INTEGRATING FEATURES IN THE DEVELOPMENT OF SOFTWARE PRODUCT LINE ARCHITECTURE

A THESIS IN

Computer Science

Presented to the Faculty of the University
of Missouri-Kansas City in partial fulfillment
of the requirements for the degree

MASTER OF SCIENCE

By

VARUN NARISETTY

B.TECH, JNTU Hyderabad, 2010

Kansas City, Missouri

2015

©2015

VARUN NARISETTY

ALL RIGHTS RESERVED

INTEGRATING FEATURES IN THE DEVELOPMENT OF SOFTWARE PRODUCT LINE ARCHITECTURE

Varun Narisetty, Candidate for the Master of Science Degree

University of Missouri-Kansas City, 2015

ABSTRACT

Software product line architecture (PLA) is one of the most promising applications of software architecture. This paper presents a pragmatic PLA development approach with tool support. It addresses two existing issues of PLA development, the difficulty of relating product line features to PLA, and the overhead of manually creating and maintaining variation points in PLA. The approach is implemented and integrated in ArchStudio, an Eclipse-based architecture development toolset. The developed tool supports (1) side-by-side integrated development of features, PLA, and their relationships, (2) automatic variability modeling in PLA, and (3) derivation of architecture instances from the PLA model. To evaluate the scalability and effectiveness of the approach, I have used the work done by Adam Carter and Jeffrey Lanning [30] as a case study using the developed tool to create a feature-integrated architecture for the Apache Solr software system - a Java-based enterprise search server used in the Cerner Corporation.

APPROVAL PAGE

The faculty listed below, appointed by the Dean of the School of Computing and Engineering, have examined a thesis titled “Integration Features in the Development of Software Product Line Architecture,” presented by Varun Narisetty, candidate for the Master of Science degree, and certify that in their opinion, it is worthy of acceptance.

Supervisory Committee

Yongjie Zheng, Ph.D., Committee Chair
School of Computing and Engineering

Praveen Rao, Ph.D. Committee Member
School of Computing and Engineering

YugYung Lee, Ph.D. Committee Member
School of Computing and Engineering

CONTENTS

ABSTRACT iii

LIST OF ILLUSTRATIONS vii

LIST OF TABLES viii

ACKNOWLEDGEMENTS ix

Chapter

1. INTRODUCTION 1

1.1 Motivation 1

1.2 Problem Statement 2

1.3 Thesis Outline 3

2. BACKGROUND AND RELATED WORK 5

2.1 Background 5

2.1.1 Extension of xADL 5

2.2 Related Work 8

2.2.1 Ménage 8

2.2.2 EASEL 9

2.2.3 Feature Maper 9

3. TOOLS DEVELOPED 14

3.1 Integrated Development of Features and PLA 14

3.2 Automatic Variability Modeling in PLA 17

3.3 Deviation of Architecture Instances from PLA 20

4. IMPLEMENTATION 23

4.1 Implementation Environment: ArchStudio 23

4.2 Integration with ArchStudio 23

5. RESULTS AND EVALUATION 26

5.1 Case Study 26

5.1.1 Objectives 26

5.1.2 Apache Solr 26

5.1.3 Threats to Validity 30

CONCLUSION, AVAILABILITY, AND FUTURE WORK 32

REFERENCES 33

VITA 37

ILLUSTRATIONS

Figure Page

1. PLA Modeling Environment 15

2. Screenshot of the Selector Tool 21

3. Example of a Derived Architecture Instance. 22

TABLES

Table Page

1. A Comparison of Feature-Model Mapping Tools 12

2. Solr's Features and Architecture Mapping 34

ACKNOWLEDGEMENTS

I take this opportunity to express my gratitude to Dr. Zheng for his continuous support throughout my thesis. I also want to thank all my friends and family for their encouragement. This work was in part supported by a grant from the University of Missouri Research Board.

CHAPTER 1

INTRODUCTION

1.1 Motivation

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 [5, 19]. It addresses the problems (e.g. architecture mismatch and maintenance of redundant code) of traditional software reuse [27] by promoting planned reuse. Specifically, SPLE requires that the differences (i.e. variability [21], anticipated changes [18]) among the products of a product line must be explicitly represented in the artifacts such as feature model [7] and product line architecture (PLA) [5] 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 [7]. 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 [24]. It is usually characterized as a configuration of components communicating with each other 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. With

PLA playing an increasingly important role in SPLE [23, 29], development of methods and tools to support PLA becomes necessary.

1.2 Problem Statement

A common approach to modeling PLA is to use a single monolithic architecture with variation points embedded in variable architecture elements [8, 9, 11, 17]. Variation points in PLA identify the places where the products differ [21]. Each variation point is accompanied by a guard condition that determines 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. The monolithic PLA modeling approach maintains the integrity of PLA so that it can be used for communication and system comprehension as regular software architecture [24]. Using Boolean expression in the definition of variation points also makes it relatively easy to represent the advanced existence logics, such as one variation point involved with multiple features, when Boolean operators (e.g. AND, OR) can be used. Meanwhile, the approach also faces some significant challenges. Two of them are listed below and are specifically addressed in this research study.

- 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 [13]. 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 variable architecture elements are usually developed and saved in two separate artifacts (i.e. feature models, PLA) at different abstraction levels. An existing solution to this problem is creating traceability links between the feature model and PLA

[1]. However, it is difficult to automatically update the links when both artifacts frequently evolve [29].

- Creation and maintenance of variation points and guard conditions in PLA have to be manually done. This can cause significant overhead in PLA development and prevent the user from focusing on architecture design. In particular, an architecture element (e.g. component) may contain child elements (e.g. interfaces) related to different features. A single architecture element also may be related to multiple features. It is even challenging under these circumstances to manually create and maintain variation points in PLA so that a valid PLA model can be used to derive architecture instances.

1.3 Thesis Outline

In this paper we present a pragmatic approach to developing PLAs. I extended an existing XML-based architecture description language (ADL), xADL [9], and integrated definition of product line features into the language. Based on it, I built a tool that includes 1) a PLA modeling environment where features and PLA can be developed side-by-side, and 2) a selector tool that can automatically generate an architecture instance based on a given feature configuration. The user can add/remove/edit a product line feature, and modify the PLA model for the corresponding feature in the modeling environment. The relationships between features and PLA elements can be created either automatically based on the changes that the user made to the PLA model for a specific feature, or manually with the user explicitly assigning a selected architecture element to a feature. In either way, creation and maintenance of the variation points (e.g. update of the associated guard condition) in the PLA are automatically done and completely encapsulated from the user. When a feature is selected, all the corresponding variation points in the PLA can be highlighted in a user-

specified color. Similarly, removing a feature will automatically remove all the related variation points from the PLA model. The included selector tool automatically loads the information (e.g. name, default value) of defined features from the PLA model and shows them to the user as a default feature configuration. This further automates the process of PLA instantiation.

The approach is implemented and integrated in ArchStudio [26], an Eclipse-based toolset for developing software architectures. As a case study, we used the developed tool to model the architecture of the Apache Solr software system [2] used in the Cerner Corporation [6] - an information technology company providing health care solutions and services. Solr is a Java-based open-source standalone enterprise server that has approximately 146K SLOC. We identified and developed a list of Solr's features, and successfully integrated them into the architecture model that we created for the Solr system. All the involved variation points were automatically created.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Background

2.1.1 Extension of xADL

I used xADL, an XML-based ADL to model PLAs in this project. The existing definition of xADL includes a set of XML schemas providing constructs to model both single system's architecture and PLA. The architecture in xADL is modeled as a configuration of components. A component is connected to other components via explicitly defined provided interfaces or required interfaces. A provided interface of a component contains the operations implemented inside the component. A required interface includes the operations that the component needs other components to implement. In terms of PLA modeling, xADL uses the monolithic architecture approach mentioned in Section I. The code in List.1 shows an example definition of a variation point in xADL. A variation point (i.e. <optional>, Lines 02-11) is embedded into the definition of a component to represent an optional component. It includes a guard condition (Lines 03-10) defined as a Boolean expression indicating when the component should be included. The symbol element (Line 06) is the name of the involved product line feature. The corresponding value element (Line 07) could be true, false, or a variant depending on the type of the feature as further discussed below.

```
01: <component id="…">
02:   <optional>
03:     <guard>
04:       <booleanExp>
```

```
05:       <equals>
06:         <symbol> … </symbol>
07:         <value> … </value>
08:       </equals>
09:     </booleanExp>
10:   </guard>
11:  </optional>
12:  … <!--The remainder of the component-->
13: </component>
```

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

In order to extend xADL to model product line features and the relationships between features and related variation points in PLA, I developed new xADL schemas and integrated them into the existing definition of xADL. Development of xADL schemas was relatively straightforward based on the XML knowledge that I have. Integrating them into the language and writing code to support new language elements (e.g. reading/writing) could be difficult. At this point, xADL has a tool called Apigen [9] that can parse a collection of xADL schemas and automatically generate a code library to access the language elements defined in all the included schemas. This significantly reduced our workload. The generated library provides high-level APIs such as addComponent and removeFeature, based on which I was able to build graphical tools to manipulate the xADL model. Specifically, the extended xADL introduced a new language element, feature, that is parallel to the component element shown in List.1. Each feature element has an identifier, and includes the following six child elements.

- Type: depicting the way a feature varies. My work currently support three types of product line features: optional feature, alternative feature, and optional-alternative feature. 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 instances (i.e. variants) 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, by default I refer to both as alternative features unless explicitly distinguished.

- Description: a user-readable text message describing the corresponding feature.

- BindingTime: the time when a decision will be made on the feature. Variations include development time, link time, initialization time, or runtime. In this project, I only focus on features to be resolved at development time. The decisions of some features may be delayed to runtime (e.g. dynamic adaptation).

- DefaultValue: the default decision on a feature. It could be true or false for an optional feature, and a specific variant for an alternative feature. This information is useful in derivation of architecture instances, so that the user does not need to make an explicit decision for every feature.

- Links: traceability links to the architecture elements (e.g. an optional component) that are related to the feature and should be included if the feature is selected in a product. As features and architecture elements are both defined in the same xADL document, I use Xlink to capture this information. This is an important extension of xADL as it essentially serves as a bridge between product line features and their implementation in the architecture.

- DisplayOptions.: the way (e.g. color) a feature and its associated architecture elements should be displayed. This information is not essential to the feature definition. It is included mainly to facilitate visualization of feature implementations in the architecture editor.

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 development of PLA, the next step is building tools to support activities such as creation of features and relating features to PLA elements. This is specifically discussed in the following section.

2.2 Related Work

2.2.1 Ménage

Ménage [11] is one of the early tools for PLA modeling. The idea of embedding variation points governed by guard conditions into architecture elements to represent variability in PLA was first used in Ménage. However, all the variation points and guard conditions have to be manually created in Ménage. In contrast, creation of variation points is fully automated with our tool as the user edits the architecture for a specific feature. Another important difference between Ménage and our tool is that Ménage offers little support in terms of relating features to their implementation in PLA. It only shows optional elements in PLA using dashed lines, and does not distinguish elements for different features in visualization.

2.2.2 EASEL

EASEL [13] is another tool that supports PLA modeling. Similar to the tool presented in this paper, 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 changesets. 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 problem of EASEL is that it makes a PLA model less understandable as the definition of an architecture element is spread into multiple changesets. Derivation of architecture instance in EASEL is essentially about composition of changesets, which can become difficult when the number of overlapping elements between change sets increases.

2.2.3 Feature Mapper

FeatureMapper [12] is a tool that supports mapping features from feature models to solution artifacts expressed in EMF/Ecore-based languages (e.g. UML2). It is similar to the tool presented in this paper in a number of aspects. Both can either manually or 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 that the tool I developed is specifically focused on the PLA model consisting of components and connections. All the variability information (e.g. guard condition) is embedded in the involved architecture elements as I believe variability is an essential part of PLA. In contrast, FeatureMapper is mostly used for UML models that are of a lower level of abstraction. Particularly, the variability information and the mappings of

features to solution artifacts are saved in a separate mapping model. This potentially increases the overhead of software maintenance as one more artifact is introduced in addition to feature model and UML model.

Gears [16] 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. Different from Gears, the approach presented in this paper is focused on the relationship between features and PLA as I believe architecture should play a central role in software development [29], including product line engineering.

Other related tools or methods also include Koala [17], XVCL processor [14], and CIDE [15]. Koala is one of the first representations of PLAs. It uses special language constructs (e.g. switch) to represent variability in the architecture. Koala does not support feature-oriented PLA modeling or visualization. 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 our tool, variations in XVCL have to be manually defined. Finally, 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 the visualization technique presented in this paper. The difference is that CIDE is focused on program development and cannot support automatic creation of variation points in the code.

Table II compares the tools described above and the tool presented in this paper along five criteria: supported features, target model, feature-model relationship, variability modeling, and product derivation. From the table, it can be seen that FeatureMapper and the tool I developed both offer comprehensive support in terms of relating features to the target model (e.g. automatic creation and visualization of the relationship). In particular, the tool I developed is the only one supporting automatic creation and maintenance of variation points in the target model (e.g. PLA). A main limitation of the tool is that it does not fully support feature relationships as many other tools do.

Feature template [8] advocates superimposition of all variants in a single model called model template that refers to features through annotations. At this point, it is similar to the work presented in this paper. In particular, feature template also suggests use of Boolean formulas over the set of feature names from the feature model. Its template instantiation process is also similar to our derivation of architecture instance 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.

FeatureIDE [25] is an Eclipse-based tool that mainly supports feature-oriented software development (FOSD) [4]. FOSD is similar to EASEL in the sense that they both separate software elements into fragments based on the feature they correspond to, and generate product instance by composing the fragments. The difference is that EASEL is focused on the architecture level, while FOSD is mainly at the code level. Therefore, they both face the

challenge of composition when the overlapping between separated fragments increases or the order of composition has to be considered [15].

Table 1: A Comparison of Feature-Model Mapping Tools

Tools	Supported Features	Target Model	Feature-Model Relationship	Variability Definition in the Model	Product Derivation
Ménage	Optional, alternative features only.	The monolithic PLA model including both common elements and variation points.	Optional elements are shown in dashed lines.	The user manually creates and maintains variation points.	A prototype tool was built, requiring the user's intervention.
Feature Template	Feature model (i.e. features and feature relationships).	Model template with presence conditions and meta-expressions.	Variation points are defined in terms of features.	The user manually creates and maintains presence conditions.	Fully automated with a prototype support.
EASEL	Feature dependencies and compatibilities.	A number of change sets, each containing a subset of architecture.	Each change set corresponds to a feature.	A change set.	Composition of change sets of architecture.
FeatureIDE	Feature model.	Feature module (i.e. a piece of source code.)	Each feature module corresponds to a feature.	A feature module.	Composition of code feature modules.
FeatureMapper	Feature model.	Models defined in Ecore-based languages (UML2).	Automatic/manual creation and removal of relationship; visualization of relationship.	Variability is saved in a separate mapping model.	Fully automated with tool supported.
Gears	Feature model.	A set of artifacts (e.g. requirements, design, and	Variation points are defined in terms of features.	The user manually creates and maintains	Fully automated by the product configurator.

		source code).		variation points based on Gears APIs.	
Koala	Features are not explicitly supported.	A PLA model defined in the Koala language.	Not addressed.	Variability is captured using special language constructs.	Supported by the Koala complier.
XVCL	Optional features.	A hierarchy of meta-components.	Not addressed.	The user manually edits the meta information.	Fully automated using XVCL processor.
CIDE	Feature model.	Source code.	Uses a color mechanism to represent the feature-code fragment relationship.	Highlighted code fragment.	Fully automated with tool support.
Tool presented in this paper	Optional, alternative, optional-alternative features.	The monolithic PLA model including both common elements and variation points.	Automatic/manual creation and removal of relationship; visualization support.	Variation points are automatically created and maintained in PLA.	Fully automated with tool support.

CHAPTER 3

TOOLS DEVELOPED

This section presents a tool that I built to support integrated development of features and PLA. It includes a PLA modeling environment and a selector tool. The modeling environment has two novel functions: integrated development of features and PLA, and automatic variability modeling. The selector tool can be used to automatically derive an architecture instance from the developed PLA model.

3.1 Integrated Development of Features and PLA

Figure 1 is a screenshot of the PLA modeling environment that I developed in ArchStudio, an existing Eclipse-based architecture development toolset that is specifically introduced in Section IV. The modeling environment consists of two primary user interface elements: a feature tree and an architecture editor. The example shown in the figure is the PLA model of a chatting application under development. The core elements of the model include two client components and one server component. The clients exchange messages via the server. In addition, there are also variable architecture elements (e.g. interfaces, components) created for a number of features, such as ChatLog, Game, and Send File listed in the feature tree. Different chatting application products may include different sets of features.

The feature tree in Figure 1 contains a list of features of the product line application. The user can add an optional feature or an alternative feature. For each feature, the user can further define its associated information such as binding time and default value discussed in Section II. The user can right click an alternative feature in the feature tree to add a variant to it. For example, the ChatLog feature in Figure 1 is an alternative feature that has three

variants: Multimedia, Plain Text, and Rich Text, representing three different formats of chatting history. The user can also choose a specific variant as the default value of the corresponding feature by right clicking the variant and selecting “Mark as Default”.

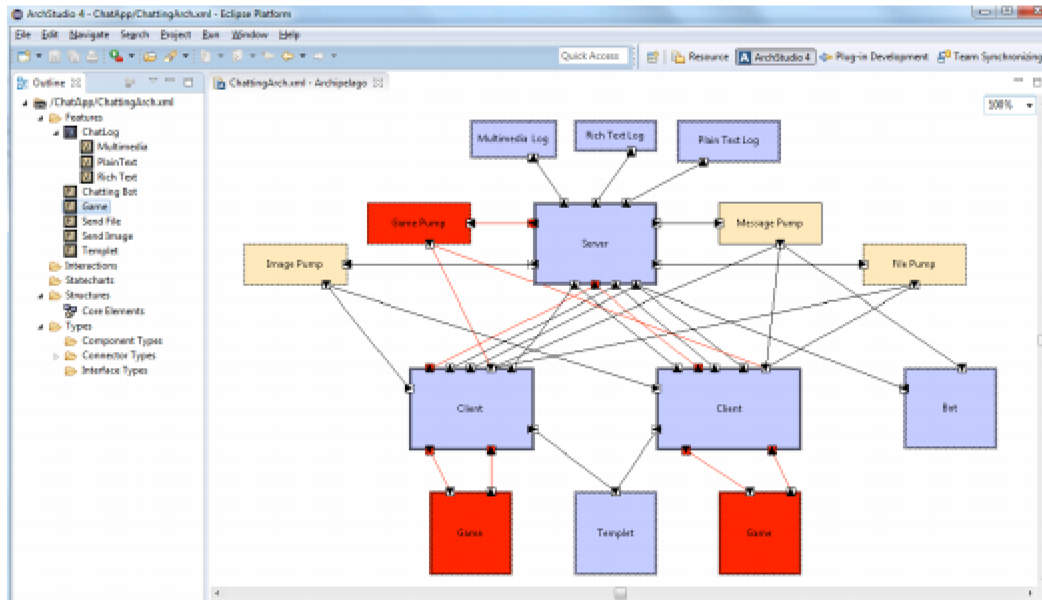

Figure 3.1 PLA Modeling Environment.

On the right of Figure 1 is a graphical editor to visualize and edit an architecture model specified in the xADL language. It is based on an existing architecture editor of ArchStudio that supports basic operations such as adding 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, so that the user does not need to directly access the xADL file. A specific change I made to the editor was adding the support for creation, removal, and visualization of the relationships between features in the feature tree and architecture elements in the editor. Relating a product line feature to a variable architecture element involves two specific operations: creating a traceability link from the feature to the architecture element, and updating the guard

condition of the variation point embedded in the architecture element to include the corresponding feature. Both operations are automatically done with the developed tool.

The developed PLA modeling environment supports two complementary ways to relate a feature to a PLA element. The user can double click a feature in the feature tree and enter the architecture editor to modify the PLA model to implement the feature. The relationships between the feature and all the new architecture elements that the user created will be implicitly and automatically established. By default, all the architecture elements that the user created will be automatically related to the selected feature and marked as optional. To create or edit a common architecture element, the user should double click "Core Elements" shown in Figure 1. In this case, the architecture elements that the user worked on would not be associated to any feature. In addition, the user can explicitly establish the relationship between a feature and an existing architecture element. The user can select the feature in the feature tree, right click an element in the architecture editor 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 wanted to create features and relate them to an existing architecture model.

The established feature-PLA relationship is used by the developed tool in two primary ways. First, the tool offers visualization support. All the variation points related to a feature will be highlighted in a user-specified color in the architecture editor when the feature is selected in the feature tree. For example, the architecture elements (e.g. components, interfaces, connections) related to the Game feature of the chatting application are shown in red in Figure 1. This helps the user understand and communicate with how a product line

feature is implemented in the product line system. The user can also right click a feature or a feature variant and change the display color. If the selected feature is an alternative feature that contains several variants, the variation points related to all the feature variants will be simultaneously highlighted. Second, removal of a feature from the feature tree will automatically remove all the related variation points embedded in the PLA model. The tool follows the traceability links of the feature to be removed and automatically updates the guard conditions of all the involved architecture elements.

3.2 Automatic Variability Modeling in PLA

The second main function of the developed PLA modeling environment is automatic creation and maintenance of variation points related to a feature. Specifically, the user can open a PLA model in the architecture editor by double clicking an optional feature or a variant of an alternative feature in the feature tree shown in Figure 1. After that, a guard condition will be automatically set in the background: Feature Name = true in case of an optional feature or Feature Name = Feature Variant in case of an alternative feature variant. A variation point with the present guard condition will be automatically created and inserted into the architecture elements that become related to the feature. Similarly, the guard condition can also be automatically removed when an architecture element is not related to the feature any more. A primary benefit of automatically creating/removing variation points is that the user can focus on editing architecture for application logics as the modeling environment shields variability modeling in PLA from the user.

As introduced in Section II, the guard condition of a variation point is defined by a Boolean expression in xADL. To support variation points that are related to multiple features, I use the Boolean operators such as AND, OR to connect the guard condition

corresponding to each feature. For example, the user may want to relate the same architecture element to two different features consecutively. When this occurs, the developed tool will automatically combine their respective preset guard conditions with an OR operator. The user can also have two features selected at the same time in the feature tree, representing the condition when the two features must be selected at the same time. In this case, the tool will automatically insert the AND operator into the guard conditions of the involved variation points. In the future, I plan to support more advanced operators such as NOT, NOR, as I believe they potentially address the relationships (e.g. mutual exclusion) between features.

A main challenge of automatically creating and maintaining variation points 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 instance. Therefore, it is important to ensure that the parent or containing element is always included if any of its child elements is included in a derived architecture instance given a feature configuration. If the parent element is a common 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 based on a feature configuration. In fact, I believe the following rule of thumb must be held in the developed PLA model.

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. Gparent = gchild1 || gchild2 || … || gchildn, G and g represent guard conditions, || is the OR operator)

Manually enforcing the above rule 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 the workload of editing the involved Boolean expressions. The PLA modeling environment I developed integrates some special logics that can automatically enforce the rule. Two example scenarios are given below to illustrate how this is achieved.

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

When an architecture element is associated with a feature (i.e. getting a new guard condition), the developed tool 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 an OR operator (e.g. FeatureB=true || FeatureA=true). This process stops either when the parent is a core element or when the parent element can exist independently in an architecture. It 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, FeatureB=true.

When an existing core architecture element becomes variable (e.g. optional) and gets a new guard condition, the developed tool will automatically extract all the guard conditions (e.g. FeatureB=true above) of its direct variable child elements and append them to its own guard condition (e.g. FeatureA=true above) using an OR operator (e.g. FeatureA=true || FeatureB=true). This represents a top-down process of maintaining the rule of thumb.

In addition to the two examples described above, the tool includes other logic to automatically manage the variation points included in a PLA model. For example, the tool will automatically remove an optional architecture element from the PLA model if its relationship with the only feature that it corresponds to is deleted. Otherwise, the tool only updates the guard condition of the element if the element is still related to some other features. Overall, the developed PLA modeling environment automatically creates and manages the variation points embedded in the PLA model. It guarantees the correctness of the developed PLA model in terms of the Rule of Thumb described above. This plays an important role in derivation of architecture instances from the PLA model as introduced in the following subsection.

3.3 Derivation of Architecture Instances from PLA

The selector tool can be used to generate an architecture instance from a PLA. Figure 2 shows a screenshot of the tool. When a PLA is opened in the selector, all features defined in the PLA are automatically loaded into the selector. Related information of each feature, such as type (i.e. optional, alternative), default value, and descriptions are also loaded and displayed to the user in a panel. The user can customize the PLA by changing the value

setting of each feature. A drop down list is provided and the user can simply choose the legal values for each feature. 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.

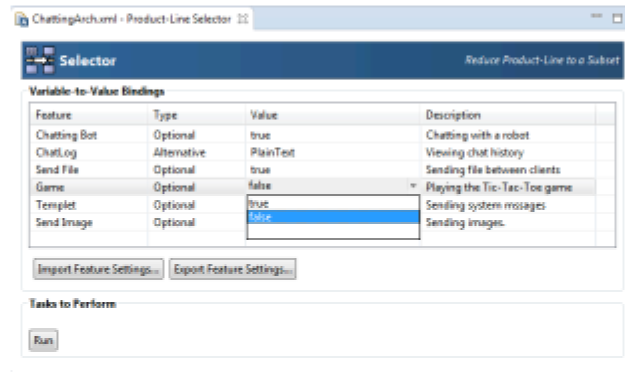

Figure 2 Screenshot of the Selector tool

In addition, the selector includes an import/export function (e.g. the buttons labelled Import Feature Settings, Export Feature Settings in Figure 2) that allows the user to save/load their feature selections. By this means, the user does not need to re-select features if they want to generate an architecture instance that includes the same set of features. They can simply load the setting file that was exported before. Alternatively, the user can also pre-define some recommended system settings such as basic version, professional version, and advanced version. Each version is defined by its own feature configuration file that the user can reuse.

After the feature configuration is done, the user can click the “Run” button in the selector tool to generate the architecture instance. Figure 3 below shows the generated architecture instance of the chatting application based on the feature setting shown in Figure 2. Note that the architecture elements highlighted in Figure 1 do not exist in Figure 3. This is because the corresponding feature (i.e. the Game feature) is not selected (i.e. set to false) in Figure 2

when running the selector tool. Similarly, only the Plain Text Log component is included for the feature ChatLog in the generated architecture. The process of deriving architecture instances involves the activities of evaluating guard conditions and pruning elements. This is further discussed in Section IV.

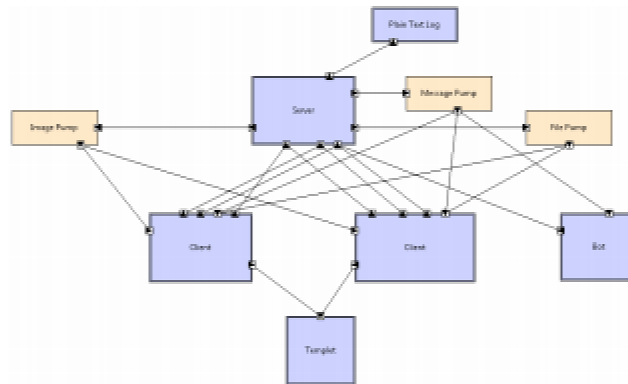

Figure 3. Example of a derived architecture instance

CHAPTER 4

IMPLEMENTATION

The tool presented in Section III is implemented and integrated in ArchStudio, an Eclipse-based architecture development toolset. A main task of the development work is to integrate the functions such as automatic creation of variation points and feature development into ArchStudio. During this process, I was able to reuse several existing tools and code provided by ArchStudio.

4.1 Implementation Environment: ArchStudio

ArchStudio is an architecture development environment integrated within the Eclipse platform as a plug-in project. It supports developing, visualizing, and analyzing architecture models using the xADL language introduced in Section II. On the one hand, ArchStudio has provided a number of ancillary tools, such as Archipelago, ArchEdit, and TypeWrangler [26]. These tools support some essential activities (e.g. visualization, editing, and validation) of architecture development, and can be extended to address new architecture concerns. On the other hand, new tools that are independent of the tools mentioned above can also be built and integrated with ArchStudio for the purpose of some other development activities, such as architecture-implementation mapping [28], architecture-centric traceability [3], and PLA development focused in this study. This process includes developing new xADL schemas and building specific tools to explore new functions.

4.2 Integration with ArchStudio

The main functions of the tool are developed and integrated in 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.

Based on them, our implementation work was mainly focused on developing variability-related functions described in Section III, including 1) creation, removal, and edit of product line features and their relationships to the architecture; 2) automatic maintenance of variation points; 3) visualization of the variation points when a feature is selected; 4) fully automated derivation of architecture instance. The first three functions are implemented in Archipelago, and the last function is done in 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. I made significant changes to the code of Archipelago in this regard. A specific issue I addressed is implementation of alternative features in PLA. Our original plan was to define alternative variation points (e.g. <variant>) correspondingly and embed them in involved architecture elements. For example, there could be an alternative architecture component that includes 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. The problem is that this would require operations such as add/remove variant to be included in Archipelago to modify architecture elements. More importantly, it introduces the product line concepts into the operations of Archipelago. This conflicts with one of our goals in this project – freeing the user from product line specific operations (e.g. creating

variation point) so that they can focus on editing architecture for application logics. As a result, I decide to exploit the expressive power of Boolean guard included 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 [11], 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 terms 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, I made the entire process of architecture instantiation fully automated and feature-oriented. All the features and related information are automatically loaded into the selector tool. The user only needs to select values for each feature from the drop-down list I provided.

CHAPTER 5

RESULTS AND EVALUATION

5.1 Case Study

Case study in software engineering is an empirical research method emphasizing study of an object in its natural context [20]. It is used in this project to assess the effectiveness of the presented approach. I used the work done by Adam Carter and Jeffrey Lanning [30] of modeling feature integrated Architecture model of Apache Solr Project using the tool proposed in this document

5.1.1 Objectives

There are two primary objectives of the case study. First, I want to assess how the presented approach performs with a real software system. Scalability will be our main concern at this point. I 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.

5.1.2 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.

We started the case study by recovering Solr's architecture from its source code. We followed the typical two-step architecture recovery procedure: extraction and abstraction [10]. The first step generates low-level UML diagrams (e.g. class diagrams) by using existing UML tools. The second step groups classes into components, and usually has to be manually done. After that, we used the Apache Solr Reference Guide [22] as a reference and identified a list of important features of Solr. In order to discover the relationship between these features and Solr's architecture elements, we used the UML's sequence diagram generated in the architecture recovery process. This provided us with some clues about where and how a specific feature is implemented in the architecture. There were also cases when we had to manually go through the code to find out how a feature is implemented. After we determined the entire architectural structure, all of the features, and the feature-architecture relationships, we used the tool presented in this paper to model the recovered features and architecture of Solr in ArchStudio. We developed the recovered features, with each feature corresponding to a number of architecture elements. The involved architecture elements are all accompanied by guard conditions as shown in List. 1. At the end, we collected data, analyzed the results, and made a conclusion.

Table I shows the list of the features that we discovered and modeled for Solr. Each feature in the table is depicted by feature type, number of feature variants (not available for optional features), and number of involved architecture elements. We have captured twenty eight features in total. Fourteen of them are optional features, eleven are alternative features, and three are optional-alternative features. The total number of feature variants that the alternative features contain are one hundred and forty three. For example, the feature of Query Processors in the table has sixteen variants, each responsible for handling the work of a certain type of query. The system configuration file specifies which of them should be included and instantiated. The developed architecture model of Solr has one hundred and eighty three components. There are only twenty seven 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 2: Solr's Features and Architecture Mapping

Feature	Type	Number of Variants	Number of Related PLA Elements
Query Caching	Optional	n/a	3
Index Analysis	Optional	n/a	3
Result Clustering	Optional	n/a	5
Faceting	Optional-Alternative	5	11
Function Queries	Optional	n/a	2
Result Highlighting	Optional-Alternative	3	10
Recommendation	Optional	n/a	6
NoSQL Support	Optional	n/a	6
Query Boosting	Optional	n/a	7
Geospatial Search	Optional	n/a	4
Spell Checking	Optional-Alternative	5	12
Statistics Collection	Optional	n/a	6

Query Suggest	Optional	n/a	6
Term-based Query	Optional	n/a	7
SolrCloud	Optional	n/a	3
Extract PDF/Word	Optional	n/a	2
Posting Tool	Optional	n/a	1
Query Processors	Alternative	16	16
Query Handler	Alternative	6	6
Query Parsers	Alternative	24	24
Response Writers	Alternative	9	9
Query Commands	Alternative	2	2
Service Client	Alternative	13	13
Solr Parameters	Alternative	16	16
Update Commands	Alternative	6	6
Update Handlers	Alternative	4	4
Update Loaders	Alternative	4	4 4
Update Processors	Alternative	30	30

Despite the large size of Solr and the number of features we modeled in the case study, all the variation points were automatically created and updated for the involved architecture elements. When we were developing the relationships between features and architecture, we exercised both explicit mode and implicit mode. We did this by changing the order of creating a feature and its related architecture elements. If the feature is created first, the implicit mode is activated and all the new architecture elements we created afterwards were automatically (i.e. implicitly) related to the feature. Otherwise, we would need to right click an architecture element in the editor and explicitly select "Add to Current Feature". As mentioned at the beginning of this section, we validated the established relationships by using the visualization function provided by the developed tool. All these functions worked as designed. In particular, there were several architecture components related to multiple features. For example, Components QueryParser and QueryParameters are both related to

the features of Result Highlighting, NoSQL Support, Term-based Query shown in Table 1. The tool handled this situation well and we did not need to manually intervene. One problem we found in the case study is that the existing feature types are not sufficient for all the situations. Specifically, an OR feature is needed to allow more than one alternative to be selected from a feature set. We plan to include this in our future work.

Additionally, the experience of the case study further shows that it is beneficial to integrate features in the development of PLA. It was relatively 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 included feature visualization function of the modeling environment made it straightforward to review the elements where a specific feature is implemented. In contrast, this would require examination of the entire architecture or even the source code if features and PLA were developed and managed separately.

5.1.3 Threats to Validity

A primary concern that we had about the case study is that we essentially recovered the architecture of an existing software system. In particular, the architecture we developed was for a single software system, rather than a product line. All the features we recovered belong to the same system, instead of different products of a product family. This is a threat to validity of the case study. The case study results described above are valid because 1) converting the architecture of a single system into a PLA is a typical PLA development approach, especially when the product line development did not start from the beginning; 2) all the operations that we exercised in the case study are essential to features of both single system and a product line, such as creating feature, variability modeling, and relating feature

to architecture. 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. This further strengthens the validity of the work done.

CONCLUSION, AVAILABILTY AND FUTURE WORK

This paper presents a pragmatic PLA modeling and instantiation tool. The modeling tool bridges the conceptual gap between abstract features and PLA by integrating their development in a single environment. It maintains and visualizes the mapping from each feature to its implementation in the PLA. The tool can automatically create and maintain variation points in variable architecture elements, and reduces the overhead of manual variability modeling in PLA. The included selector tool fully automates the process of resolving variability in PLA to derive architecture instances.

The source code of the tool can be downloaded from https://github.com/varunnarisetty/Archstudio_PLA. A video demo is available online at http://youtu.be/ZGgx2AA0ALI.

Future work will be focused on modeling the relationships between features (e.g. mutual dependency, mutual exclusion) and automatically enforcing them in the PLA model. I believe the guard condition presented in Section II can be potentially extended to address this issue, based on the Boolean operators such as AND, OR. At this point, the feature list in the development PLA modeling environment is a flat list. After the feature relationship is supported, the feature tree in the environment will contain a hierarchy of features.

REFERENCES

[1] Aizenbud-Reshef, N., Nolan, B.T., Rubin, J., and Shaham-Gafni, Y. Model Traceability. IBM Systems Journal. 45(3), p. 515-26, 2006.

[2] Apache Software Foundation. Apache Solr. http://lucene.apache.org/solr/, 2014.

[3] Asuncion, H.U., Asuncion, A.U., and Taylor, R.N. Software traceability with topic modeling. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1. p. 95-104, ACM: Cape Town, South Africa, 2010.

[4] Batory, D., Sarvela, J.N., and Rauschmayer, A. Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering. 30(6), p. 355-371, June, 2004.

[5] Bosch, J. Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. ACM Press, Addison-Wesley Professional: Reading, Massachusetts, 2000.

[6] Cerner Corporation. http://www.cerner.com/.

[7] Czarnecki, K. and Eisenecker, U. Generative Programming: Methods, Tools, and Applications. Addison-Wesley Professional: Reading, Massachusetts, 2000.

[8] Czarnecki, K. and Antkiewicz, M. Mapping features to models: a template approach based on superimposed variants. In Proceedings of the 4th international conference on Generative Programming and Component Engineering. p. 422-437, Springer-Verlag: Tallinn, Estonia, 2005.

[9] Dashofy, E., van der Hoek, A., and Taylor, R.N. A Comprehensive Approach for the Development of Modular Software Architecture Description Languages. ACM Transactions on Software Engineering and Methodology (TOSEM). 14(2), p. 199-245, April, 2005.

[10] Garcia, J., Ivkovic, I., and Medvidovic, N. A comparative analysis of software architecture recovery techniques. In Proceedings of the Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on. p. 486-496, IEEE. 2013.

[11] Garg, A., Critchlow, M., Chen, P., Van der Westhuizen, C., and van der Hoek, A. An Environment for Managing Evolving Product Line Architectures. In Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2003). p. 358-367, Amsterdam, The Netherlands, September 22-26, 2003.

[12] Heidenreich, F., Kopcsek, J., and Wende, C. FeatureMapper: mapping features to models. In Companion of the 30th international conference on Software engineering. p. 943-944, ACM: Leipzig, Germany, 2008.

[13] Hendrickson, S.A. and van der Hoek, A. Modeling Product Line Architectures through Change Sets and Relationships. In Proceedings of the 29th International Conference on Software Engineering (ICSE 2007). p. 189-198, Minneapolis, MN, May 20-26, 2007.

[14] Jarzabek, S., Bassett, P., Hongyu, Z., and Weishan, Z. XVCL: XML-based variant configuration language. In Proceedings of the Software Engineering, 2003. Proceedings. 25th International Conference on. p. 810-811, 3-10 May 2003, 2003.

[15] Kastner, C., Apel, S., and Kuhlemann, M. Granularity in software product lines. In Proceedings of the 30th international conference on Software engineering. p. 311-320, ACM: Leipzig, Germany, 2008.

[16] Krueger, C.W. The BigLever Software Gears Unified Software Product Line Engineering Framework. In Proceedings of the 2008 12th International Software Product Line Conference. p. 353, IEEE Computer Society, 2008.

[17] Ommering, R.v., Linden, F.v.d., Kramer, J., and Magee, J. The Koala Component Model for Consumer Electronics Software. IEEE Computer. 33(3), p. 78-85, March, 2000.

[18] Parnas, D.L. Designing Software for Ease of Extension and Contraction. IEEE Transactions on Software Engineering. 5(2), p. 128-137, 1979.

[19] Pohl, K., Böckle, G., and van der Linden, F.J. Software Product Line Engineering: Foundations, Principles and Techniques. 1 ed. 468 pgs., Springer: New York, New York, 2005.

[20] Runeson, P. and Host, M. Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering. 14(2), p. 131-164, 2009/04/01, 2009.

[21] Sinnema, M., Deelstra, S., Nijhuis, J., and Bosch, J. COVAMOF: A Framework for Modeling Variability in Software Product Families. In Proceedings of the Third International Software Product Lines Conference (SPLC 2004). p. 197-213, Springer Berlin / Heidelberg. Boston, MA, USA, August 30-September 2, 2004.

[22] Targett, C. Apache Solr Reference Guide. https://cwiki.apache.org/confluence/display/solr/Apache+Solr+Reference+Guide, 2014.

[23] Taylor, R. and van der Hoek, A. Software Design and Architecture: The Once and Future Focus of Software Engineering In Future of Software Engineering 2007, Briand, L.C. and Wolf, A.L. eds. p. 226-243, IEEE-CS Press, 2007.

[24] Taylor, R.N., Medvidovic, N., and Dashofy, E.M. Software Architecture: Foundations, Theory, and Practice. 736 pgs., John Wiley & Sons, 2010.

[25] Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., and Leich, T. FeatureIDE: An extensible framework for feature-oriented software development. Science of Computer Programming. 79(0), p. 70-85, 2014.

[26] University of California Irvine, Institute for Software Research. ArchStudio. http://www.isr.uci.edu/projects/archstudio/.

[27] Yael, D., Julia, R., Thorsten, B., Slawomir, D., Martin, B., and Krzysztof, C. An Exploratory Study of Cloning in Industrial Software Product Lines. In Proceedings of the Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on. p. 25-34, IEEE. 2013.

[28] Zheng, Y. and Taylor, R.N. Enhancing architecture-implementation conformance with change management and support for behavioral mapping. In Proceedings of the 2012 International Conference on Software Engineering. p. 628-638, IEEE Press: Zurich, Switzerland, 2012.

[29] Zheng, Y. and Taylor, R. A classification and rationalization of model-based software development. Software & Systems Modeling. 12(4), p. 669-678, October 2013, 2013.

[30] Adam Carter and Jeffrey Lanning. Modeling Feature integrated Architecture model of Apache Solr Project using Integration features in the development of Software Product line Architecture tool. As a part of Class project for Software Architecture and Design during Fall Semester 2014 at University of Missouri Kansas City.

VITA

Varun Narisetty was born on February 14th 1989, in Andhra Pradesh, India. He completed his Bachelor's degree in Information Technology from JNTU University Hyderabad in 2010. Started his career in a start up and later worked for Samsung Research Institute Delhi. In August 2013 he came to the United States to pursue his Master's degree in Computer Science at University of Missouri-Kansas City (UMKC). During this period he worked under Dr. Zheng as a Graduate Research Assistant.

He is a Software Engineer by profession and a Traveler by passion. Currently living in Kansas City, MO. Still looking for a perfect soulmate.