A Quantitative Evaluation of Change Impact Reachability and Complexity Across Versions of Aspect Oriented Software

Senthil Suganantham, Chitra Babu, and Madhumitha Raju
Department of Computer Science and Engineering, SSN College of Engineering, India

Abstract: *Software developed using a proven methodology exhibits an inherent capability to readily accept the changes in its evolution. This constant phenomenon of change is managed through maintenance of software. By modelling software using Aspect Oriented Software Development (AOSD) methodology, the designer can build highly modularized software that allows changes with lesser impact compared with a non-AOSD approach. Software metrics play a vital role to indicate the degree of system inter-dependencies among the functional components and provide valuable feedback about the impact of changes on reusability, maintainability and reliability. During maintenance, software adapts to the changes in requirements and hence it is important to assess the impact of these changes across different versions of the software. This paper focuses on analysing the impact of changes towards maintenance for a set of Aspect Oriented (AO) applications taken as case study. Existing versions of three AO benchmark applications have been chosen and a set of metrics are defined to analyze the impact of changes made across different versions. An AO Software Change Impact Analyzer (AOSCIA) tool was also developed to study the impact of the changes across the selected versions. It was found that the impact of changes and the related ripple effect is less for AO modules compared to the Object Oriented (OO) modules. Hence, we deduce that the maintainability is improved by adopting the AO methodology.*

Keywords: *AOSD, change propagation reachability, cognitive complexity, software metrics, software maintenance.*

Received September 29, 2013; accepted May 9, 2014.

1. Introduction

Aspect Oriented Programming (AOP) paradigm [9, 14, 19] had proposed a set of constructs with focus on improving the modularity of a software. Modular decomposition is achieved by allowing the separation of concerns that cut across the core functionalities modelled in a software system by encapsulating them into individual units. In order to design a high quality Aspect Oriented (AO) software [16, 24], the software developers need to continuously identify the rationale behind modularizing the functionalities of the system. By doing so, the AO software can evolve into better versions over a period of time. With the emergence of Aspect Oriented Software Development (AOSD), there is an increase in awareness about the observable criteria behind the evolution [20, 21] of concerns. This criterion is a key factor and plays a vital role in the deterioration of software maintainability. A probable cause for these negative characteristics is the increase of scattering and tangling functionalities during the evolution.

Any software system is designed by the software architect to meet a set of clearly defined requirements. These requirements need to be mapped onto the design elements using a specific methodology. It is very often found that requirements are not static and keeps changing due to various factors. In order to effectively deal with the ever changing set of requirements, any software requires necessary features to accommodate the upcoming desired changes over versions.

Maintainability is a key software quality attribute that measures the relative impact of changes made across different versions of software. It can be said that the first rule of maintainability is the ability of software to change or to accommodate changes reflected as versions of the particular software. The second rule of maintenance is the depth of cascading effect due to the changes made to software. In fact, in the context of Object Oriented Software Development (OOSD), many researchers have contributed by studying the ripple effect [3, 12, 13, 17] of changes. Many software have successfully evolved over different versions, because, the designers have provided mechanisms in the design stage that enable the easier inclusion of modular elements. Hence, it can be inferred that all software systems which are able to evolve over multiple versions, have built-in characteristics with positive effect on the ease of maintainability.

Aspects of AOP are modular elements that abstract features which cut across the core and non-core functionalities of a well designed software. AOSD methodology neatly encapsulates concerns that might be tangled and scattered in a software developed using other methodologies. By adopting the AOSD methodology, a software designer is expected to have higher degree of reusability, improved evolution and easier maintenance. The focus of this paper will be the ease of maintenance for software developed using AO methodology. Towards measuring the ease of maintenance, the impact of changes in AO software is analyzed by quantifying them on both AO and non-AO entities over multiple versions.

Software metrics indicate the degree of system interdependence among the components and provide valuable feedback for better reusability, maintainability and reliability. It is important to minimize the ripple effects because of the changes made to the software. In the case of AOP, the identified functionalities are separately modelled as core and cross-cutting concerns. Hence, any change made to the software might have ripple effect in one or both of the concerns. This necessitates the study of the changes made and their impact towards the classes, aspects and their internal constructs. The constructs considered over here are methods of classes and join points, pointcuts and advices of aspects. The impacts of the changes made to these constructs are measured using a new set of metrics. Additionally, two more metrics have been defined which focus on the complexity of weaving an aspect to the base code and complexity of control flow in the base code over the versions. The metrics related to complexity are derived from the weighted class complexity measure [2, 22, 26] defined for measuring the complexity of Object Oriented (OO) applications.

Our work focuses on quantitatively assessing the reachability of changes made to a software during its evolution. The uniqueness of measuring the reachability is by considering both the effect of changes and one level of ripple effect caused by the changes. The following are the major contributions of this quantitative assessment:

- Measurement of the affected constructs namely classes, methods, aspects, pointcuts, join points and advices caused by changes in three AspectJ benchmark applications during its evolution.
- Considering the impact on both direct and one level of indirect constructs over versions of the AspectJ benchmark applications. In the literature, we have not found any work similar to the work done by us for the measurement of the impact of changes in AspectJ programs.
- Additionally, a new set of metrics to capture the complexity of weaving and control flow are defined and applied for the benchmark applications. These metrics are used to quantify the complexity change in association with change impact during evolution.

The remainder of the paper is organized as follows: Section 2 provides the motivation for this work. Section 3 provides a brief explanation on the research objective and scope. Section 4 explains the mathematical concepts and terminologies behind the metrics and also provides a rigorous definition for the proposed metrics. An example that provides motivation behind the study of changes is given in section 5. The case study applications are introduced and elaborated in section 6. The proposed set of metrics has been explained in section 7. Section 8 lists the values obtained for the metrics for the different versions of the AO applications taken as case study. Section 9 provides a detailed analysis and discussion on the metric values for the case studies. Section 10 elaborates on related work in the area of change impact assessment and section 11 concludes and provides future directions.

2. Motivation

The underpinning of a scientific process is the measurement of the effects of applying a methodology to the commonly followed development approach. The pervasive OOSD has been able to provide mechanisms to effectively encapsulate properties and attributes of real world objects. Inspite of this capability, the methodology lacks constructs and mechanisms to encapsulate cross-cutting functionalities into independent units. Inorder to address this deficiency, AOSD has provided new structures to modularize these functionalities. The effect of such modularization needs to be quantitatively assessed using well defined metrics.

Fenton and Pfleeger [10] has clearly asserted the need for measurement to study the impact of using any specific development methodology. Chidamber and Kemerer [6] have proposed design level metrics to measure the impact of using OOSD and it has been extended by several researchers in the literature. One such extension is done by Ceccato *et al.* [5, 29] by proposing extended metrics to measure AOP. Recently, Piveta *et al.* [25] has performed an empirical study with a subset of metrics proposed by Ceccato and Tonella [5] by measuring them for ten projects written in Java and AspectJ programming language. An elaborative explanation indicating the improvements and shortcomings in the quality of AO software has been provided for the case studies.

After a careful investigation of the above-mentioned research works, it becomes evident that researchers have not made an attempt to measure both the direct and indirect impact of making changes to versions of AspectJ benchmark applications. Infact, a work done by Sharafat and Tahvildari [27] measures the probabilities for the direct and indirect impact of changes during the evolution of an OO application. Our work attempts to adapt this approach for measuring the change impact analysis for AspectJ applications. This paper proposes metrics for measuring the direct and one level indirect impact of changes made to AspectJ benchmark applications over their versions. Case study based empirical investigation technique was chosen to study the impact of changes resulting over versions of AspectJ benchmark applications.

3. Problem and Scope

The focus of this research is to quantitatively assess the impact of changes and their ripple effect in AO software. In the open source community, the number of AO based applications is limited. Nevertheless, three open source AO software have been chosen as case study applications. In order to measure the change impact across versions of the chosen case study applications, an exhaustive set of metrics are proposed and evaluated. In addition, a couple of metrics have been defined to understand the complexity of weaving aspects to the base code and the complexity of the flow of control in the base code. A suitable tool called as AO Software Change Impact Analyzer (AOSCIA) is designed and developed to measure using metrics over the available versions. Once the tool obtains the values of the proposed metrics, an inference on the impact on AO maintenance is deduced.

4. Terminologies

Any proposition of metrics requires an in-depth mathematical explanation supporting the validity and reasons behind the need for its existence. The proposed set of metrics is explained using mathematical fundamentals and formalisms. Firstly, the basic terminologies of the AO software are defined using the set notations as given below:

- System: A version of AO software is defined as a system, *S*. Considering one version of a AO software, the set of classes contained in it can be defined as $\{C\}$ and the set of aspects as $\{A\}$. It is understood that an AO software system *S*, is a collection of classes and aspects. Hence, it can be mathematically defined as $S=\{C \cup A\}$.
- Class: In an AO software *S*, a class *C* is a collective set of methods and properties. The set of methods can be defined as $\{M\}$ and the set of properties are defined as $\{P\}$. Thus, for a version of AO software system, a class is a set defined as $C=\{M \cup P\}$.
- Aspects: For an AO software *S*, an aspect *A* is a collection of set of pointcuts defined as $\{PC\}$, set of advices defined as $\{Ad\}$ and a set of introductions defined as $\{I\}$. Hence, for a version of AO software system, an aspect is a collective set and defined as $A=\{PC \cup Ad \cup I\}$. It is possible to have an empty set of pointcuts, advices and introductions in an aspect ie., $PC = \{\emptyset\}$, $Ad = \{\emptyset\}$ and $I = \{\emptyset\}$.
- Directly Affected Units: During the evolution of an AO software *S*, consider S_n as the n^{th} version and S_{n+1} as $n+1^{th}$ version. In the later version the directly affected units are the classes, methods, aspects, pointcuts and advices that are directly affected by a change *c*. As an example, if a method is added to a class because of a change in the requirement, then all or some methods will be directly affected within that class. A motivating example of a directly affected aspect with practical explanation is given in section 5.
- Indirectly Affected Units: Consider the evolution given in the definition of directly affected units. The indirectly affected units are the classes, methods, aspects, pointcuts and advices that are affected due to the effect of changes in the directly affected units. If a method is added to a class *A*, then other classes which have access to this method will be indirectly affected and are considered as indirectly affected classes.

Secondly, the foundation for the proposed metrics are explained and illustrated as given below:

- Change Propagation Reachability (CPR): In a multi-version AO software, consider all the changes that occur over its versions. The *CPR* of a unit (method, pointcut, joinpoint and advice) for a change *c* can be defined as the inverse of the count of elements in a set, whose elements are the union of the set of directly U_d and set of indirectly U_i affected units. *m* and *n* are the number of directly and indirectly affected units, respectively. The mathematical expression for the same is shown in Equation 1.

$$CPR(c) = \frac{1}{\sum_{d=1}^{m} |\{U_d\}| + \sum_{i=1}^{n} |\{U_i\}|} \qquad (1)$$

Two additional set of metrics are also proposed to capture the complexity of the process of weaving and control flow. The generalized form of the proposed metrics is defined mathematically as given below:

- Complexity of Process (CP): The two parts of code in AOP are: Base code-code implementing core concerns (Classes of Java) and, cross-cutting code-code implementing cross-cutting concerns (Aspects of AspectJ). The control flow in the base code happens through the method invocations and method return type is considered while calculating the complexity of control flow. Similarly, while considering the cross-cutting code, the join point designators and their respective number of occurrences need to be considered to compute the complexity of weaving. Both the processes can be generalized and expressed using Equation 2. In the formula, *n* is the number of categories of method return types or join points, CV_i is the assigned complexity value and T_i is the number of methods or join points of a type *i*.

$$CP = \sum_{i=1}^{n} CV_i \times \left|\{T_i\}\right| \quad (2)$$

All the proposed metrics satisfy Briand's *et al.* [4, 7] mathematical properties for measurement. An explanation of the four properties and their validation of the change propagation reachability metric are given below:

1. Non-Negativity: The change propagation reachability, *CPR* of the calculated units will never be negative. Since, the number of affected units can possibly be ≥ 0, the sum of the same will always lead to a positive number. Hence, it can be inferred that respective values of *CPR* will always be ≥ 0.

   $CPR(c) \geq 0$
2. Null Value: The value of the change propagation reachability will be 0, if the number of affected units is empty. ie., if no units are affected by the change *c*, then the respective value of *CPR* will be 0.

   if $\{Affected\ Units\} = \{\emptyset\}$ *then* $CPR(c) = 0$
3. Monotonicity: If the number of units, namely, pointcuts, advices and methods are increased over versions, then the respective value of *CPR* (for classes, aspects, pointcuts, advices and methods) will increase or remain the same. ie., if S_2 is the second version of the AO software S_1 then the *CPR* of a unit in the second version will be more or equal to the *CPR* of the same unit in the first version.

   $CPR(c)$ for a unit in $S_2 \geq CPR(c)$ for a unit in S_1
4. Non-Decreasing Monotonicity: Consider m_1 and m_2 as two different classes or aspects with no common relationship, then by merging them together will not reduce the *CPR* values across two versions of the AO software.

   $CPR(c)$ of m_1+$CPR(c)$ of m_2 is $\geq CPR(c)$ of m_1+m_2

A similar explanation is possible for the complexity of process metric CP to show that it also obeys all the four mathematical properties suggested by Briand *et al.* [4].

5. Motivating Example

In order to understand the impact of changes on the different versions of AspectJ benchmark applications, a simple example is shown in Figures 1 and 2.

```
public class savingsAccount extends Account {
double balance = 0;

savingsAccount(double initialDeposit) {
      setBalance(initialDeposit); }
private setBalance(double deposit) {
      balance = balance + deposit; }
public withdraw(double amount) {
      setBalance(-amount); }
public deposit(double amount) {
      setBalance(amount); }              }
public aspect taxCalculator          {
pointcut accessTax(SavingsAccount t, double amt) : call(*
*.withdraw(*)) ||
              call(* *.deposit(*)) &&
              args(amt) &&
              target(t);
after() : accessTax(amount) {
      t.setBalance(-10.00); }        }
public aspect logger {
pointcut log() : call(* *.withdraw(*) ||
                 call(*.*.deposit(*);
after() : log() {
// routine for logging the changes  } }
```

Figure 1. Example of computing tax (version 1) and logging.

```
public aspect taxCalculator {
pointcut accessTax(SavingsAccount t, double amt) : call(*
*.withdraw(*)) ||
              call(* *.deposit(*)) &&
              args(amt) &&
              target(t);
pointcut serviceTax(SavingsAccount t, double amt) : call(*
*.withdraw(*)) ||
              args(amt) &&
              target(t);
after() : accessTax(amount) {
      t.setBalance(-1.00); }

after() : serviceTax(amount) {
      if(amount > 10000)
           t.setBalance(-1.00); }  }
```

Figure 2. Example of computing tax (version 2).

The code given in Figure 1 is the first version of taxCalculator aspect with a single accessTax pointcut and an aspect that encapsulates the non-functional requirement namely, logging. The accessTax pointcut weaves an after() advice which deducts US$ 1 after the execution of withdraw() and deposit() methods in the base code and the log pointcut weaves another after() advice to record the changes for the historical purpose.

The second version of the code is given in Figure 2 with one more pointcut serviceTax added to the taxCalculator aspect. The advice of this pointcut additionally deducts US$ 1 as service tax after the execution of withdraw() only if the withdrawn amount is more than US$ 1000.

Hence, based on the metrics proposed in this paper, while considering these new pointcuts from one version to another version it is counted as direct impact on the joinpoint where these advices are weaved in the base code. If another aspect defines a pointcut at the same joinpoint then it will be calculated as indirectly affected construct for the metrics evaluation. The log() pointcut and its after() advice are counted as units that are indirectly affected by the addition of serviceTax() pointcut. A similar type of calculation is done while looking at the classes in the benchmark applications.

6. Case Study Applications

Quantitative evaluation of metrics requires case study applications. Three different AO applications namely spacewar, tracing and bean which are part of the AspectJ development toolkit [8] have been identified as case study applications to evaluate the impact of changes across versions.

6.1. Spacewar

Spacewar is a classic video game that is intended to provide a modest-sized example of a program that uses aspects. The code for this example is still evolving with the addition of new features to AspectJ and also with a better understanding of how to use the features.

When the game starts, there will be two different displays. These are two built-in display aspects of the game. In each display, there will be a single white ship and two red ships. The white ship is for the user to control; the red ships are enemy robots. The white ship is controlled with the four arrow keys to turn, thrust and stop and the spacebar is used as a firing weapon. As the user continues to play, the game will be displayed in both windows. The user can quit the game by pressing the Ctrl-Q key in the keyboard. The numbers of aspect and class elements are given in Table 1.

Table 1. Case studies-count of classes, aspects and their numbers.

Version	Classes	Methods	Aspects	Pointcuts	Advices
Spacewar					
1	12	123	-	-	-
2	17	159	7	6	22
Tracing					
1	5	36	1	3	4
2	5	42	3	7	8
3	5	42	4	10	10
Bean					
1	2	11	-	-	-
2	2	17	1	1	3

6.2. Tracing

Tracing is an example that is shipped with the standard AspectJ Development Tools (AJDT) plug-in and is used to create graphical objects like circle and square. In software engineering, tracing is specially used for logging or recording information about a program's execution. This information is typically used by programmers for debugging purposes. Additionally, depending on the type and detail of information contained in a trace log, the experienced system administrators or technical support personnel use software monitoring tools to diagnose common problems with the software. Tracing functionality is modelled using aspects as it is a cross-cutting concern in the application. Table 1 shows the number elements in the different available versions.

6.3. Bean

Bean is a simple example that shows the method of converting a class into a Java Bean. Adding bound properties and serialization to point objects are modelled as aspects. The count of the aspect and base entities are specified in Table 1.

7. Proposed Metrics

The proposed set of metrics separately captures the changes by identifying the impact of the changes made to aspect elements as well as to the base elements (class elements). While evaluating the respective metric values for the first version of the selected case study applications, all the classes and aspects are considered as changed elements. In the subsequent version, the additions from the previous version are considered for the computation of metric values.

7.1. Metrics for Change in Aspects and its Elements

The various changes that are possible in an aspect are changes in aspects, advices and pointcuts, calculation of *ACPR*(*c*):

- *Step* 1: Calculate total #of pointcuts and advices directly affected by change *c*.
- *Step* 2: Calculate total #of pointcuts and advices indirectly affected by change *c*.
- *Step* 3: Add all the calculated values and find the reciprocal of this to obtain the value of *ACPR*(*c*) in the range between 0 and 1.

Calculation for the other metrics can follow similar set of steps by counting the respective directly and indirectly affected entities for a particular change *c*.

7.1.1. Aspect Change Propagation Reachability

Aspect CPR (ACPR) of a change *c* is calculated using formula 3.

$$ACPR(c) = \frac{1}{\sum_{d=1}^{m}\left[\#P_d + \#Ad_d\right] + \sum_{i=1}^{n}\left[\#P_i + \#Ad_i\right]} \tag{3}$$

Where *ACPR*(*c*) stands for *ACPR* of a change *c*, *m* is the number of aspects directly affected by *c*, $\#P_d$ is the number of pointcuts within the d^{th} directly affected aspect, $\#Ad_d$ is the number of advices within the d^{th} directly affected aspect, *n* is the number of aspects indirectly affected by *c*, $\#P_i$ is the number of pointcuts within the i^{th} indirectly affected aspect, and $\#Ad_i$ is the number of advices within the i^{th} indirectly affected aspect.

7.1.2. Pointcut Change Propagation Reachability

Pointcut CPR (PCPR) of a change *c* is evaluated as specified in formula 4.

$$PCPR(c) = \frac{1}{\sum_{d=1}^{m}\left[\#J_d + \#Ad_d\right] + \sum_{i=1}^{n}\left[\#J_i + \#Ad_i\right]} \tag{4}$$

Where *PCPR*(*c*) stands for *PCPR* of a change *c*, *m* is the number of pointcuts directly affected by *c*, $\#J_d$ is the number of join points for the d^{th} directly affected pointcut, $\#Ad_d$ is the number of advices for the d^{th} directly affected pointcut, *n* is the number of pointcuts indirectly affected by *c*, $\#J_i$ is the number of join points for the i^{th} indirectly affected pointcut, and $\#Ad_i$ is the number of advices for the i^{th} indirectly affected pointcut.

7.1.3. Advice Change Propagation Reachability

Advice CPR (ADCPR) of a change *c* is calculated using formula 5.

$$ADCPR(c) = \frac{1}{\sum_{d=1}^{m}[\#J_d] + \sum_{i=1}^{n}[\#J_i]} \quad (5)$$

Where *ADCPR*(*c*) stands for *ADCPR* of a change *c*, *m* is the number of advices directly affected by *c*, $\#J_d$ is the number of join points for the d^{th} directly affected advice, *n* is the number of advices indirectly affected by *c*, and $\#J_i$ is the number of join points for the i^{th} indirectly affected advice.

7.2. Metrics for Change in the Class and its Elements

The various changes that are possible in the base code are changes to classes and methods measured using class and method change propagation reachability respectively.

7.2.1. Class Change Propagation Reachability

The Class CPR (CCPR) value of classes is calculated using formula 6.

$$CCPR(c) = \frac{1}{\sum_{d=1}^{m}[\#M_d] + \sum_{i=1}^{n}[\#M_i]} \quad (6)$$

Where *CCPR*(*c*) stands for *CCPR* of a change *c*, *m* is the number of classes directly affected by *c*, $\#M_d$ is the total number of methods in the d^{th} directly affected class, *n* is the number of classes indirectly affected by *c*, and M_i is the total number of methods in the i^{th} indirectly affected class.

7.2.2. Method Change Propagation Reachability

Finally, the CPR value of Methods (MCPR) within classes in AO software is calculated as specified by the formula 7. This metric is different from *CCPR* and considers only the affected methods of changes over versions of AO software.

$$MCPR(c) = \frac{1}{\sum_{d=1}^{m}[\#M_d] + \sum_{i=1}^{n}[\#M_i]} \quad (7)$$

Where *MCPR*(*c*) stands for *MCPR* of a change *c*, *m* is the number of classes directly affected by *c*, $\#M_d$ is the number of directly affected methods in the d_{th} directly affected class, *n* is the number of classes indirectly affected by *c*, and $\#M_i$ is the number of indirectly affected methods in the i^{th} indirectly affected class.

7.3. Complexity Estimation

Metrics such as *ACPR*, *PCPR*, *ADCPR*, *MCPR* and *CCPR* have been proposed to measure the impact of changes in the AO applications identified as case study. Extending the impact of changes in order to analyze the behavior of various aspect and base elements (during the maintenance phase of a software), two additional metrics have been proposed to evaluate the cognitive complexity of weaving and control. Complexity values have been assigned to the designators specified in the aspects and method return types used in the classes. While assigning the values, the most complex designator used in the pointcuts of the aspects and the most complex return type found in the methods of classes are assigned the same complexity value. Similarly, a complexity value of 1 is assigned for the least complex designator and method return type of the respective elements. This assignment enables to compare the complexity of weaving to the complexity of control flow of AO software.

7.3.1. Complexity of Weaving

Weaving of aspects is normally done by executing the advices defined in the aspect at a resolved join point. The complexity of this process Complexity of Weaving (CW) can be evaluated by the equation specified by Formula 8. The complexity value assigned for the different designators in AOP is specified in Table 2. The commonly occurring designators are ordered based on the cognition required to resolve the join points. Further, a value reflecting the cognitive complexity has been assigned to the ordered designators. The Join Point Complexity (JPC) value assigned to the initialization designator is higher than this designator. This is because the cognitive complexity involved in the initialization of objects is higher than simply identifying the current object. In Table 4, the designator cflow has the maximum *JPC* because the flow of control requires a very high cognizance compared to the other designators.

Table 2. Complexity of join point types.

Type of Designator	JPC
This	1
Initialization	1
Call	2
Execution	3
Handler	4
Target	5
Within	5
Cflow	6

$$CW = \sum_{i=1}^{n}(JPC_i \times \#JP_i) \quad (8)$$

Where *n* is the total number of categories of join points, JPC_i is the cognitive complexity value assigned for the i^{th} designator type of the matching joinpoint, $\#JP_i$ is the number of join points of a listed type *i*.

7.3.2. Complexity Of Control Flow (Through Method Return Types)

The Complexity of Control Flow (CCF) of the class methods defined in the AO application can be calculated by the return type of each defined method. This metric can be calculated using formula 9. Similar to the way of assigning complexity values to pointcut designators, control complexity values are assigned for methods in classes. The assigned complexity values are ordered and listed in Table 3.

Table 3. Complexity of method return types.

Method Return Type	MRTC
void	4
primitive	5
class	6

$$CCF = \sum_{i=1}^{n} \left(MRTC_i \times \#M_i \right) \qquad (9)$$

Where n is the total number of method return types, $MRTC_i$ is the cognitive complexity value assigned for the i^{th} return type of methods, and $\#M_i$ is the number of methods of a listed type i. The system design that captures the workflow of the AOSCIA tool is shown in Figure 3. The tool takes the paths containing the three case study applications as input through the interfaces in the tool. It then sorts the files in the given folder into Java and AspectJ files. The cross-references between the aspects and classes are identified to find the impact of changes in the current version. The respective signatures are further extracted and the values of the proposed metrics are computed. The computed values are further compared across versions of the case study applications. Finally, the impact of evolution over versions of AO software towards maintainability is inferred based on the metric values.

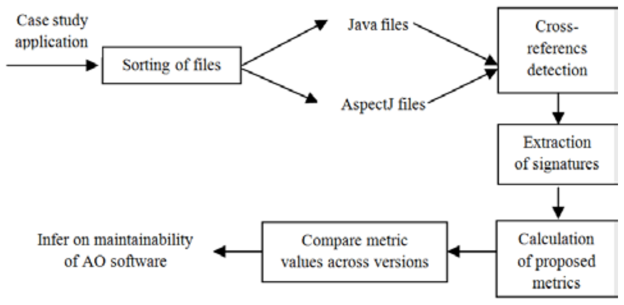

Case study application
Sorting of files
Java files
AspectJ files
Cross-references detection
Extraction of signatures
Calculation of proposed metrics
Compare metric values across versions
Infer on maintainability of AO software

Figure 3. Workflow system design of AOSCIA tool.

8. Tool Results for Case Study Applications

The proposed metric values, both *CPR* and complexity values are evaluated for the case study applications using to the AOSCIA tool.

8.1. Tracing

8.1.1. Change Propagation Reachability Metrics

For the different versions of tracing application, the values of the different *CPR*s are evaluated and the average of the *CPR* values are calculated which are tabulated in Table 4.

Table 4. Case study-average CPR values of versions.

Version	Avg. CCPR	Avg. MCPR	Avg. ACPR	Avg. PCPR	Avg. ADCPR
Tracing					
1	0.2932	0.3154	0.1428	0.8450	0.2851
2	0.3111	0.2836	0.1226	0.7500	0.2236
3	0.3111	0.2836	0.1277	0.6250	0.2136
Spacewar					
1	0.1720	0.1641	-	-	-
2	0.2126	0.2632	0.2434	0.5029	0.7292
Bean					
1	0.1964	0.0647	-	-	-
2	0.2135	0.0836	1.0000	1.0000	0.3242

8.1.2. Method/Joinpoint Analysis

The complexity values have been found for all the three versions of tracing case study application and are tabulated in Tables 5 and 6.

Table 5. Complexity of control flow (tracing).

Method Return Type	MRTC	Count	CCF
Version 1			
Void	4	23	92
Primitive	5	9	55
Class	6	4	36
Average CCF			183/36 = 5.03
Versions 2 and 3			
Void	4	29	116
Primitive	5	9	55
Class	6	4	36
Average CCF			207/42 = 4.93

Table 6. Complexity of weaving (tracing).

Pointcut Designators	JPC	Count	CW
Version 2			
Execution	3	2	6
Within	5	2	10
Average CW			16/4 = 4.0
Version 3			
Execution	3	2	6
Within	5	2	10
Average CW			16/4 = 4.0

8.2. Spacewar

8.2.1. Change Propagation Reachability Metrics

The code of the second case study application namely Spacewar was given as input to the AOSCIA Tool and the values of *CPR* metrics were measured and the average is computed and tabulated in Table 4.

8.2.2. Method/Joinpoint Analysis

Similar to the previous case study, the complexity values are also computed for the versions of spacewar application and are tabulated in Tables 7 and 8.

Table 7. Complexity of control flow (spacewar).

Method Return Type	MRTC	Count	CCF
Version 1			
Void	4	78	312
Primitive	5	15	75
Class	6	30	180
Average CCF			567/123 = **4.6**
Version 2			
Void	4	97	388
Primitive	5	19	95
Class	6	43	258
Average CCF			741/159 = **4.66**

Table 8. Complexity of weaving (spacewar-version 2).

Pointcut Designators	JPC	Count	CW
Initialization	1	1	1
Call	2	15	30
Execution	3	2	6
Target	5	3	15
Average CW			52/21 = 2.47

8.3. Bean

8.3.1. Change Propagation Reachability Metrics

Similar to the spacewar and tracing applications, the *CPR* values of the bean case study is computed using the AOSCIA tool. The obtained values are tabulated in Table 3.

8.3.2. Method/Joinpoint Analysis

Since, the Bean case study has two different versions, the complexity values are evaluated for each version and tabulated in Tables 9 and 10.

Table 9. Complexity of control flow (Bean).

Method Return Type	MRTC	Count	CCF
Version 1			
Void	4	7	28
Class	6	2	12
Primitive	5	2	10
Average CCF			50/11 = 4.54
Version 2			
Void	4	13	52
Class	6	2	12
Primitive	5	2	10
Average CCF			74/17 = 4.35

Table 10. Complexity of weaving (bean-version 2).

Pointcut Designators	JPC	Count	CW
Call	2	3	6
Average CW			6/3 = 2.0

9. Discussion on Measurements

Numerical values of the change propagation reachability metrics were obtained using the AOSCIA tool for the different versions of the case study applications. Further, the tool computed values of the proposed metrics were analysed to infer on the reachability of the impact of changes in the chosen versions. The complexity metrics for weaving and control flow were also analyzed to understand the impact of changes in the versions of the applications. The observations of variation in metric values for the versions are explained in the subsections given below.

9.1. Change Propagation Reachability: Tracing

Version 1 is programmed with 5 classes and 36 methods. The respective *CPR* values calculated for average *CCPR* and average *MCPR* are 0.2932 and 0.3154. Versions 2 and 3 are modeled with the same 5 classes and extended methods to increase the count to 42. Consequently, the average *CPR* values are calculated as 0.3111 and 0.2836. Even though the number of classes is the same in versions 2 and 3, since there is a change in the total number of methods, the average *CCPR* values are different and found to be increased. By adding methods to the existing classes, the *CCPR* value has been increased and thereby increasing the ripple effect. This reduces the maintainability of tracing application with respect to its modeled classes in latter versions.

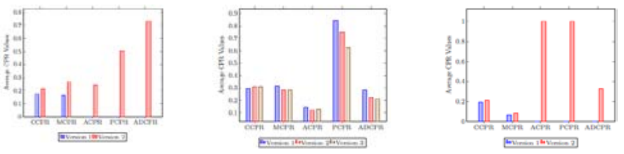
Considering the AO constructs, version 1 consists of a single first class cross-cutting entity (aspect) with 3 pointcuts and 4 advices. The corresponding *CPR* values calculated are 0.1428, 0.845 and 0.2851 respectively. Version 2 has the inclusion of 2 aspects which includes an abstract aspect. This is reflected by the change in the average values of *CPR* for aspects and their constructs. The respective values of average *CPR*s for aspects, pointcuts and advices are 0.1226, 0.75 and 0.2236. The last version (version 3) of tracing is programmed with 4 aspects, 10 pointcuts and 10 advices and the evaluated average *CPR* values are 0.1277, 0.625 and 0.2136 respectively. Considering the elements modelling the cross-cutting concerns, there is a steady increase in the number of aspects, pointcuts and advices. Analyzing the decrease of the average values of *CPR*, it is clearly evident that the values are decreasing over versions. The average *CPR* values of aspects and advices have also stabilized in version 2 and there are only minimal respective increases in the last version. It can be inferred that the ripple effects caused by the changes in the aspects are minimized in the later version thereby improving the maintainability of AO constructs.

From the graph shown in Figure 4-b, it is clearly evident that there is a clear fall in the respective average *CPR* values of AO constructs, while the average *CPR* values of OO constructs are slightly increasing and stabilizing in latter versions.

9.2. Change Propagation Reachability: Spacewar

In version 1 of the spacewar application, there are 12 classes and 123 methods and the calculated average values of *CPR* are 0.1720 and 0.1641 respectively. The number of classes and methods increases to 17 and 159 in version 2 and the respective average values of *CPR* are calculated as 0.2126 and 0.2632. The average *CPR* values of OO classes and methods in version 2, are higher than that of version 1. This increase of values will in turn increase the ripple effect and have a negative impact on the maintainability of the application considering the modelled OO elements.

Considering the AO specific entities, version 1 does not contain any aspects whereas version 2 is modeled with 7 aspects, 6 pointcuts and 22 advices. The respective average values of *CPR* are computed as 0.2434, 0.5029 and 0.7292. As far as the AO elements are concerned, since, only one version containing aspects is available, not much about the change impact can be inferred upon. However, version 2 contains aspects that model both functional requirements like coordination, display, objectpainting and non-functional requirements like synchronization and debug, we can conclude that a good number of scattering and tangling concerns have been encapsulated as aspects.



a) Spacewar. b) Tracing. c) Bean.

Figure 4. Average CPR over versions.

The graph in Figure 4-a clearly shows the increase in the average values of *CPR* of the OO entities. Since, version 2 models all the possible cross-cutting functional and non-functional requirements as aspects, the average *CPR* values of the AO elements are high.

9.3. Change Propagation Reachability: Bean

The numbers of OO elements in version 1 are 2 classes and 11 methods and the computed average values of *CPR* are 0.1964 and 0.0647. In version 2, the number of classes remains the same as in version 1, whereas the number of methods is increased to 17. Correspondingly, the average values of *CPR* are calculated as 0.2135 and 0.0836. The average *CPR* values of classes and methods are clearly increasing over the respective versions. This can be considered as a negative characteristic considering the evolution of the plug-in. The ripple effect will increase in the latter versions and hence will reduce the maintainability of OO elements.

There are no AO specific elements in version 1, but considering version 2, there is only 1 aspect, 1 pointcut and 3 advices. The corresponding average values of *CPR* are calculated as 1.0, 1.0 and 0.3242 respectively. Since only one aspect with a single pointcut has been included in version 2, it is very difficult to clearly study the impact of ripple effect. However, the single aspect models a functional requirement BoundPoint of the bean plug-in and there are only 2 classes in version 2. All the possible cross-cutting concerns have been modelled as aspects.

The graph shown in Figure 4-c clearly identifies the increase in OO element's average *CPR* values.

9.4. Weaving and Control Flow Complexity: Tracing

The values of *CCF* in three versions are 5.03, 4.93 and 4.93. There is no difference in the values obtained for versions 2 and 3, because no OO elements are added to the later version.

While considering the AO specific entities, the aspects in version 1 are modelled as abstract entities and hence do not contain any designators and consequently the value for *CW* is 0. Version 3 contains 4 aspects, 10 pointcuts and 10 advices and version 2 contains 3 aspects, 7 pointcuts and 8 advices. The increases in elements are modelled as abstract entities and hence no designators are attached to the pointcuts. Consequently, there is no increase in the value of *CW* from version 2 to version 3.

Considering the respective values of *CCF* and *CW* over versions, it is clearly evident that complexity of control flow is higher than that of weaving. The change in the values of *CCF* and *CW* is clearly depicted in the graph shown in Figure 5-b.

9.5. Weaving and Control Flow Complexity: Spacewar

The numbers of OO elements modelled in version 1 are 12 classes and 123 methods and the corresponding *CCF* value is 4.6. Considering version 2, there is an increase in the number of classes and methods and the evaluated value *CCF* is 4.66. By carefully looking at the number of classes and methods, it is clearly evident that methods with different return types are proportionately increased while comparing version 2 with version 1. Hence, based on the number of functionalities that are modelled in the later version of the application, we can infer that the complexity value is reasonably stabilized.

There are no aspects in version 1, but version 2 is modelled with 7 aspects, 6 pointcuts and 22 advices. Hence, the value of *CW* in version 2 is 2.47.

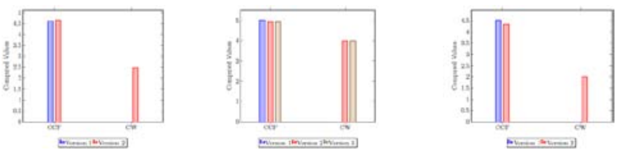
Again, while comparing the values of *CW* and *CCF* with respect to each version, the cognitive complexity attached with weaving of aspects is always lower than the cognitive complexity of control flow. The values of *CCF* and *CW* over versions are plotted in the bar graph shown in Figure 5-a and it clearly shows the difference between *CCF* and *CW* over versions.

9.6. Weaving and Control Flow Complexity: Bean

The number of classes and methods in version 1 is 2 and 11 and the value of *CCF* is 4.54. Version 2 is modelled with 2 classes and 17 methods and the calculated value of *CCF* is 4.35. There is a minor decrease in the value of cognitive complexity of control flow because of the increase in the number of methods with void as return type.

Version 1 is not modelled with any aspect and version 2 has a single aspect with a pointcut. The evaluated value of *CW* is 2.0 for this version. Even though the number of aspects is only one, while looking at the fact that only 2 classes are part of the version, the cognitive complexity of weaving is very less compared to the cognitive complexity of control flow. By modelling the cross-cutting functionality as aspect, the cognitive complexity of the plug-in has been reduced in version 2.

The *CCF* and *CW* values have been plotted in the graph shown in Figure 5-c and it is clearly seen that weaving complexity is less than the control flow complexity in version 2.



a) Spacewar. b) Tracing. c) Bean.

Figure 5. Complexity of weaving and complexity of control flow.

10. Related Works

In order to understand the ripple effect of changes made to a software, researchers have proposed methodologies to infer the impact of changes on maintenance of OO or AO software.

Zhang *et al.* [30] proposes a new change impact analysis technique for AspectJ programs. This technique captures the semantic difference between versions of 8 AspectJ benchmarks. Call graphs are used to identify the atomic changes and a catalogue of atomic changes has been defined for the AspectJ programs. It is an extension of Ryder's method for measuring the atomic changes in OO programs. The indirect impacts of changes over versions have not been captured and the focus was only on measuring the atomic changes.

Shen *et al.* [28] has proposed a fine-grained coupling metrics suite that captures the atomic changes in the aspect and the base code. Further, these coupling metrics are correlated with the maintainability of the AO software and a correlation analysis has been performed to understand the impact of software changes.

Sharafat and Tahvildari [27] has developed a model to predict the probability that a class will change in the future based on the information collected through successive generations of given OO software. This probability is obtained by adding the probability of internal changes in a class and external changes that affect other related classes.

Munoz *et al.* [23] has defined group of metrics to measure the usability and testability of *AOP*. Evolution is considered and the metrics are applied for a sample case study AJHealthWatcher. A theoretical framework based on Briand's formalism for analysing OO programs has been developed for AO programs. Based on the proposed framework, metrics have been defined to capture the usage of Pointcut Designators (PCDs). Even the proportion of invasive aspects in the sample case study has been analysed to throw insight into the effect of aspectization. It was found that the Response For a Module (RFM) metric defined by Chidamber and Kemerer [6] has improved the transitive closure of RFM, identified by reduction of RFM'. This increase is due to one more level of redirection that occurs because of the introduction of aspects of AOP. This increase negatively affects the testability and the maintainability of AO software. Evolution of the AO software is not considered while measuring the RFM metric. During the evaluation of *CPR*, evolution over versions is considered since it directly affects the maintainability of software.

A method to identify the potential cross-cutting functional and non-functional requirements in the early stages of software development has been attempted by Amirat *et al.* [1]. The benefit of adopting this approach is achieve better traceability of broad set of requirements thereby improving the maintainability of AO software during its evolution.

An empirical study on four metrics defined by Chidamber and Kemerer [6] and two metrics defined by Ceccato and Tonello [5] were measured for ten different Java and AspectJ projects by Piveta *et al.* [25]. Lessons learnt for the six metrics have been analysed in detail with discussion on *AOP*. Evolution is not captured in the analysis since only one version was considered for each project.

Kumar *et al.* [15] has measured the changeability of AO software by looking at the consequence of changes made to its modules. It has been inferred that this kind of software is able to easily absorb changes by computing the values for some of the design level metrics defined to measure the AO software.

Li *et al.* [18] has performed change impact simulation to assess the propagation of the changes by measuring the atomic level changes related to the modifications in the object oriented software. The different types of dependency relationships that are possible between classes are defined using a proposed software change propagation model. This concept of change propagation reachability can be extended to AO software.

Figueiredo *et al.* [11] is towards analyzing the design stability of two aspectual Software Product Lines (SPLs) by focussing on modularity, change propagation and the dependency between different features in the test applications. This study measures only the aspects, classes, operations, lines of code and pointcuts across versions of SPLs and does not count the number of advices and join points as well as the indirect effect of the changes across versions.

11. Conclusions and Future Directions

The changes made across the versions of any software significantly influences the effort involved in maintaining that software. With this in mind, this paper had proposed a group of metrics to measure the reachability of changes made to an AO software over the available versions. An additional set of metrics were also proposed to identify and measure the cognitive complexities of both the classes and aspects that model the base and cross-cutting functionalities respectively. Based on the measurement, AO constructs exhibited better maintenance characteristics over versions than the OO constructs.

Three different case study applications with multiple versions were identified and the values of the metrics for the respective versions were measured. Based on the measured metric values, a set of inferences were derived with logical explanations on the impact of changes, considering the objects, aspects and their internal elements.

This work can be further extended by investigating more case study applications having higher number of versions to analyze the impact of changes. Also, an analysis can be done to independently relate the changes made to functional and non-functional requirements and their impact on aspects and classes in the versions of an application. This will in turn provide a software developer to have a better insight on the change propagation reachability and subsequently on the maintenance in the design stage of software development.

References

[1] Amirat A., Laskri M., and Khammaci T, "Modularization of Cross-cutting Concerns in Requirements Enginering," *The International Arab Journal of Information Technology*, vol. 5, no. 2, pp. 120-125, 2008.

[2] Arockiam L. and Aloysius A., "Attribute Weighted Class Complexity: A New Metric for Measuring of OO Systems," *World Academy of Science, Engineering and Technology*, vol. 5, no. 10, pp. 1151-1156, 2011.

[3] Black S., "Computing Ripple Effect for Software Maintenance," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 13, no. 4, pp. 263-279, 2001.

[4] Briand L., Morasca S., and Basili V., "Property-based Software Engineering Measurement," *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 68-86, 1996.

[5] Ceccato M. and Tonella P., "Measuring the Effects of Software Aspectization," *1st Workshop on Aspect Reverse Engineering (WARE)*, 2004.

[6] Chidamber S. and Kemerer C., "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476-493, 1994.

[7] Dubey S. and Rana A., "Assessment of Maintainability Metrics for Object-oriented Software System," *ACM SIGSOFT Software Engineering Notes*, vol. 36, no. 5, pp.1-7, 2011.

[8] Eclipse Foundation, http://www.eclipse.org/ajdt/, Last visited 2013.

[9] Elrad T., Filman R., and Bader A., "Aspect-oriented Programming: Introduction," *Communications of the ACM*, vol. 44, no. 10, pp. 29-32, 2001.

[10] Fenton N. and Pfleeger S., *Software Metrics: A Rigorous and Practical Approach,* PWS Publishing Co., 1997.

[11] Figueiredo E., Cacho N., Sant'Anna C., Monteiro M., Kulesza U., Garcia A., Soares S., Ferrari F., Khan S., Filho F., and Dantas F., "Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability," *in Proceeding of 30th International Conference on Software Engineering (ICSE)*, USA, pp. 261-270, 2008.

[12] Haider B. and Black S., "Ripple effect: A Complexity Measure for Object Oriented Software," *in Proceeding of European Conference on Object Oriented Programming*, Germany, 2007.

[13] Haider B. and Black S., "Using the Ripple Effect to Measure Software Quality," *in Proceeding of International Conference Software Quality Management*, UK, pp. 183, 2005.

[14] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J., and Irwin J., "Aspect-oriented Programming," *in Proceeding of European Conference on Object Oriented Programming,* pp. 220-242, 1997.

[15] Kumar A., Kumar R., and Grover P., "An Evaluation of Maintainability of Aspect-oriented Systems: A Practical Approach," *International Journal of Computer Science and Security,* vol. 1, no. 2, pp. 1-9, 2007.

[16] Laddad R., "Aspect-Oriented Programming will Improve Quality," *IEEE Software,* vol. 20, no. 6, pp. 90-91, 2003.

[17] Li B., Sun X., Leung H., and Zhang S., "A Survey of Code-based Change Impact Analysis Techniques," *Software Testing, Verification and*

Reliability, Wiley Online Library, vol. 23, no. 8, pp. 613-646, 2013.

[18] Li L., Zhang L., Lu L., and Fan Z., "Assessing Object-oriented Software Systems based on Change Impact Simulation," *in Proceeding of IEEE 10th International Conference on Computer and Information Technology (CIT)*, pp. 1364-1369, 2010.

[19] Mens K., Lopes C., Tekinerdogan B., and Kiczales G., "Aspect-oriented Programming: Workshop Report," *in Proceeding of European Conference on Object Oriented Programming Workshops*, Finland, pp. 483-496, 1997.

[20] Mens T. and Demeyer S., "Introduction and Roadmap : History and Challenges of Software Evolution," *Springer Berlin Heidelberg*, 2008.

[21] Mens T. and Wermelinger M., "Formal foundations of Software Evolution : Workshop Report," *ACM SIGSOFT Software Engineering Notes*, vol. 26, no. 4, pp. 27-29, 2001.

[22] Misra S. and Akman K., "Weighted Class Complexity: A Measure of Complexity for Object Oriented System," *Journal of Information Science and Engineering*, vol. 24, no. 6, pp. 1689-1708, 2008.

[23] Munoz F., Baudry B., Delamare R., and Le Y., "Usage and Testability of AOP: An Empirical Study of AspectJ," *Information Software Technology*, vol. 55, no. 2, pp. 252-266, 2013.

[24] Murphy G., Walker R., Baniassad L., Robillard M., Lai A., and Kersten M., "Does Aspect-oriented Programming Work? Determining the Best Method for Evaluating the Effectiveness of a New Technology," *Communications of the ACM*, vol. 44, no. 10, pp. 75-77, 2001.

[25] Piveta E., Moreira A., Pimenta M., Arajo J., Guerreiro P., and Price R. , "An Empirical Study of Aspect-oriented Metrics," *Science of Computer Programming*, vol. 78, no. 1, pp. 117-144, 2012.

[26] Shao J. and Wang Y., "A New Measure of Software Complexity based on Cognitive Weights," *Canadian Journal of Electrical and Computer Engineering*, vol. 28, no. 2, pp. 69-74, 2003.

[27] Sharafat A. and Tahvildari L., "Change Prediction in Object Oriented Software Systems: A Probabilistic Approach," *Journal of Software (JSW)*, vol. 3, no. 5, pp. 26-39, 2008.

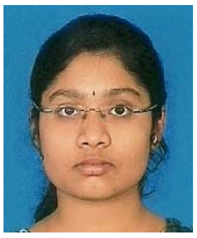
[28] Shen H., Zhang S., and Zhao J., "An Empirical Study of Maintainability in Aspect-oriented System Evolution using Coupling Metrics," *in Proceeding of 2nd IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE)*, China, pp. 233-236, 2008.

[29] Yokomori R., Siy H., Yoshida N., Noro M., and Inoue K., "Measuring the Effects of Aspect-oriented Refactoring on Component Relationships: Two Case Studies," *in Proceeding of 10th International Conference on Aspect Oriented Software Development*, Brazil ,pp. 215-226, 2011.

[30] Zhang S., Gu Z., Lin Y., and Zhao, J., "Change Impact Analysis for AspectJ Programs," *in Proceeding of 24th IEEE International Conference on Software Maintenance (ICSM)*, China, pp. 87-96, 2008.

Senthil Suganantham received his M.S. in Computer Science from Wichita State University, USA and B.E. in Electronics and Communication Engineering from MK University, India. He is currently working as Associate Professor in SSN College of Engineering, India, and pursuing PhD in Anna University, India. He has published 10 papers in International Journals and Conferences. His research areas include AOSD and empirical software engineering.

Chitra Babu Professor and Head, Department of Computer Science and Engineering, SSN College of Engineering, India, received her Ph.D. from the Indian Institute of Technology, Madras (IITM), India, M.E. in Computer Science and Engineering from Anna University, India, M.S. in Computer and Information Systems from the Ohio State University, USA, B.E. from PSG College of Technology, India. She has published over 50 research publications in refereed International Journals and Conferences. Her research interests include object and aspect oriented software engineering, service oriented architecture and cloud computing.



Madhumitha Raju received her B.E. in Computer Science and Engineering from SSN College of Engineering, India. She worked as a Trainee Software Engineer by Ford Technology Services, India specializing in Mainframes. She is currently a software engineer in Ford Technology Services, India.