture instance.

The approach was implemented in ArchStudio [3], an Eclipse-based toolset for developing software architectures. It was evaluated in a case study where the ArchFeature tool was used to develop a full-featured architecture model of the Apache Solr software system [2]. Solr is an open-source enterprise server that has approximately 146K SLOC. It is currently used in Cerner Corporation [7], an information technology company providing health care solutions and services. The case study was independently conducted by two employees of Cerner. They were not involved in development of the approach presented in this paper, and did not have the experience of using ArchStudio either.

The rest of the paper is organized as follows. Section II describes the xADL modeling language and the extension that we made to it. Section III specifically introduces the functions of the ArchFeature tool, and its working mechanisms. Section IV is devoted to the implementation work done in ArchStudio. Section V presents the case study with the Solr system. Section VI reviews the related tools and methods and compares them with the approach presented in this paper. Section VII concludes the paper.

II. EXTENSION OF xADL

We use xADL, an existing XML-based ADL to model PLA in this study. xADL is a modular language that is highly extensible. It provides tool support for extension of the language. The current definition of xADL includes a set of XML schemas providing constructs that can be used to model both a single system’s architecture and PLA. The architecture in xADL is modeled as a configuration of components. A component is a locus of computation and state in a system. It is connected with other components via explicitly defined interfaces. xADL uses the monolithic approach described in Section I for PLA modeling. The code in Listing 1 shows an example definition of a variation point in xADL. A variation point or the <optional> element (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. It could be true, false, or the name of a variant, depending on the type of the feature as discussed later in this section.

Listing 1. Example of a variation point definition in xADL.

01: <component id="MultimediaLog">
02:   <optional>
03:     <guard>
04:       <BooleanExp>
05:         <equals>
06:           <symbol>ChatLog</symbol>
07:           <value> Multimedia </value>
08:         </equals>
09:       </guard>
10:   </optional>
11: </component>

Listing 2. Example of a feature definition in the extended xADL.
The type element (Line 02) in Listing 2 depicts the way a feature varies. We currently support three types of product line features: Optional, Alternative, and Optional-Alternative. An optional feature only exists in some products of the product line. An alternative feature exists in all the products of the product line, and each product may contain different variants (Line 09-13) of the feature. An optional-alternative feature is same as an alternative feature except that some products do not have it. In the rest of this paper, we refer to both as alternative features unless explicitly distinguished. Another important element in the feature definition is the archElements element (Lines 06-08). It includes links to the related PLA elements. Given that features and architecture elements are defined in the same xADL document, we use XLink [12] to capture this information. For example, Listing 2 contains a link (Line 07) referring to the MultimediaLog component defined in Listing 1.

This is an important extension of xADL and it serves as a bridge between product line features and their implementation in the architecture. The elements of defaultValue (Line 03) and displayColor (Line 05) contain the information about a feature’s default decision and the display color of its related PLA elements respectively. They are needed by the PLA modeling tool that we built to support feature visualization and architecture derivation introduced in Section III. Finally, bindingTime (Line 04) represents the time when a decision will be made on a feature. It is included for future usage. This study is only focused on the features resolved at development time (i.e. product line development).

Overall, the extension of xADL provides a modeling language that can be used to capture product line features, architecture elements, and their relationships in a PLA model. It serves as the basis of this research study. To fully integrate features in the PLA, the next step is building tools to support their development and evolution.

III. TOOL SUPPORT

This section presents a tool called ArchFeature that we built to support integrated development of features and PLA. It includes a PLA modeling environment and a Selector tool. The modeling environment can automatically capture, maintain, and visualize the feature-PLA relationship. It completely encapsulates variability modeling from the user.

A. Integrated Development of Features and PLA

Fig. 1 is a screenshot of ArchFeature’s PLA modeling environment. It consists of two main user interface elements: a feature tree and an architecture editor. The feature tree contains a list of optional/alternative features of a product line application under development. Each feature item corresponds to a feature definition shown in Listing 2, and is preceded by a blue icon (i.e. “F”) in the figure. The user can add a new feature, remove an existing feature, or edit the information (e.g. default value, binding time) of a feature. The user can also right click an alternative feature and add a new variant. The variant is then listed as its child element, and is preceded by a yellow icon (i.e. “V”) in the feature tree. The user can choose a specific variant as the default value of the corresponding feature by right clicking the variant and selecting Mark as Default. On the left of Fig. 1 is a graphical editor to visualize and edit an architecture model. It supports operations such as creating an architecture component, removing a connection, and adding a new interface to a component. All the modifications made to the architecture are automatically saved in the underlying xADL specification. Architecture elements shown in the figure are a set of components, with optional ones represented using dashed boxes. Each component is labelled with its name, and is connected with other components via interfaces that are represented by boxed arrows.

![Fig. 1. PLA modeling environment of the ArchFeature tool.](image)
An important function of ArchFeature is automatic creation and maintenance of the feature-PLA relationship, including 1) the links from a feature to the related PLA elements and 2) the variation points in all the involved PLA elements. An architecture element can be related to multiple features at the same time. This is enabled by the Boolean expression introduced in Section II. Specifically, the tool supports two complementary ways to relate a feature to PLA elements.

- The user can double click a feature in the feature tree shown in Fig. 1, and enter the architecture editor to modify the PLA model. All the new architecture elements that the user created, such as interfaces and connections will be automatically and implicitly related to the selected feature and marked as optional. The user can also choose to create or edit a common architecture element by not selecting any feature in the feature tree. In this case, the feature-PLA relationship will not be created.

- The user can explicitly establish the relationship between a feature and an existing architecture element. The user can select a feature in the feature tree, right click an architecture element, and select Add to Current Feature in the pop-up menu. To remove an established relationship, the user can right click the involved architecture element and select Remove from Current Feature instead. This function is particularly useful if the user wants to create features and relate them to an existing architecture model.

Additionally, ArchFeature can automatically highlight the architecture elements related to a feature that is selected in the feature tree. This is based on the links from features to PLA elements described above. It helps the user understand how a product line feature is implemented in the product line system. For example, the elements related to the BloomIndex feature variant are all shown in red in Fig. 1. The highlighting color can be changed by the user. If the selected feature is an alternative feature containing several variants, the elements related to all the feature variants will be simultaneously highlighted. Similarly, removal of a feature from the feature tree will automatically remove all the related variation points embedded in the PLA model. The involved architecture element will also be removed if it is optional and is only related to the removed feature. Otherwise ArchFeature just updates the guard conditions of the involved architecture elements.

B. Automatic Variability Modeling in PLA

A product line feature is usually implemented as a set of variation points (e.g. the \(<optional>\) element in Listing 1) in PLA [6]. Another important function of ArchFeature is that it automatically creates, updates, and removes variation points in PLA, and completely encapsulates management of variation points from the user. Specifically, a guard condition is automatically set in the background when a feature is selected in the feature tree in Fig. 1: \([Feature Name = true]\) for an optional feature and \([Feature Name = Feature Variant]\) for an alternative feature. A variation point with the preset guard condition will then be automatically created and embedded into involved architecture elements. Note that both optional and alternative features are implemented as optional variation points (i.e. the \(<optional>\) element) in the PLA model. The only difference is the format of the included guard condition mentioned above. It simplifies variability modeling in PLA, and is further discussed in Section IV. To support variation points related to multiple features, we use Boolean operators such as AND, OR to connect the guard condition corresponding to each feature. For example, the user can relate an architecture element to two different features consecutively. When this occurs, the tool simply combines their respective guard conditions with the OR Boolean operator.

A main challenge of automatically managing variation points in PLA is related to the hierarchical structure of PLA: an architecture element (e.g. a component) may contain child elements (e.g. interfaces) corresponding to different features. In particular, these child elements cannot exist independently in a valid architecture model. Therefore, it is important 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 of the product line. If the parent element is a core element, these child elements cannot exist independently in a valid architecture model. Therefore, it is important 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 of the product line. If the parent element is a core element that exists in the architectures of all the products of the product line, this is not a problem. It becomes tricky when the parent element itself is variable (e.g. optional) and contains its own guard condition. In that case, the guard condition of the parent element must be evaluated as True (i.e. to be included) if the guard condition of any of its child elements is evaluated as True in derivation of an architecture instance.

Rule of Thumb: The guard condition of a variable element in the PLA model must cover all the guard conditions of its variable child elements (i.e. \(G_{parent} = G_{child1} \lor G_{child2} \lor \cdots \lor G_{childn}\)) where \(G\) represent guard conditions, \(\lor\) is the OR operator.

Manually enforcing the rule above in PLA development can be expensive and error prone, as an architecture element often contains multiple child elements. Each may evolve (e.g. getting related to different features) in specific ways. The user can easily get overwhelmed by manually editing all the involved Boolean expressions. ArchFeature integrates a special logic that can automatically enforce the rule. Two example scenarios are given below to illustrate how this is achieved.

Scenario #1: An optional interface related to Feature A (i.e. Feature A is selected in the feature tree) is added to an existing optional component that is related to another feature, Feature B. As a result, the guard condition of the new (optional) interface is set to \([FeatureA=true]\). Meanwhile, the guard condition of the involved component is updated to \([FeatureB=true || FeatureA=true]\).

When an architecture element that cannot exist alone (e.g. the interface in the scenario above) gets related to a feature, ArchFeature will check its parent element (e.g. the component above) first. If the parent element is variable and has its own guard condition (e.g. \([FeatureB=true]\) above), the tool will append the child element’s new guard condition (e.g. \([FeatureA=true]\) above) to the parent element’s guard condition using the OR operator (e.g. \([FeatureB=true \lor FeatureA=true]\)). This process stops either when the parent element is a core element or when the parent element can exist independently in the architecture. This represents a bottom-up process of propagating guard conditions to maintain the rule of thumb described above.
Scenario #2: An existing core component containing an optional interface corresponding to Feature B gets related to Feature A, and becomes an optional component. As a result, the guard condition of the component is [FeatureA=true || FeatureB=true]. The interface’s guard condition remains same, which is [FeatureB=true].

When an existing core architecture element gets related to a feature and becomes variable (e.g. optional), the ArchFeature tool automatically extracts all the guard conditions (e.g. [FeatureB=true] above) of its direct child elements. It then appends the extracted guard conditions to the new guard condition (e.g. [FeatureA=true] above) using the OR operator (e.g. [FeatureA=true || FeatureB=true]). This represents a top-down process of maintaining the rule of thumb.

A primary benefit of automatically managing PLA’s variation points in ArchFeature is that the user can focus on making application-specific design decisions. In particular, it increases quality of the developed PLA model in terms of the validity of variability definition as discussed above. This plays an important role in the process of deriving architecture instances from the PLA model. It ensures that an element that must not exist alone is always contained by its parent element in the derived architecture.

C. Derivation of Architecture Instances from PLA

ArchFeature includes a Selector tool that can be used to generate an architecture instance from the PLA. Fig. 2 shows a screenshot of the tool. When a PLA is opened in Selector, all the features defined in the PLA are automatically loaded and listed to the user in a panel. The related information, such as feature type, default value, and description are also loaded and displayed. This is enabled by the guards modeled in product line features and PLA introduced in Section II. In contrast, many existing architecture derivation tools often require the feature information either be manually entered [15], or be loaded from another external model (e.g. a feature model) [9, 22, 23]. The user can customize a PLA in the Selector tool by simply changing the value of each feature from a drop-down list. The drop-down list includes pre-defined values of each feature that the user can choose from. Only true and false are shown for an optional feature. Feature variants are shown for alternative features. False and feature variants are shown for optional alternative features (e.g. Feature Query – Spell Checking selected in the figure).

In addition, the Selector tool includes an export/import function that allows the user to save/load feature selections. By this means, the user can reuse a previous feature configuration, and does not need to re-select features every time to generate an architecture instance. The user can also pre-define some recommended system settings for an application, such as basic version, professional version, and advanced version. Each version is defined by its own feature configuration file. 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.

IV. IMPLEMENTATION

We implemented and integrated the ArchFeature tool presented in Section III in ArchStudio, an Eclipse-based architecture development toolset. We developed functions such as integrated development of features and PLA, and automatic modeling of variation points in PLA. During this process, we were able to reuse the code and several existing tools provided by ArchStudio.

A. Implementation Environment: ArchStudio

ArchStudio includes a number of ancillary tools, such as Archipelago, ArchEdit, and TypeWrangler [3]. They support developing, visualizing, and analyzing architecture models using the xADL language introduced in Section II. Similar to xADL, ArchStudio can be extended with new tools to address additional architectural concerns, such as security [27] and traceability [4]. In a prior project [35], we successfully developed and integrated an architecture-implementation mapping tool in ArchStudio. Underlying ArchStudio is a code library that is shared by all the integrated tools. The library encapsulates the logic of accessing (e.g. reading/writing) the xADL specification, and provides high-level APIs such as addComponent and removeInterface for tool developers to use. In particular, new APIs (e.g. addFeature) can be automatically generated by a tool called Apigen from newly-defined xADL schemas. This reduces the effort of tool development in ArchStudio, and makes it a highly extensible platform for the architecture-related research.

B. Integration with ArchStudio

The main functions of the ArchFeature tool were developed based on two existing tools of ArchStudio, Archipelago and a selector prototype. They provide some basic functions such as visualization of architecture models, modification operations, and file management. Our implementation work was mainly focused on developing variability-related functions described in Section III, including 1) development and evolution of product line features and their relationships to the architecture; 2) automatic maintenance of variation points; 3) visualization of the PLA elements related to a selected feature; 4) automatic derivation of an architecture instance. The first three functions were implemented in Archipelago, and the last function was done in the selector tool.

Fig. 2. Screenshot of the Selector tool.
Archipelago is an existing graphical architecture editor that provides a symbolic boxes-and-arrows editing interface. The user can add/update/remove architecture components and links in Archipelago. The current version of Archipelago is focused on architecture modeling for single system development. It provides limited support for PLA modeling: the user can define an optional component and edit its guard condition. However, Archipelago does not support modeling of features or feature-PLA mapping. All the variability-related operations (e.g. creation of a variation point) in PLA have to be manually done. We made significant changes to the code of Archipelago in this regard. A specific issue we addressed was implementation of alternative features in PLA. Our original plan was to define alternative variation points (e.g. \(<\text{variant}\>) correspondingly and embed them in the involved architecture elements. For example, there could be an alternative architecture component that includes two \(<\text{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. The problem is that this would require operations such as adding/removing a variant be included in Archipelago to modify architecture elements. More importantly, it introduces variability into the operations of Archipelago. This conflicts with one of our goals in this project — automating variability definition so that the user can focus on application-specific design. As a result, we decided to exploit the expressive power of Boolean guards used in the definition of a variation point to resolve this problem. 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 (e.g. feature name) and different guard values that are mutually exclusive.

A product line selector prototype was built in a prior research project [15], where an exploratory study of PLA was done. It included the functions such as evaluating guard conditions and pruning un-selected architecture elements in a PLA. A main problem of the tool was that the user had to manually prepare the configuration information in the format of symbol-value pairs, based on which a corresponding architecture instance can be generated. This was primarily because there was no feature information explicitly defined in the PLA model. The user had to manually identify the symbols and allowed values from the guard conditions included in the PLA. In our implementation, we made the entire process of architecture derivation fully automated and feature-oriented. Features and related information are automatically loaded into the selector tool. The user can easily customize a PLA based on the provided feature information.

V. CASE STUDY

Case study in software engineering is an empirical research method emphasizing study of an object in its natural context [28]. It is used to evaluate the approach presented in this paper. A software architect and a software engineer from Cerner Corporation used the ArchFeature tool and developed a full-featured architecture model for an open-source software system that they use in Cerner.

A. Objectives

There are two primary objectives of the case study. First, we want to assess how the presented approach performs with a real software system. Scalability will be our main concern at this point. We will consider the approach to be successful if it works well with a system that has considerable size and a significant number of features. For example, the functions of implicitly and explicitly relating features to PLA should work as described in Section III. This can be validated by using the feature visualization function included in the tool. The other objective of the case study is to explore how the integrated development of features and PLA presented in this paper can help the user manage and understand the system.

B. Apache Solr

We have chosen to build the PLA for Apache Solr 4.0, a Java-based open-source standalone enterprise search server with a REST-like API. Solr is currently used in Cerner Corporation to support a variety of solutions. The Solr project has over fifty Java packages, more than a thousand classes, and approximately 146K SLOC. In addition, Solr has been through more than eight 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, and caching. In particular, an explicit architecture model that distinguishes elements related to different features does not exist yet. As Solr is increasingly popular, many companies began to experiment with extending the capabilities of Solr. This has launched a request for a public architecture model that can be used to describe the system and associated features.

The two participants of the case study from Cerner were not involved in development of the approach presented in this paper. They did not have the previous experience of using ArchStudio either. Their tasks in the case study were 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 in the ArchFeature modeling environment, and 3) writing a report about the experience using ArchFeature and the problems encountered. The first step above is essentially a process of architecture recovery [14]. The features were mainly identified from examining Apache Solr Reference Guide [30]. The components and the feature-component relationships were identified based on analysis of the UML diagrams (e.g. class diagrams, sequence diagram) automatically generated from Solr’s source code with an existing UML tool. Development of the architecture model in the second step was the focus of the case study, and a tutorial was provided by the first author at the beginning of the case study. Specifically, the core architecture structure was created first in the environment. The features and the related components were added into the architecture one by one after that. In particular, the relationships between features and architecture were created using both the explicit mode and the implicit mode introduced in Section III. The purpose was to fully exercise all the supported operations.

Table I shows the list of the features that were recovered and modeled for Solr in the case study. Each feature in the table is depicted by feature type, number of feature variants...
Not applicable for optional features), and number of involved architecture elements, 28 features have been captured in total. There are 14 optional features, 11 alternative features, and 3 optional-alternative features. The total number of feature variants that the alternative features contain is 143. For example, Feature Query Processors in Table 1 has 16 variants, each responsible for processing a specific type of query. The system configuration file specifies which of them should be included and instantiated in a Solr instance. The developed architecture model of Solr has 183 components. There are only 27 common components, representing the kernel functions of Solr, such as evaluating queries, executing commands, and generating response. These functions are not included in Table I as they exist in every Solr instance.

### TABLE I. SOLR’S FEATURES AND ARCHITECTURE MAPPING

<table>
<thead>
<tr>
<th>Feature</th>
<th>Type</th>
<th>Number of Variants</th>
<th>Number of Related PLA Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query Caching</td>
<td>Optional</td>
<td>n/a</td>
<td>3</td>
</tr>
<tr>
<td>Index Analysis</td>
<td>Optional</td>
<td>n/a</td>
<td>3</td>
</tr>
<tr>
<td>Result Clustering</td>
<td>Optional</td>
<td>n/a</td>
<td>5</td>
</tr>
<tr>
<td>Faceting</td>
<td>Optional-Alternative</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>Function Queries</td>
<td>Optional</td>
<td>n/a</td>
<td>2</td>
</tr>
<tr>
<td>Result Highlighting</td>
<td>Optional-Alternative</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>Recommendation</td>
<td>Optional</td>
<td>n/a</td>
<td>6</td>
</tr>
<tr>
<td>NoSQL Support</td>
<td>Optional</td>
<td>n/a</td>
<td>6</td>
</tr>
<tr>
<td>Query Boosting</td>
<td>Optional</td>
<td>n/a</td>
<td>7</td>
</tr>
<tr>
<td>Geospatial Search</td>
<td>Optional</td>
<td>n/a</td>
<td>4</td>
</tr>
<tr>
<td>Spell Checking</td>
<td>Optional-Alternative</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>Statistics Collection</td>
<td>Optional</td>
<td>n/a</td>
<td>6</td>
</tr>
<tr>
<td>Query Suggest</td>
<td>Optional</td>
<td>n/a</td>
<td>6</td>
</tr>
<tr>
<td>Term-based Query</td>
<td>Optional</td>
<td>n/a</td>
<td>7</td>
</tr>
<tr>
<td>SolrCloud</td>
<td>Optional</td>
<td>n/a</td>
<td>3</td>
</tr>
<tr>
<td>Extract PDF/Word</td>
<td>Optional</td>
<td>n/a</td>
<td>2</td>
</tr>
<tr>
<td>Posting Tool</td>
<td>Optional</td>
<td>n/a</td>
<td>1</td>
</tr>
<tr>
<td>Query Processors</td>
<td>Alternative</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Query Handler</td>
<td>Alternative</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Query Parsers</td>
<td>Alternative</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>Response Writers</td>
<td>Alternative</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Query Commands</td>
<td>Alternative</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Service Client</td>
<td>Alternative</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>Solr Parameters</td>
<td>Alternative</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Update Commands</td>
<td>Alternative</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Update Handlers</td>
<td>Alternative</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Update Loaders</td>
<td>Alternative</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Update Processors</td>
<td>Alternative</td>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>

Based on the result presented above and the report that our Cerner participants provided, we made the following conclusion about the modeling approach described in this paper. We believe the conclusion is generalizable to other complex software development given the setting of the case study (e.g. the size of Solr, the fact that the case study was conducted independently).

- It is beneficial to integrate product line features in the development of PLA. According to the case study participants, it is 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 also makes it straightforward to review the elements implementing a specific feature.

- ArchFeature can automatically create and maintain the feature-architecture relationship for a system of up to Solr’s size. This was validated by utilizing the feature visualization function mentioned above. The participants manually examined all the highlighted architecture elements of each feature, and compared them with the result of their analysis of Solr’s source code. There were several components related to multiple features in Solr’s architecture, ArchFeature successfully created and maintained all these relationships. In addition, ArchFeature can automatically create, update, and remove variability definition for all the involved architecture elements in PLA, despite the size of Solr and the number of features modeled in the case study. No human intervention was required on these operations in the case study.

- Several minor issues about the approach were also reported in the case study. One issue was that the existing feature types defined in the approach are not sufficient for all the situations in modeling of a software system. Specifically, an OR feature is needed to allow more than one variants to be selected from a feature. This can be addressed by modifying xADL’s schema discussed in Section II, and is made our future work.

### C. Threats to Validity

A primary concern that we had about the case study is that we essentially recovered the features and architecture of an existing software system. Ideally, the approach should be evaluated in development of the PLA model for an ongoing software project. This is a threat to validity of the case study. We think the case study results are valid because 1) converting the architecture of an existing system into a PLA is a typical PLA development approach [34], especially when the product line development did not start from the beginning; 2) all the operations that we exercised in the case study, such as creating a feature and relating features to architecture, are essential to product line development. In addition, we addressed the threat to validity by selecting the features that are relatively loosely coupled with the Solr system, as product line features typically are. For the features that are closely tied to the system and require significant changes to the system to be included or excluded, we made them core functions and did not model them in the case study.
VI. RELATED WORK

Many variability modeling approaches are around [10, 16, 21, 26, 29]. 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. A comparison between these approaches and the approach presented in this paper is made at the end. The purpose is to highlight advantages and limitations of the approach.

Ménage [15] is one of the early tools supporting PLA modeling. The variation point definition shown in Listing 1 is based on the design of Ménage, which defines PLA as a monolithic architecture model. A primary limitation of Ménage is that all the variation points and guard conditions in the PLA have to be manually created. Additionally, it offers limited support for relating features to the PLA. It shows optional elements in the PLA using dashed lines, and does not distinguish elements related to different features.

Feature template [9] advocates superimposition of all variants in a single model called model template that refers to features through annotations. Similar to the work presented in this paper, feature template defines Boolean formulas over the feature names from the feature model. Its template instantiation process is also similar to our derivation of architecture instances in the sense that evaluation of presence conditions (i.e., Boolean formulas) based on a specific feature configuration is involved. A main difference is that the work presented in this paper is mainly focused on PLA modeling. Some of its main functions, such as automatic creation of variation points and visualization of feature-PLA mapping are not supported by feature template.

EASEL [19] is another tool that supports PLA modeling. Similar to ArchFeature, EASEL is also based on an extension of xADL. The difference is that it separates variable architecture elements of a PLA into a number of change sets. Each change set only contains the architecture elements that implement a specific feature. In addition, EASEL explicitly manages the relationships between different change sets, such as structural dependencies and compatibilities. A main issue of EASEL is that it makes a PLA model less understandable as the definition of an architecture element is spread into multiple change sets. Derivation of an architecture instance in EASEL involves composition of change sets.

FeatureIDE [33] is an Eclipse-based framework supporting several product line implementation techniques, including feature-oriented programming (FOP) [5], aspect-oriented programming (AOP), and preprocessors. FeatureIDE integrates views and editors that can be reused among these techniques for domain analysis, implementation, and product derivation. Different from ArchFeature, FeatureIDE is mainly focused on mapping features defined in a feature model to the source code (i.e., feature module), instead of the PLA model.

FeatureMapper [18] is a tool that 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 can automatically relate a feature to elements in the solution space. Both support visualization of the elements in the solution space that are related to a specific feature, and both have the function of deriving a model instance based on a feature configuration. A main difference between them is how variability is defined in the target model. All the variability information (e.g. guard condition) is embedded in the involved architecture elements in ArchFeature, as we believe variability is an essential part of PLA. In contrast, FeatureMapper saves variability information and the feature-model relationships in a separate mapping file. In other words, the model (e.g. UML model) in FeatureMapper does not contain variability information.

Gears [23] is a product line framework emphasizing automatic derivation of product-specific artifacts, such as source code, requirements, and design. It includes a product configurator, a feature model, and a set of reusable artifacts containing defined variation points. The product configurator automatically customizes each reusable artifact based on a feature portfolio, and derives artifacts from each stage of the development lifecycle that belong to a product instance of the product line. Gears does not support integrated modeling of features and PLA as ArchFeature does.

Dopler [13] is a modeling language and toolset that supports mapping variability from the problem space to the solution space. It models the problem space using decision models and defines the solution space using asset models (e.g. PLA). Decisions and assets are linked with inclusion condition defined in the asset elements. The inclusion condition is similar to the guard condition used in ArchFeature. The difference is that Dopler does not support tracing from a decision to the related assets.

LISA [17] is a model and toolkit that also supports integrated modeling of product line 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 model [26]. Similar to ArchFeature, LISA can automatically capture and highlight the relationship between features and architecture elements. LISA currently does not support automatic derivation of architecture instances.

Other related tools or methods include Koala [24], XVCL processor [20], and CIDF [22]. Koala is one of the earliest ADLs supporting PLAs. It uses special language constructs (e.g. switch) to represent variability in the architecture. Koala does not explicitly support feature development and feature-PLA mapping. XVCL is another XML-based approach to capturing variability in product line development. It follows a composition with adaptation process in terms of product derivation. The XVCL processor can be used to automate the derivation process. Different from ArchFeature, variations in XVCL have to be manually defined. Finally, CIDF 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 the visualization technique presented in this paper. The difference is that CIDF is focused on program development and the programmer has to manually assign code to different features.
Table II compares the approaches described above with ArchFeature 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, CI DE) 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).

Compared with the existing approaches, ArchFeature is limited in terms of supported features. ArchFeature currently does not support feature relationships and special feature types such as OR mentioned in Section V. This is a limitation that we will address in future work.

### VII. Conclusion and Availability

This paper presents an approach integrating specifications of product line features and PLA in a single model, and their development in the same modeling environment. It includes an extended ADL, and an Eclipse-based PLA modeling tool that supports co-evolution of features and PLA elements. In particular, all the variation points in the PLA are automatically developed and maintained. The result of the case study shows that the approach can be effectively applied in real software development to bridge the gap between product line features and PLA, and to automate definition of product line variability in PLA development.


**ACKNOWLEDGMENT**

We would like to thank Adam Carters and Jeffrey Lanning of Cerner Corporation for their help with the case study. We would also like to thank the students of UMKC’s Fall 2014 graduate class in Software Architecture and Design for their helpful feedback of the ArchFeature tool.
REFERENCES


