Change Impact Analysis Model-Based Framework for Service Provisioning in a Grid Environment

EKABUA, OBETEN OBI, BSc. (Hons.), MSc.

A Thesis Submitted in Fulfilment of the Requirements for the award of the Degree of Doctor of Philosophy (PhD) in Computer Science

Department of Computer Science
Faculty of Science and Agriculture
University of Zululand
South Africa

February, 2009.

DECLARATION

I declare that this research study on **Change Impact Analysis Model-Based Framework for Service Provisioning in a Grid Environment** is my work, and has never been presented for the award of any degree in any University. All the information used has been duly acknowledged both in text and in the references.

Signature_____ Date_____

Ekabua, Obeten Obi

Approval

Signature_____ Date_____

Supervisor: **Professor M. O. Adigun**
Department of Computer Science
University of Zululand
South Africa

DEDICATION

To

Fanny, Honour and Vine

ACKNOWLEDGEMENT

First and foremost, I want to express my heartfelt gratitude to the Most High God, with whom I have come to find out that there is nothing impossible for Him to do. He took leadership of the initial plan and has brought it to completion. I will forever praise Him for these wonderful acts and give Him all the Glory.

I remain grateful to Prof. Matthew Adigun, my supervisor, for advice, support, useful criticism, enjoyable discussions, and an amazing ability to find solutions to even the most complex problems. I would like to say thank you for allowing God to use you in fulfilling this life time enviable vision and dream in my academic pursuit.

I am also grateful to my previous supervisors Dr. Sue Black, Prof. Dilip Patel, Prof. Shushma Patel for the initial help and useful supervision when I first started this PhD programme in London South Bank University, United Kingdom, before transferring to University of Zululand, South Africa. I remain indebted to you for the help and support I received during my stay in the UK.

I am also grateful to Dr S. S. Xulu, for the support and encouragement in pursuit of this degree. I would like to thank Doctoral forum and all the members of the Centre of Excellence for Mobile e-Services for Development, Department of Computer Science, the University of Zululand, for their kind support and words of encouragement and creating a pleasant research atmosphere throughout my studies.

I remain thankful to all the members of Centre for Systems and Software Engineering, London South Bank University – Peter Rosner, David, Louis Spring, Gary Bell and Haidar Bilal, my PhD colleague, for collaboration, discussions and feedback.

I will like to thank Chris Kuba, Chris Ente and Emeka Godwin Idagu for their individual support and encouragement during this period of my studies.

I want to thank the University of Calabar, Calabar, Nigeria, for affording me study leave to enable me undertakes a Doctoral degree.

I am thankful to Elder Etobe for your words of encouragement while in Zululand University. Ijeoma Mba, Esien, Nejo and Kayode, my good Christian friends.

I also want to say thank you to Torti Ekabua, Omini Ekabua and Lydia Ekabua (now late), John Ekabua, Caleb Ekabua, Emmanuel Ekabua for your great support and fruitful encouragement leading to this achievement. God bless you all. My mum, you have always been there for me in spite of all. Thank you.

I would also love to say a big thank you to members of Power of Faith Bible Church – Rev. Dr & Rev. Dr. Mrs Umoren, Pastor Dr & Pastor Mrs Usang Bassey, Pastor and Mrs Odey, Pastor and Mrs Ajom, Pastor and Mrs Inwem, Bro Joe James and the rest of the PFBC family.

I devote this thesis to my family. Without your support, I would not be where I am today, or who I am today. Thank you! Fanny, Honour and Vine for your support, compassion, and understanding of my ups and downs, all of which have made this thesis a reality. I would not have desired a better family than you are to me.

LIST OF FIGURES

LIST OF TABLES

Table of Contents

Chapter 4

Chapter 5

Chapter 6

ABSTRACT

Grid-based Utility Infrastructure for Small, Medium and Micro Enterprise (SMME) Enabling Technology (GUISET) is an architecture in which distributed applications must interact across platforms via the concept of services. The building block in a GUISET application are business process services which are expected to undergo maintenance just like any other software components. The basic operation of software evolution is change, thus making change inevitable in software development. Throughout the entire lifecycle of a software system, from conception to retirement, things happen that require the system to be changed.

Changes are required to fix faults, improve or update products and services. Under these conditions, it is crucial to have full control over and knowledge about what changes mean to the system and its environment. Otherwise, changes might lead to deterioration and a multitude of other problems. The effectiveness and efficiency with which a company can predict or control these changes could have a significant impact on its competitiveness. In a complex product, where the constituent parts and systems are closely dependent, changes to one item of a system are highly likely to result in a change to another item, which in turn can propagate further. The activity of assessing the implications of realizing a change is termed change impact analysis. One glaring issue is to anticipate changes and structure the service in such a way that changes will be discovered early to avoid change propagation. This is because, it is likely that services will be exposed to many unanticipated changes during its lifetime, and the support for unanticipated changes creates a gap for an important research goal. Two issues are involved here: (1) the assessment of the consequences of altering (or not) the functionality of the service, and (2) the identification of the service dependencies that are affected by the change.

Traditionally, research on change impact analysis has mainly focused on technical aspects such as traceability analysis and software code change propagation. By contribution and extension of existing knowledge, the research presented in this thesis

is exploratory in its nature, conducted through overarching emphasis of monitoring change propagation during service provisioning, with the understanding that software is now being consumed as services. The main goal is to advance the current state of change impact analysis by evolving a model-based framework for validating, analyzing and monitoring change propagation in our typical grid service provisioning environment (GUISET). The framework emanating from this endeavour consists of two associated formal models of change Impact Analysis Factor Adaptation mechanism, and a Fault and Failure Assumption Model for service provisioning in our GUISET grid environment.

As a part of the empirical validation of the framework, we graphically represent the relationship between change and impact, indicating that, as the number of changes increased, the impact also increased. This is because the number of changes was affected by the number of dependent services. Therefore, if the dependencies were high, the number of changes will be high, and consequently the impact due to fault propagation will be high.

CHAPTER ONE

INTRODUCTION AND BACKGROUND

1.1 Introduction

Software change occurs for several reasons, for example, in order to fix faults, add new features or restructure the software to accommodate future changes [1]. Changing requirement is one of the most significant motivations for software change. Requirements change from the point in time when they are elicited until the system has been rendered obsolete. Changes to requirements reflect how the system must change so as to be useful for its users and remain competitive in the market. At the same time, such changes pose a great risk as they may cause software deterioration. Thus, changes to requirements must be captured, managed and controlled carefully to ensure the survival of the system from a technical point of view. Factors that can inflict changes to requirements during both initial development as well as in software evolution are, according to Leffingwell and Widrig [2]:

(i) The problem that the system is supposed to solve changes, for example for economic, political or technological reasons.

(ii) The users could change their minds about what they want the system to do, as they understand their needs better. This could happen because the users initially were uncertain about what they wanted, or because new users entered the picture.

(iii) The environment in which the system resides changes. For example, increase in speed of computers can affect the performance of the system.

(iv) The new system is developed and released leading users to discover new requirements.

(v) One of the interconnected hardware components is faulty or its warranty is expired and unable to provide the services expected.

The last two factors are real and common. When the new system is released, users realize that, they want additional features, that they need data presented in other ways, that there are emerging needs to integrate the system with other systems, and so on. Thus, new requirements are generated by the use of the system itself. According to the "laws of software evolution" [3], a system must be continually adaptable, or it will be progressively less satisfactory in its environment.

Problems arise if requirements and changes to requirements are not managed properly by the development organization [2]. For example, failure to ask the right questions from the right people at the right time during requirement development will most likely lead to a great number of requirements change during subsequent phases. Futhermore, failure to create a practical change management process may mean that changes cannot be timely handled, or that changes are implemented without proper control.

Maciaszek [4] points out: "Change is not a kick in the teeth, but unmanaged change is." In other words, a software development business requires a proper change management process in order to mitigate the risks of constantly changing requirements and their impact on the system.

Impact analysis (IA) is an abstract concept, denoting the activity of analyzing how something affects something else. Without a proper context, impact analysis can refer to anything. Searching for the term using an Internet search engine uncovers a number of variants, such as: business impact analysis, environmental impact analysis, poverty and social impact analysis, and financial impact analysis. The epithet *system change* or *service change* narrows the field to what is discussed in this thesis, namely the analysis of the effects of a proposed change to the artefacts of a service component, and thereby to its development (e.g., time and cost) and use. Perhaps the most popular definition of impact analysis in this particular context is that of Bohner and Arnold [5]: "*. . . identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change*." This definition explicitly differentiates between *consequences* and *modifications* and satisfies the focus of this research. A typical consequence is increased project lead time and cost. However, a change may also affect the functional content (i.e., the feature set) of the system, which can have both positive and negative consequences on, for example, sales, customers' attitudes, and competitive position. A change must be analyzed both from the perspective of the development organization, and from the perspective of external actors such as customers, the market, and competitors [6].

Now knowing what impact analysis is, our concern centres on how we can manage change propagation in our Grid-based Utility Infrastructure for Small, Medium and Micro Enterprise (SMMEs)-enabled Technology (GUISET) research environment. We will like to provide the answers to this concern in two ways. First, by active use of any non-trivial service provisioning software system stimulates the flow of changes from a

number of different change sources. Added functionality to the GUISET user interface will in most cases require changes to the current system. Identified defects, anticipated environmental changes (i.e., changes to the environment in which the system runs, for example the underlying operating system), and improvement requests should be taken care of (requiring *corrective*, *adaptive*, and *perfective* maintenance, respectively [7]). Users evolve their knowledge about the problem solved by the system, and expect the system to evolve with them [6, 8]. In any large-scale software development effort where services are provided, for example GUISET, all of these change sources are active, thus creating a massive flow of changes. Because of change conflicts, lack of service resources, bad timing, service unavailability, etc., changes should not and cannot be unconditionally implemented in a system. Impact analysis provides a means for determining the characteristics and effects of a proposed change, which form a basis for its conditional approval or disapproval. Second, by a modification to the source code of the GUISET user interface following a proposed change is seldom isolated. Other parts of the interface or service other than the one modified must probably be updated, otherwise it may cause defects to be accidentally introduced. Defects must of course be avoided, but modifications necessitated by other modifications are inevitable, in particular in interfaces with high coupling and low cohesion. Impact analysis can be used to get a better idea of the scope and complexity of the modification, and thereby the risk associated with it. For example, if impact analysis determines that a modification will lead to other modifications throughout the *entire* service or interface, the proposed change may be put off with the argument that it is not worth the effort required to implement it, unless the expected gain is very high [6].

1.2 Background Information

It is widely recognized that change is a necessary and an inevitable property of any software, but software change can, and will, if not controlled, lead to software deterioration. For example, when Mozilla's 2,000,000 Source Lines of Code (SLoC) were analyzed, there were strong indications that the software had deteriorated significantly due to uncontrolled change, making the software very hard to maintain [9]. Software deterioration occurs in many cases because changes to software seldom have the small impact they are believed to have [10]. In 1983, some of the world's most expensive programming errors each involved the change of a single digit in a previously correct program [11], indicating that a seemingly trivial change may have immense impact. As we build more and more software systems, we see situations where systems fail. Because software systems are growing in complexity as the need and requirement for their service is becoming more useful, we naturally want to reuse the functionality of existing systems rather than build from the scratch. A real dependency is a state of affairs in which one system depends on the functionality provided by another. The problem is that we also create artificial dependencies along with real dependencies [12]. Impact analysis is the activity involving identifying what needs to be modified in order to make a successful change, or to determine the consequences on the system if the change is implemented. Therefore, changing services can provide us with information regarding the quality of services provided. Change Impact Analysis (CIA) determines what impact change to services will have on the rest of the system. It can be used during service maintenance to keep the system at a high level of quality, avoiding degradation of the system or during development to ensure that the quality of the system is

maintained throughout the development process. The CIA tailored for this research is based on the effect that a change to a particular service or service provider will have on the rest of a service. It determines the scope of the change and provides a measure of the service's complexity [13].

A Service Oriented Architecture (SOA) provides different services ranging from middleware to support service management, data communication, e-commerce and e-marketing. A SOA is an architectural style whose goal is to achieve loose coupling among interacting software agents. A service is a unit of work done by a service provider to achieve desired end results for a service consumer. Both the consumer and the provider are roles played by software agents on behalf of their owners. The results of a service are usually a change of states for the consumer but can also be a change of state for the provider or for both [14]. SOA's have also been regarded as services exposed using the Web Service Protocol suites that include transport protocol (such as HTTP, SMTP and FTP), XML messaging (such as XML-RPC, SOAP, and REST), Service Description (typically using Web Service description Language-WSDL, Web Service Endpoint Language – WSEL, Web Service Metadata Exchange, and Web Service Dynamic Discovery – WS-Discovery), and Service Discovery (a common registry of services such as the UDDI API, Electronic Business XML-ebXML, Directory Services Markup Language - DSML) [15]. Service related software applications play an important role in our lives. Products that affect people's lives must have quality attributes. Therefore, good quality software is required and in order to determine the quality of software we need methods to measure it - metrics. A key point here is that the quality of a service product may change over time and web service related applications are no exception. In the early days of computing, software

costs represented a small percentage of the overall cost of a computer-based system. Hence, a sizable error in estimates of software cost had relatively little impact. Today, software is the most expensive element in many computer-based systems. Therefore steps taken to reduce the cost of software can make the difference between the profit and loss of a company. So by determining the quality attributes of software, more precise, predictable and repeatable control over the software development process and product will be achieved [16].

Software is supposed to change. So, why does the software community struggle with the problems of software maintenance and the software's requisite change? Much of the concern has more to do with the complexity and sheer size of current applications than it has to do with change. As we develop large software systems (now in the 10s of millions of lines of code) incorporating more features and newer technology, the need for new CIA technology has emerged [17]. Changing requirements are endemic to software [18] and many researchers have written about software changes and their consequences [19]. Final requirements seldom exist for software systems since they are continually being augmented to accommodate changes in user expectations, operational environment, and the like. Therefore, many software systems are never really complete until their function in the organization becomes obsolete.

Development cycles for large software systems can be lengthy. Continuously changing requirements represent a key management challenge for these efforts. Similarly, software change cycles for large, complex systems can be slow. Without the requisite CIA and management mechanisms, software changes during maintenance can

have unpredictable consequences that often delay their implementation [20]. CIA is integral to software release planning and the software maintenance process [21].

1.3 Statement of the Problem and Research Questions

Impact analysis is one of the most tedious and difficult parts of software change. Manual impact analysis is labor intensive and error prone. Systematic approaches to impact analysis are frequently not part of formal engineering training [21]. It is performed only when absolutely necessary due to the cost involved. Therefore, it effectively limits the quality, consistency, and number of changes that can be made to a software system. The tools used in most impact analysis processes are primitive and low level, and need a substantial human interaction to accomplish the task. Automated impact-analysis tools often provide a rather limited analysis. Software change processes do not adequately address impact analysis [22, 23, 24].

Business applications encode various business processes within an organization. They assist users in performing their daily work more efficiently and effectively. Business processes are a set of logically related tasks that are performed to achieve business objectives.

Due to marketing strategies, organizations tend to reengineer their business processes in order to improve performance indices, such as cost and quality of services. When a business process is changed (e.g. adding a task, and removing a task), the source code that implements the business application needs to be updated in order to support the new business requirements. However, determining the impact of a business process change is not trivial but is challenging since business analysts are not experts in

the source code and the spread of the changes across services may be too complex. Business process changes usually result in code changes which may cause unexpected side-effects to other business processes. It is important for business analysts to evaluate the impact of business process changes especially when considering various business alternatives. For example, a simple change to a business process such as the addition of a welcome screen may seem trivial but may require a substantial amount of code changes. The code changes may be too costly in particular when the business value of such a change is considered [23, 25].

To remain competitive, managers frequently modify their processes. Determining the cost of modifying a business process is a dependent function on maintenance. CIA is central to maintenance and cost considerations resulting from change inconsistencies [25]. This research intended to develop a model-based framework approach that would give business analysts a rough estimate of the impact and cost of changing a business process within a service oriented environment without having to study the code.

The motivation for this work was to improve the maintainability of service provisioning in a frequently changing requirement environment. Therefore, this research addressed the challenges of change impact analysis for service provisioning in a grid environment. The research considered the effect of change propagation determination by developing a general framework for change propagation.

The main research questions (MRQs) answered by this research were as follows:

MRQ1: Can we develop a change propagation framework that can be used to control change impact specifically in a grid environment?

This research question stemmed from the consideration of our GUISET research focus with a desire for the research to lead to process improvements. This perspective is also very rewarding as it enhances the development of a change propagation framework as well as two associated formal models as methods and tools that would work in practice given the constraints induced by processes, culture, and ways of working at the maturity of our ongoing GUISET research. Also, providing answers to this question gave rise to one journal and one conference paper showing the associated models that worked alongside the framework. Details of this are discuss in chapter 4 of this thesis.

MRQ2: In the face of service management's need to improve productivity, enhance re-use and reduce capital investment during service provisioning, can we formulate formal models for CIA around this framework?.

These research questions were formulated after (1) noting that most impact analysis research focuses on technical aspects and code design, and (2) observing that impact analysis was seen as a very narrow activity and is hardly expressed in formal mathematical models. We were interested in investigating further how software practitioners use or make use of impact analysis as part of the process of managing change. This perspective is rewarding as it provides insight into how the research community can develop impact analysis formal measures for a particular research environment as in our case where GUISET research is maturing. Our approach to this question was first to investigate existing formal measures or metrics as used in object oriented software and propose them with suggestions for its use in the new paradigm of aspect oriented software. This is because service platforms which were previously

designed in object oriented environment will soon be taking a new design look with the introduction of the new paradigm of aspect oriented programming. Therefore, providing answers to this question led to the conference publication of Software Engineering Research and Practice (SERP'07) at the World Computer Congress (WORLDCOMP'07).

1.4 Research Rationale

Service maintenance has been recognized as the most costly phase in the software life cycle [26]. Over the life of a software system, service maintenance effort has been estimated to be frequently more than 50% of the total life cycle cost [27]. By identifying potential impacts before making a change, the risks associated with embarking on a costly change can be reduced, because the cost of unexpected problems generally increases with the lateness of their discovery. The more a particular change causes other changes, the higher the cost is. If a proposed change has the possibility of impacting large, disjointed sections of a service, the change will need to be re-examined to determine whether a safer change is possible [21]. Because there is no known existing method of CIA for service provisioning, this research becomes imperative.

Maintenance is costly and difficult. It is not always clear where modifications will have to be made to a service, or what the impact of any type of change to a service may have across the whole service. CIA in this research is expected to show the maintainer what the effect of any change will be on the rest of the system. The strategy employed by this research not only permits evaluation of the consequences of planned changes, but also allows trade-offs between suggested service change approaches to be considered.

1.5 Research Goal and Objectives

The main goal of this research was to evolve a model-based framework for managing (validating, analyzing and monitoring) change propagation in a typical grid service provisioning environment. In achieving the goal for this research, the following research objectives (ROs) were useful:

RO1: Analyzing and reviewing current Object Oriented Metrics and proposing an extension as a metric suite for the relatively new paradigm of Aspect Oriented Software, as there are no existing and agreeable known metrics suites for Aspect Oriented software. Since accurate scientific measurement will not be effective outside metrics, this review becomes the first essential objective of this research. Design of service oriented software is done within the object oriented environment and the new paradigm of Aspect Oriented Programming.

RO2: Designing a generic change propagation framework for monitoring and validating service changes during service provisioning and enhancing service provisioning in a typical grid environment.

RO3: Formulating two formal models: (1) A Fault and Failure assumption model and, (2) A Change Impact Analysis Adaptation Model - as stability metrics to the framework and thereby enhancing service change prediction in a grid environment.

1.6 Research Methodology

The primary approach in this research was framework design. In order to achieve this, we had to employ literature survey arguments and model formulation as secondary methods. These are elaborated further in the subsections below.

1.6.1 Argument based on Literature Survey

This method involved the review and analysis of related literature. Although various works of literature about CIA are available, they formed their basis around software code design, but no known work is done with respect to service provisioning. The research argument, therefore, is that, if software is delivered and consumed as services, then the analysis of related literature would help build a contributive approach towards applying this CIA technique into service provisioning.

1.6.2 Formulative Method

The knowledge acquired from the literature survey serves as a foundational approach to the formulative method which involved framework design and formal models formulation**.**

1.6.3 Framework Design

Existing frameworks on software change propagation served as a guide into building a generic framework for this research. We combined Knowledge gained from literature on service oriented architecture and grid to build a generic framework to support change automation in a grid service provisioning environment.

1.6.4 Model Formulation

The knowledge gained during literature survey about metrics and formal mathematical approaches to evaluate the effect of changes in software enhanced the formulation of the two models for this research. The first model only predicted which services would be impacted if a change were really made. Therefore the impact is syntactic (impact dependent on the static nature of the provisioning system) and the change hypothetical. The first model is therefore able to provide us with an Impact Analysis Factor (IAF) as a metric for such a hypothetical change.

The second model is called a *fault and failure assumption model*. The fault component of the model was to analyze the error type that a faulty service may induce in a grid environment, while the failure component of the model incorporates common failures which are due to unavailable infrastructure, client crash, service failure, server crash, session failure and component failure to enhance change prediction. The assumption of the second model is justified by the first model.

1.7 Contribution to knowledge

Business applications encode various business processes within an organization. Business processes are a set of logically related tasks that are performed to achieve business objectives. Our Business objective is service provisioning. When a business process is changed, the source code that implements the business application needs to be updated in order to support the new business requirements. However, determining the impact of a business process change is challenging since business analysts are not

experts in the source code and the spread of the changes across services may be too complex. Business process changes usually result in code changes which may cause unexpected side-effects to other business processes [23, 25]. Service changeability is one of the essential properties of software and involves all software technologies. It is the core of software service evolution. One of the proposed solutions is to anticipate changes and structure the service in such a way that changes will be discovered early to avoid change propagation. This is because, it is likely that services will be exposed to many unanticipated changes during its lifetime, and the support for unanticipated changes creates a gap for an important research goal. Formal models, frameworks or tools for change propagation would be required to achieve this research goal [26, 27, 28, 29]. Therefore, our main contributions of this research to change impact analysis, change management in particular, GUISET research and software engineering as a discipline are:

(i) The introduction of a framework whose implementation can help service providers and business analysts, to assess and predict service provisioning entities requiring a change and where a particular change will likely propagate, and determine the cost implications before maintenance schedule. This contribution would support our GUISET technology research focus and any SOA-specific service provisioning environment, as there is no known existing framework targeted at monitoring change propagation during service provisioning but rather existing frameworks are dedicated to only software code design change propagation monitoring.

(ii) Because frameworks alone would not give service providers adequate information, the need for formal models to support the framework which can be

adopted in any SOA-specific CIA environment became necessary. This has led to another contribution of two formal models intended for quality assurance and to help our GUISET research.

1.8 Included and Related Publications

This thesis builds on a number of research studies which have previously been reported in conference and journal papers. This section provides a brief description of the manuscripts that have been converted into thesis chapters or part of a chapter. The following manuscripts appear in the thesis (Ekabua, Obeten and Adigun M. O. are the main authors):

(1) Part of Chapter 5 is based on a paper entitled "Reviewing Object Oriented Metrics for Aspect Oriented Paradigm". Presented and published in the *Proceedings of International Conference on Software Engineering Research and Practice* (SERP'07), WorldComp'07, Las Vegas, Nevada, July, 2007. The paper reviews and examines existing works on metrics as used in Object Oriented software and by analyses and cognate reasons, proposes an extension of some existing object oriented metrics for use in the new aspect oriented paradigm.

(2) Part of Chapter 4 is based on a paper entitled "A Generic Change Propagation Framework to Enhance Service Provisioning in a Grid Environment." The paper is published in *Asian Journal of Information Technology* (AJIT), 6(10): 1015-1019, 2007. ISSN: 1682-3915. The paper presents a framework as a support instrument or tool for enhancing

change propagation control process in a grid environment. The typical grid environment where this framework is targeted is our GUISET environment but because its design is generic implying that it can also be adopted in any SOA-specific CIA environment.

(3) Part of Chapter 4 and 5 is based on a paper entitled "A Framework and Associated Models for Determining Change Impact Analysis during Utility Service Provisioning in a Grid Environment." The paper is presented and published in *Proceedings of International Conference on Software Engineering Research and Practice* (SERP'08), WorldComp'08, Las Vegas, Nevada, USA. July 14 – 17, 2008. The paper describes the framework published in (2) above and presents two associated models – Change Impact Analysis factor Adaptation Model and Fault and Failure Assumption Model- as formal models to enable the framework provide adequate quality assurance in managing change propagation.

(4) The paper entitled "Experienced Report on Assessing and Evaluating Change Impact Analysis through a Framework and Associated Models." Published in *Journal of Information Science and Engineering* (JISE), Special Issue on *Enterprise Computing Systems and Applications*, Vol. 25, No. 2, 2009. This paper attempted the use of regression based model alongside the models of the framework and compared the results obtain with Bayesian statistics. It concludes that the Bayesian method is a useful

technique for service maintainability prediction by achieving significantly better prediction accuracy as compared to the regression method. Another interesting direction of this paper is using the Bayesian model with a different service structure (whose relationship is static but not causal), for example, a tree augmented Naïve-Bayes' classifier to predict service provisioning effort. The data were generated manually in line with the manual strategy for impact analysis determination.

The following manuscript does not appear as chapters in the thesis, but assists to demonstrate the validity of the concepts presented in this research and are currently under review for publication:

(5) The paper entitled "On Using Change Impact Analysis Model-Based Framework Implementation to Predict the Effect of a Change of Service in a Grid Environment." Submitted for Publication in South Africa Computer Journal (SACJ). This paper reports on the implementation of the framework and models through the use of bayesian statistics to predict the effect of a change of a service in a grid environment. Its application was mainly on manual methods as described in chapter two for impact analysis determination. Data generated through the models and Bayesian statistics indicates the viable practicality of the use of Bayesian statistics satisfying accurate prediction criterion for predicting the effect of a change of service in a grid environment.

1.9 Thesis Summary

The remaining part of this thesis is organized as follows:

Chapter 2 gives a historical overview of change impact analysis and its overview in relation to the main subject matter as seen by this research. The key terminologies used in this research are also explained.

Chapter 3 describes service change configuration as a related process to change impact analysis. It also outlines the various processes involved in change configuration and describes how change configuration in service provisioning environment can be managed.

Chapter 4 gives background information on the process of change propagation while expressing the need for a framework to support change propagation. The Chapter presents the change propagation framework for this research and introduces issues in our ongoing GUISET research, the gaps and challenges of the GUISET research from which this research is derived. The major interest of this chapter which is in line with the objective and research question is the introduction of a change propagation framework to enhance service provisioning in our GUISET research area.

Chapter 5 explains the second major part of this research by presenting two formal models as associated metrics to support the change propagation framework crafted to support GUISET research. The chapter also presents existing metrics used for impact analysis to differentiate it from what this research introduces. We pointed out clearly that existing metrics focus on software code and our proposal is at the service level. The argument of this research leading to this proposal is that, gradually software capabilities will be delivered and consumed as services. Agreeably, they may be

implemented as tightly coupled systems, but the point of usage to the portal, to the device, to another endpoint, and so on, will use a service-based interface. Already, we have progressed from modules, to objects, to components, and now to services.

Chapter 6 is the concluding chapter of this work. Firstly, a summary is presented followed by the concluding remarks and finally research limitations and further work.

CHAPTER 2

CHANGE IMPACT ANALYSIS OVERVIEW

2.1 Introduction

This chapter deals with the historical context of impact analysis and gives an over-view of our research focus in Change Impact Analysis with respect to existing related work. It then progresses to define the basic concepts for change impact analysis as used in this research. Furthermore, the chapter introduces maintenance and measurement concepts, as the emphasis for this software engineering investigation.

2.2 Historical Context

Impact analysis history is long dated, not in using that particular term, but resolving the problem of accurately determining the effect of a proposed change. Therefore, the need for software practitioners to determine what to change in order to implement requirement changes has always been present. As an example, Haney's paper of 1972 on a technique for module connection analysis is often referred to as the first paper on impact analysis [28]. The technique holds the idea that for every pair of modules in a system, the probability exists that a change in one module in the pair necessitates a change in the other module. The technique was then used to model change propagation between any system components including requirements. In 1979, Weiser [29] introduced program slicing, which is a technique for focusing on a particular problem by retrieving executable slices containing only the code that a specific variable depends on. Requirements traceability was defined in ANSI/IEEE Standard 830-1984 in 1984 [30].

Traceability describes how Software Life-cycle Objects are related to each other and can be used to determine how change in one type of artefact causes change in another type of artefact. Yau and Collofello in 1980 [30, 31] introduced the notion of ripple effect. Their models made it possible to determine how change in one area of source code propagates and causes change in other areas.

Impact analysis relies on techniques and strategies that date back a long time. It is, however, possible to identify a trend in impact analysis research over the years. Early impact analysis work focused on source code analysis, including program slicing and ripple effects for code. As software engineering matures among software organizations, the need to understand how changes affect other Software Life-cycle Objects (SLOs) rather than just the source code became necessary.

For example, Turver and Munro [32] pointed out that source code is not the only product that has to be changed in order to develop a new release of the software product. In a document-driven development approach, many documents are also affected by new and changed requirements. The user manual is an example of a document that has to be updated when new user functionalities have been provided. Turver and Munro (ibid) focus on the problem of ripple effects in documentation using a thematic slicing technique. They noted that this kind of analysis has not been widely discussed before. The same approach could be applied to the requirements document itself in order to determine how a new or changed requirement impacts the requirements specification.

In 1996, Arnold and Bohner published a collection of research articles called Software Change Impact Analysis [33]. The purpose of the collection was to present the current,

somewhat scattered, materials that were available on impact analysis at the time. Reading the collection today, nearly ten years later, it becomes apparent that it is still very relevant. Related publications after 1996 seem to work with the same ideas and techniques. This research does not intend to depreciate the work that has been done, but it indicates that the field is not in a state of flux. Therefore, the focus remains on adapting existing techniques and strategies to new concepts and in new contexts. Impact analysis for utility service provisioning environment is a case in point.

At the approach of year 2000, the Y2K problem made it obvious that there is the need for an extensive impact analysis to identify software and parts of software that had to be changed to survive the century shift. This served as a revelation for many organizations, in which the software process previously had not included explicit impact analysis [34].

In an evolving technological revolution, software systems are much more complex than they were 25 years ago, and it has become very difficult to grasp the combined implications of the requirements and their relationships to architecture, design, source code and platforms. Thus, a need for impact analysis strategies that employ requirements and their relationships to other SLOs is necessary. Still, dependency webs for large software systems can be so complex that it is necessary to visualize them in novel ways. Bohner and Gracanin [35] present a research that combines impact analysis and 3D visualization in order to display dependency information in a richer format than is possible with 2D visualization. Bohner also stresses the need to extend impact analysis to middleware, COTS software and web applications. The use of these types of software is becoming more common, moving the complexity away from internal data and control

dependencies to interoperability dependencies. Current impact analysis strategies are not very well suited for these types of dependencies [34].

Software change is an essential activity for software evolution. This change involves a process that either introduces new requirements into an existing system, modifies the system if the requirements were not correctly implemented, or moves the system into a new operation environment. In actual fact, for a number of reasons, change is an inescapable property of any software. Nevertheless, software changes can, and will, if not properly controlled, lead to software degradation. As an example, during the analysis of Mozilla's 2,000,000 Source Lines of Code (SLOC), there were strong indications that the software had degraded significantly as a result of uncontrolled change, thus making the software difficult to maintain [9].

With the maturity of the software industry, resources have shifted from being devoted to developing new software systems to making modifications to evolving software systems: software maintenance. A major problem for developers in a changing environment is that small changes can ripple through software to cause major unintended impacts elsewhere. This is why software developers require a mechanism to understand how a change to a software system will impact the rest of the system. This process is called CIA. Making software changes without understanding their effects can lead to unreliable software products. CIA can then be used to reduce the amount of maintenance required, thus increasing the reliability of the software, since fewer errors would have been introduced. [36].

Software degradation occurs in many cases because changes to software seldom have the small impact they are believed to have [37]. In 1983, some of the world's most

expensive programming errors each involved the change of a single digit in a previously correct program [38], indicating that a seemingly trivial change may have immense impact. A study in the late 90s showed that software practitioners conducting impact analysis and estimating change in an industrial project underestimated the amount of change by a factor of 3 [39]. More so, as software systems become more and more complex, the problems associated with software change increased accordingly. For example, when the source code across several versions of a 100,000,000 SLOC, fifteen-year-old telecom software system was analyzed, it was noticed that the system had decayed due to frequent change. The programmers estimating the change effort drew the conclusion that the code was harder to change than it should be [40].

Impact analysis is a tool for controlling change, and thus for avoiding degradation. Bohner and Arnold define impact analysis as "*the activity of identifying the potential consequences, including side effects and ripple effects, of a change, or estimating what needs to be modified to accomplish a change before it has been made*" [17]. Consequently, the output from impact analysis can be used as a basis for estimating the cost associated with a change. The cost of the change can be used to decide whether or not to implement it depending on its cost/benefit ratio.

Impact analysis is an important part of requirements engineering since changes to software often are initiated by changes to the requirements. In requirements engineering textbooks, impact analysis is recognized as an essential activity in change management, but details of how to perform it are often left out, or limited to reasoning about the impact of the change on the requirements specification (see, for example, [6,

41, 42, 43, 44, 45]. An exception is [37], where Wiegers provides checklists to be used by a knowledgeable developer to assess the impact of a change proposal.

Irrespective of its natural place in requirements engineering, impact analysis research is more commonly found in literature related to software maintenance. In our experience, impact analysis is an integral part of every phase in software development. During requirements development, the design and the code are not yet in existence, so new and changing requirements affect only the existing requirements. Also during design, code does not yet exist, so new and changing requirements affect only existing requirements and design. Finally, during implementation, new and changing requirements affect existing requirements as well as design and code. This is captured in Figure 2.1. Note that in less idealistic development processes, the situation still holds i.e. requirement change affects all existing system representations.

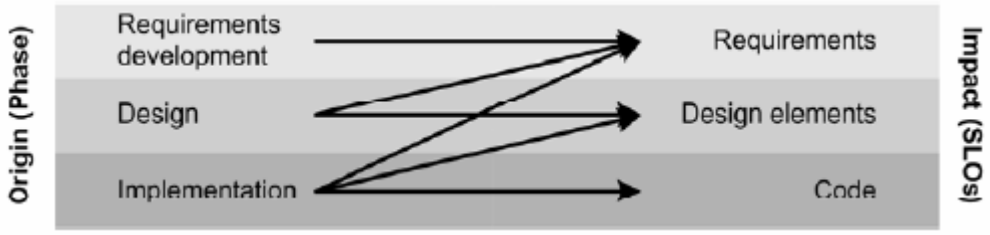

Figure 2. 1: Software life-cycle objects (SLOs) affected (right) due to requirements changes in different phases (left) [6**].**

IA has been practiced in various forms for years, yet there is no consensus definition [46]. There are different definitions of change impact analysis. Pfleeger and Bohner [58] define *change impact analysis* as "the evaluation of the many risks associated with the change, including estimates of the effects on resources, effort, and schedule." Turver and Munro [32] define *change impact analysis* as "the assessment of a change, to the

source code of a module, on the other modules of the system. It determines the scope of a change and provides a measure of its complexity." Arnold and Bohner [46] define *change impact analysis* as identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change. They emphasize the estimation of the impacts. The Pfleeger and Bohner's [47] definition extends to the evaluation of impacts. The ripple effect of a change to the source code of a software system is defined as the consequential effects on other parts of the system resulting from that change. These effects can be classified into a number of categories such as logical effects, performance effects or understanding effects.

2.3 Impact Analysis Overview

An *impact* (noun) is the effect or impression of one thing on another. Impact can be thought of as the consequences of a change. *Impact analysis* (IA) is used to determine the scope of change requests as a basis for accurate resource planning and scheduling, and to confirm the cost/benefit justification. *Service change impact analysis* estimates what will be impacted in services and related service functionality if a proposed service change is made. It is defined as the process of *assessing* the effects on other components of the services resulting from the proposed change [17]. It is evaluated by determining the scope of the change and the complexity of the change. The words *determine, estimates, assessing* and *evaluating* represent the quantitative and qualitative effects of that change on other items and are the major concerns of this research. For the purpose of this thesis and our GUISET research, we will be using the definitions above to place emphasis on our context of the use of the word "impact

analysis" and to signify different levels of the thoroughness of the analysis. The central issues involved in this research which we are over-viewing are ceiled on change process/propagation, maintenance and measurement. But to enhance understanding, we introduce key terminologies as used in the whole of this thesis.

2.3.1 Key Terminologies

Throughout this thesis, we use several terms and concepts that are relevant to our definition of Impact, Impact Analysis and Service Change Impact Analysis. In this section, we outline the working definitions of these concepts.

a) **Determine: -** refers simply "*to establish the identity*." This means the least analysis effort and information obtained through the analysis.

b) **Estimate: -** together with its synonyms "evaluate" and "assess" all mean "*to judge something with respect to its worth or significance*," though estimate implies a judgment that is made prior to or instead of an actual measurement.

c) **Assess** means to critically estimate or evaluate for the sake of gaining an understanding or obtaining an action plan.

d) **Evaluate: -** means to determine relative or actual non-monetary worth.

e) **Service -** is a unit of work done by a service provider to achieve desired end results for a service consumer. Both consumer and provider are roles that are played by software agents on behalf of their owners. The results of a service are usually a change of states for the consumer but can also be a change of state for the provider or for both.

f) **Service Oriented Architecture** – is a technical software infrastructure that enables a collection of services to be exposed using the Web Service Protocol suites which include transport protocol (such as HTTP, SMTP and FTP), XML

messaging (such as XML-RPC, SOAP, and REST), Service Description (typically using Web Service description Language-WSDL, Web Service Endpoint Language – WSEL, Web Service Metadata Exchange, and Web Service Dynamic Discovery – WS-Discovery), and Service Discovery (a common registry of services such as the UDDI API, Electronic Business XML-ebXML, Directory Services Markup Language - DSML)

g) **Software life-cycle objects** (SLOs – also called software products, or working products) are central to impact analysis. A SLO is an artefact produced during a project, such as an architectural component, connected to each other through a web of relationships. Relationships can both be between SLOs of the same type, and SLOs of different types. For example, service A can be interconnected to another service B to signify that they are related to each other.

h) **Dependency analysis** determines a detailed relationship among service entities, for example service A depends on the functionality provided by service B to achieve its aim.

i) **Traceability analysis**, on the other hand, is the analysis of relationships that have been identified during development among all types of SLOs. Traceability analysis is thus suitable for analyzing relationships amongst architectural components.

j) The **Service Set** represents the set of all dependencies (example architectural components) in the services – all the other sets are subsets of this set.

K) The **Starting Impact Set** (SIS) represents the set of services that are initially thought to be changed. The SIS typically serves as input to impact analysis approaches that are used for finding the Estimated Impact Set.

l) The **Estimated Impact Set** (EIS) always includes the SIS and can, therefore, be seen as an expansion of the SIS. The expansion is derived from the application of change propagation rules to the internal service model repeatedly until all services that may be affected are discovered. Ideally, the SIS and EIS should be the same, meaning that the impact is restricted to what was initially thought to be changed.

m) The **Actual Impact Set** (AIS), contains those dependencies that have been affected once the change has been implemented. In the best-case scenario, the AIS and EIS are the same, meaning that the impact estimation was perfect.

n) **Change propagation** is one of the key parts of system maintenance and evolution. In order to explain change propagation, we have to understand that a system consists of entities, functionalities and their dependencies. The dependency between entities A and B means that entity B provides certain services, which A requires for its correct functioning. Different services depend on other services for the provision of some functionalities requirement. The dependency is consistent if requirements of A are satisfied by what B provides. But changes to service introduce inconsistencies and in order to reintroduce the consistency into the service or system, a change propagation process keeps track of the inconsistencies and the locations where the secondary changes are to be made.

It is common to distinguish between **primary** and **secondary change**.

o) **Primary change** also referred to as **direct impact**, corresponds to the SLOs that are identified by analyzing how the effects of a proposed change affect the system. This analysis is typically difficult to automate because it is mainly based on human expertise.

p) **Secondary change** also referred to as **indirect impact**, may take two forms:

(1) **Side effects** are unintended behaviours resulting from the modifications needed to implement the change. Side effects affect both the stability and function of the service and must be avoided.

(2) **Ripple effects**, on the other hand, are effects on some parts of the system caused by making changes to other parts. Ripple effects cannot be avoided, since they are the consequence of the service's structure and implementation. They must, however, be identified and accounted for when the change is implemented.

q) **Service architecture** of a service is its basic structure, consisting of interconnected components or services.

2.3.2 Change Process

To put change impact analysis in perspective, we first need to understand the process of change. Leffingwell and Widrig discuss five necessary parts of a process for managing change [2]. These parts, depicted in Figure 2.2, form a framework for a change management process allowing the project team to manage changes in a controlled way.

Figure 2. 2: Change management process framework [2].

Plan for change involves recognizing the fact that changes occur, and that they are a necessary part of the system's development. This preparation is essential for changes to be received and handled effectively.

Baseline requirements means to create a snapshot of the current set of requirements. The point of this step is to allow subsequent changes in the requirements to be compared with a stable, known set of requirements.

A *single channel* is necessary to ensure that no change is implemented in the system before it has been scrutinized by a person, or several persons, who keep the System, the project and the budget in mind. In larger organizations, the single channel is often a change control board (CCB).

A *change control system* allows the CCB (or equivalent) to gather, track and assess the impact of changes. According to Leffingwell and Widrig [2], a change must be assessed in terms of impact on cost and functionality, impact on external stake-holders (for example, customers) and the potential to destabilize the system. If the latter is overlooked, the system (as pointed out earlier) is likely to deteriorate.

To *manage hierarchically* defeats a perhaps too common line of action. A change is introduced in the code by an ambitious programmer, who forgets, or overlooks, the potential effect the change has on test cases, design, architecture, requirements and so on. Changes should be introduced top-down, starting with the requirements. If the

requirements are decomposed and linked to other SLOs, it is possible to propagate the change in a controlled way.

This framework for change process leaves open the determination of an actual change process. Requirements engineering textbooks propose change management processes with varying levels of detail and explicitness [43, 44, 45]. The process proposed by Kotonya and Sommerville is, however, detailed and consists of the following steps [41]:

1. Problem analysis and change specification
2. Change analysis and costing, which in turn consists of:
    a. Check change request validity
    b. Find directly affected requirements
    c. Find dependent requirements
    d. Propose requirements changes
    e. Assess costs of change
    f. Assess cost acceptability
3. Change implementation

Impact analysis is performed in steps 2b, 2c and 2e, by identifying requirements and system components affected by the proposed change. The analysis should be expressed in terms of required effort, time, money and available resources. Kotonya and Sommerville (ibid) suggest the use of traceability tables to identify and manage dependencies among requirements and between requirements and design elements.

One key problem in accommodating changes in an environment is to know all the factors that impact a given change, and the consequences of this change.

2.3.3 Software Maintenance

Software evolution refers to the on-going enhancements of existing software systems, involving both development and maintenance. Software maintenance has been recognized as the most costly and difficult phase in the software life cycle [49, 50]. Over the life of a software system, the software maintenance effort has been estimated to be frequently more than 50% of the total life cycle cost. This maintenance cost shows no sign of declining [48].

Unlike many other types of products, software products are intended to be adaptable. Although software neither deteriorates nor changes with age if its media are well-presented. Software maintenance is an expensive process where an existing program is modified for a variety of reasons, including correcting errors, adapting to different data or processing environments, enhancing to add functionality, and altering to improve efficiency [51].

For programs with many interacting modules, modifying and then revalidating a program is complex. In other words, analysis, testing, and debugging may be required for each individual module and for the interactions among modules. The problem is further compounded because the maintainers are rarely the authors of the code and usually lack a complete understanding of the program. Worse still, maintainers often do not have access to specifications or design documents but just the code. As software

ages and evolves, the task of maintaining it becomes more complex and more expensive. Some of the other causes of software maintenance problems are that:

(1) Software maintainability is often not a major consideration during design and implementation.

(2) Maintenance has been largely ignored in software engineering (SE) research.

(3) Maintenance activities are not well understood.

Decades of research on maintenance activities in procedural software have produced several conclusions. Among them is the fact that a reduction in maintenance cost could be achieved by a more controlled design process, and by more rigorous testing of potential problem areas early in the life cycle [49, 52].

Software maintenance can be classified into four categories: *perfective*, *adaptive, corrective and preventive* [53, 54].

i. **Perfective maintenance** is performed to alter functionality, eliminate processing inefficiencies and enhance performance or improve maintainability.
ii. **Adaptive Maintenance** is performed to adapt software systems to changes in its environment
iii. ***Corrective maintenance*** is performed in response to software failures or to correct errors.
iv. **Preventive maintenance** is performed to update software systems in anticipation to failure problems.

Moreton [55] defines the steps of the maintenance process as: change management, impact analysis, system release planning, change design, implementation, testing and

system release/integration. These steps, which occur sequentially are supported by a further activity that continues concurrently – progress monitoring.

For maintenance work to be effective, it is vital to control the input to the process – the procedure by which change requests are notified and managed in the first place. The change management and impact analysis are the first two steps in the maintenance process. The software maintenance process can only be optimized if precise and unambiguous information is available about the potential *ripple effects* of a change on an existing system.

2.3.4 Software Measurement

Software measurement as a software engineering discipline has been around now for some thirty year [80]. Its purpose is to provide data or information which can be used either for assessment or prediction during the lifecycle. Typically it is used as assessment either during the initial development of software or during maintenance at a later date. Software maintenance is now a very complex subject. The core concepts in software measurements are based on the science of measurements. Software measurement has evolved into a key software engineering discipline and practice. In the past, many software organizations treated measurement as an additional, non-value-added task, or just "another thing to do." Measurement is now considered to be a basic software engineering practice, as evidenced by its inclusion in the Level 2 maturity requirements of the Software Engineering Institute's CMMi products and related commercial software process standards [66].

Why measure software? To begin, software has become a major factor in corporate investment and business strategies, even for "non-software- intensive" organizations. It is a key component in an organization's ability to maintain pace with rapidly changing information technology in an increasingly competitive environment. Given the large corporate investment in developing and maintaining critical information assets, there is a growing demand for more objective assessment and management of software-intensive projects. Measurement begins at the project level. Software measurement helps the project manager to do a better job. It helps to define and implement more realistic plans, to properly allocate scarce resources to put those plans into place, and to accurately monitor progress and performance against those plans. Software measurement provides the information required to make key project decisions and to take appropriate action. Measurement helps to relate and integrate the information derived from other project and technical management disciplines. In effect, it allows the software project manager to make decisions using objective information [77].

There are many aspects of software that we may wish to measure: source code length, time taken to write source code, or the price at which the executable source code is eventually sold. In each case, we are measuring a specific attribute of a particular entity: length and price are attributes of the entity source code, while time is an attribute of the process which produced it. Thus length and price are product attributes, while development time is a process attribute. But, this distinction is not always clear cut: someone buying software as a resource will regard its price as a resource attribute. Any entities that we may measure can be subdivided into [81]:

Processes – any software related activities e.g. writing source code.

Products – any artifacts which arise from processes e.g. source code.

Resources – any resource which is input to a process e.g. personnel.

2.3.4.1 Typical Software Product Measures

In considering measuring software or the service provided by the software, it is necessary to look at it from several viewpoints. Probably the intuitive measure is the **length** e.g. lines of code (LOC) which measures the physical size of the code. We can also measure **functionally** in terms of what the user actually gets from the code, which is the service. The next measure speaks of the **complexity** which is a function of the number of dependencies to its functionality. We discuss these three typical measures below.

(i) Length Measurement

The most frequently used measure is number of lines of code for measuring source code program length. Although simple, this creates room for ambiguity. It must always be made apparent which lines are included, i.e. whether to count comment lines, blank lines and data declarations along with program statements. It also needs to be made clear whether a line containing several separate instructions is counted as one line or one line per instruction. The most widely accepted definition used for measuring LOC is NCLOC (Non-Commented Line of Code) which was defined by Hewlett Packard [84, 85]. The authors further emphasis that, NCLOC includes all lines within a program except blank lines and comment lines. The length of designs and specifications can also be

measured by counting atomic objects such as processes and date stores in a data-flow diagram or type declarations and predicated in a Z specification.

(ii) Functionality Measurement

Functionality is often measured to provide effort and duration estimates from the specification or design of a system. Albrecht's function points [86] are probably the best known measure of functionality used. Computation of function points for a particular system involves counting atomic objects such as inputs, outputs and files, and then using a technical complexity factor to produce a final function point count. There are several problems with function point analysis as detailed in [83]. Other functionality measures include some cost drivers within COCOMO [84, 87]. For the purpose of this research, our functionality measure is mainly for change of service status reporting and auditing. Our approach is by the use of a formal model of fault and failure.

(iii) Complexity Measurement

Complexity as a measure of program size exists as the following [83]:

- **Computational complexity** the complexity of the underlying fault and the associated dependencies.
- **Algorithmic complexity** the complexity of the algorithm which has been implemented.
- **Structural complexity** the complexity resulting from dependencies to the main service.
- **Cognitive complexity** the effort required to understand the service's interconnectedness.

Ripple effect and logical stability which are techniques for determining CIA are measures of structural complexity.

Developing a metric for complexity has been a scientific pursuit for decades. Solutions have been found and published by academicians but some practitioners are not enamoured of them. They chose to deal with complexity just through the simple notion of size and remain content with the inadequate mapping it provides. Much work in this direction only provides answers to software codes, no mention is made to developing a complexity metric for the services provided by this software and their various dependencies.

One can perceive complexity in so many ways and pick a metric that will satisfy specific needs. Some complexity metrics are now supported with tools, frameworks and models, to enable ease of application [77, 84].

Early measurement of software complexity focused entirely on source code with the simplest complexity measure being LOC. In 1983 Basil and Hutchens [89] suggested that LOC be used as a baseline or benchmark to which all other complexity metrics is compared i.e. an effective metric should perform better than LOC so LOC should be used as a 'null hypothesis' for empirical evaluation. Much empirical work has shown it to correlate with other metrics [90], most notably McCabe's cyclomatic complexity which is discussed in more detail in section 4.3.5.5. The earliest code metric based on a coherent model of software complexity was Halstead's software science [78]. Early empirical evaluations produced high correlations between predicted and actual results but later work showed a lower correlation. Hassan and Holt [91] found only modest

correlation, with software science being outperformed by LOC. According to Gold and Mohan [90] the most important legacy of software science is that it attempts to provide a coherent and explicit model of program complexity as a framework within which to measure and make interpretations. Software science is also important because, since it deals with tokens, it is fairly language independent.

After a concentration on code level measurement for some years, focus widened to include measurement during the earlier stages of the software development lifecycle. Design level metric can in theory be obtained much earlier in the development of a project thus providing information which can be used for more informed resource management.

Structural complexity of software can be broken down into the following types:

i. **Data flow structure** shows the way that data flows through a program and its behaviour as it interacts with the program.
ii. **Control flow structure** shows the order in which program instructions are executed taking into account whether there are any loops or branches.
iii. **Data structure** involves the organization of the data itself, which is independent of the program. Data structure will not be discussed further in this thesis as emphasis will be on related measurement for this research work.

The change impact analysis measure is fundamentally concerned with the control flow structure of the implemented system, as this forms a key issue in our model design for change propagation determination.

2.3.4.2 Internal and External Attributes

The attributes of entities can be broken down into two categories: **internal and external** [BF90]. While internal attributes can be measured purely in terms of the entity itself, external attributes can only be measured with respect to how the entity relates to its environment. For example, **size** of code is an internal attribute whilst **reliability** of code is an external [81]. Within ISO 9126 [82] it is proposed that the quality of a software product may be evaluated by the following attributes: functionality, reliability, usability, efficiency, maintainability and portability. It is not immediately apparent that functionally is the only **internal** attribute, although this is in fact the case as all the others are dependent on the environment.

2.3.4.3 Direct and Indirect Measures

Classification of measures of an attribute is either **direct** or **indirect.** Direct measures are those for which the measurement does not depend upon any other attribute. Examples includes: length of source code and duration of testing process. Whereas indirect measurement of an attribute is measurement involving the measurement of other attributes. An example of indirect measurement is the measurement of some artefacts that are dependent on the main service. We refer to this artefact as functional dependencies [83, 84].

2.6 Chapter Summary

Impact analysis is an important part of requirements engineering since changes to software often are initiated by changes to the requirements. As the development process becomes less and less waterfall-like, and more of new and changed requirements can be expected throughout the development process, impact analysis becomes an integral part of every phase in software development. In some sense, impact analysis has been performed for a very long time, albeit not necessarily using that term and not necessarily fully resolving the problem of accurately determining the effect of a proposed change. The need for software practitioners to determine what to change in order to implement requirement changes has always been present. Classical methods and strategies to conduct impact analysis are dependency analysis, traceability analysis and slicing. Early impact analysis work focused on applying such methods and strategies onto source code in order to conduct program slicing and determine ripple effects for code changes. The maturation of software engineering among software organizations has, however, led to a need to understand how change requests affect other SLOs than source code, including requirements and the service provisioning environment, and the same methods and strategies have been applied. Current methods and strategies are based on analyzing traceability or dependency information, utilizing slicing techniques, consulting design specifications and other documentation, and interviewing knowledgeable developers. Interviewing knowledgeable developers is

probably the most common way to acquire information about likely effects of new or changed requirements.

In this chapter, we have presented the historical context overview of change impact analysis. We have also defined key concepts as used in this research to explain our research direction for change impact analysis and provide discussions that highlights the major concern of impact analysis beginning from change process to maintenance and measurement.

CHAPTER 3

SERVICE CHANGE MANAGEMENT FOR GUISET

3.1 Introduction

In this chapter, we introduce the concept of service change management as an approach for our Grid-based Utility Infrastructure for Small, Medium and Micro Enterprise (SMMEs)-enabled Technology (GUISET) research and as it relates to change impact analysis. A Grid network is described as the collaboration of different dispersed organizations intending to share and coordinate physically distributed resource virtualized as a single resource to the users. Because of the capabilities that Grid acquires, Adigun et al. [59] proposed the Grid-Based Utility Infrastructure for Small Medium and Micro Enterprise (SMMEs)-enabled Technology (GUISET). GUISET is an architecture targeted at providing physically distributed IT services such as application software, storage and Central Processing Unit (CPU) as utilities. This is envisioned as an approach that can provide an affordable access to Grid Services deployed in the GUISET grid environment. SMMEs lacking such utilities are targeted to benefit from this service provisioning environment, as they have access to the IT services available on demand without necessarily owning the infrastructure.

The chapter begins with background information which will be used as a frame of reference on a wider context. The chapter discusses managing change in services, software development and change evolution and the various processes involved in service change management for GUISET as this is the grid environment for this research.

3.2. GUISET: Our Foundational Reference Technology

3.2.1 Grid and Agent

Grids consist of a collection of distributed (networked) computers pooling their resources together in a coordinated manner to enable users to utilize processing, storage, software and data resources from any of the interconnected computers, leading to greater resource sharing and higher utilization ratio. The grids may be viewed as virtual organizations (VO). The core unifying concept that underlies Grids and Agents systems is that of a service. A service is an entity that provides a capability to a client through a well defined message exchange. It is also refers to a vehicle by which consumer's need is satisfied according to a negotiated contract, which includes service level agreement and the function offered [60, 64]. In 3rd Generation Grids, all entities are services since service interractions are achieved through web service mechanisms. Although, every agent (an autonomous problem solving entitiy with boundaries and interfaces) is considered a service, not all grid services are necessarily agents. Therefore, the autonomous action notion is a function of how agents and grids interoperate.

The main objective of Grids is that of resource sharing and coordinated (simple interaction protocol) problem solving in dynamic, multi-institutional virtual organizations [61]. A Grid therefore provides an infrastructure for federated resource sharing across trusted domains. Grid primary concern has been the mechanism by which communities form and operate. Thus, the grid effort is devoted to how community standards are represented via explicit policy and enforcement and how actions and commitments by community members are specified, monitored and enforced through implementation [13].

Agents are autonomous identifiable problem solving entities with well-defined boundaries and interfaces. They are situated in a particular environment, designed to fulfil a specific role and capable of exhibiting a flexible problem solving behaviour in pursuit of their design objectives. They need to be both reactive (reacting to changes in their environment on time) and proactive (taking initiatives) [62]. Agent and Grid systems consist of dynamic and stateful services. Since it is possible for new services to be created and destroyed over the system lifetime, the underlying service model for agents and grids is dynamic [63].

Grid-based Utility Infrastructure for SMME-enabling Technologies (GUISET) is the architectural technology upon which this research is based. The concept of GUISET hinges on the idea of a technology that makes SMME affordable. What is therefore required is to find an appropriate strategy to make affordable technologies available using the utility approach to service delivery. The GUISET idea stems from the following research findings:

i. The possibility of making Mobile Computing on fixed infrastructure through using approaches such as hardware-level abstraction (Internet Suspend/Respond) or OS-level abstraction (process migration) [64].
ii. The arrival of Software Architectural Support for Handheld Computing into the main stream, which would enhance the composition of large, distributed, decentralised mobile systems [65, 66]
iii. The availability of a Reference Architectures that would enhance sharing domain-specific applications [67, 68].

What is non-existent is an infrastructure for deploying centrally owned resources such as software services for a subscription. The birth of GUISET was motivated by this idea.

3.2.2 Foundation Reference Model.

Figure 3.1 is the GUISET architecture which is divided into three layers: (1) Multi-Modal Interfaces (2) Middleware Layer (3) Grid Infrastructure layer.

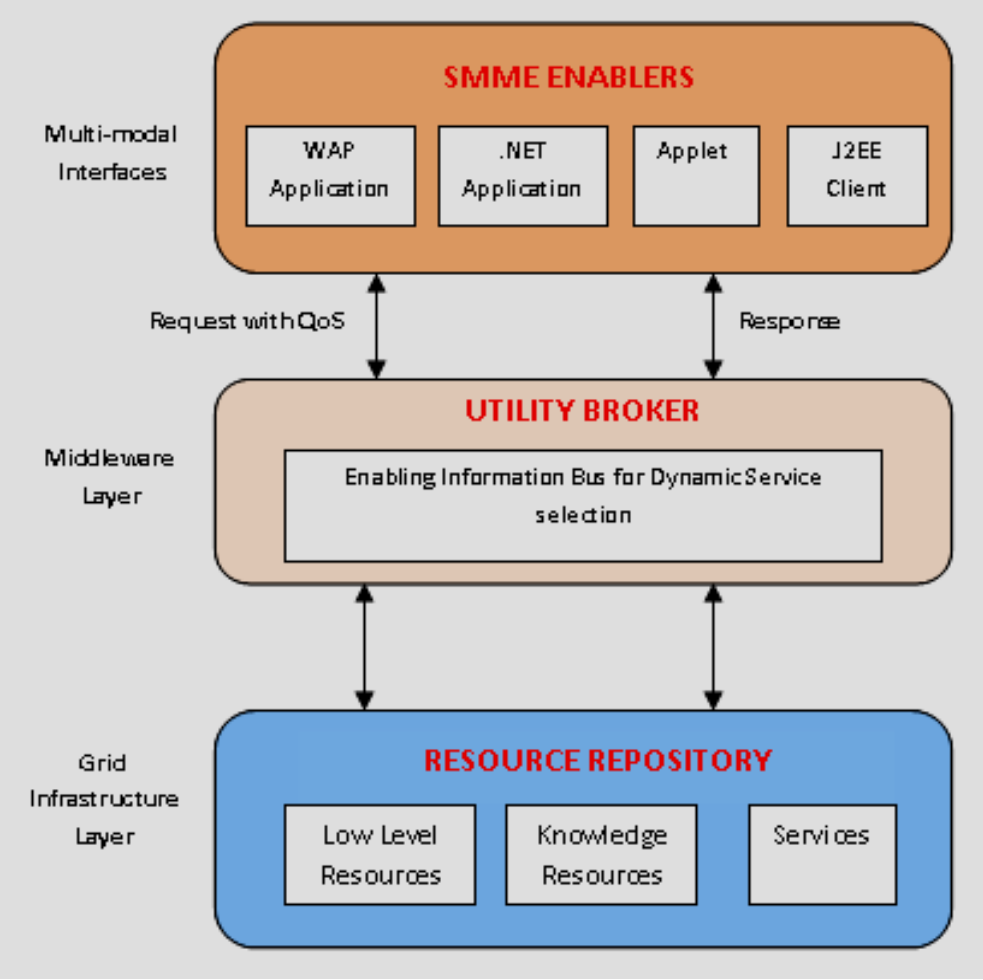

Figure 3.1: GUISET Architecture

This research focuses on the Middleware layer where secondary services are resident to manage the interaction with the primary services at the Grid Infrastructure Layer. On

the basis of classical distributed event-based Architecture [64], it is possible to deploy applications as published services offered to subscribers. Producers and consumers as components in a typical architecture will be communicating via a network of brokers, exchanging notifications on network lines. For simplicity of the architectural framework, we crafted our own reference architecture as a set of archetypes (Fig. 3.2) namely Technology, Client, and Services (three types of which are distinct: Information, Transaction and Third-party). The Technology archetype refers to the set of producers in the framework. Client is the archetype that refers to devices on which the services are deployed. Finally, a Service archetype is a consumer of event notifications emanating from the Technology archetype.

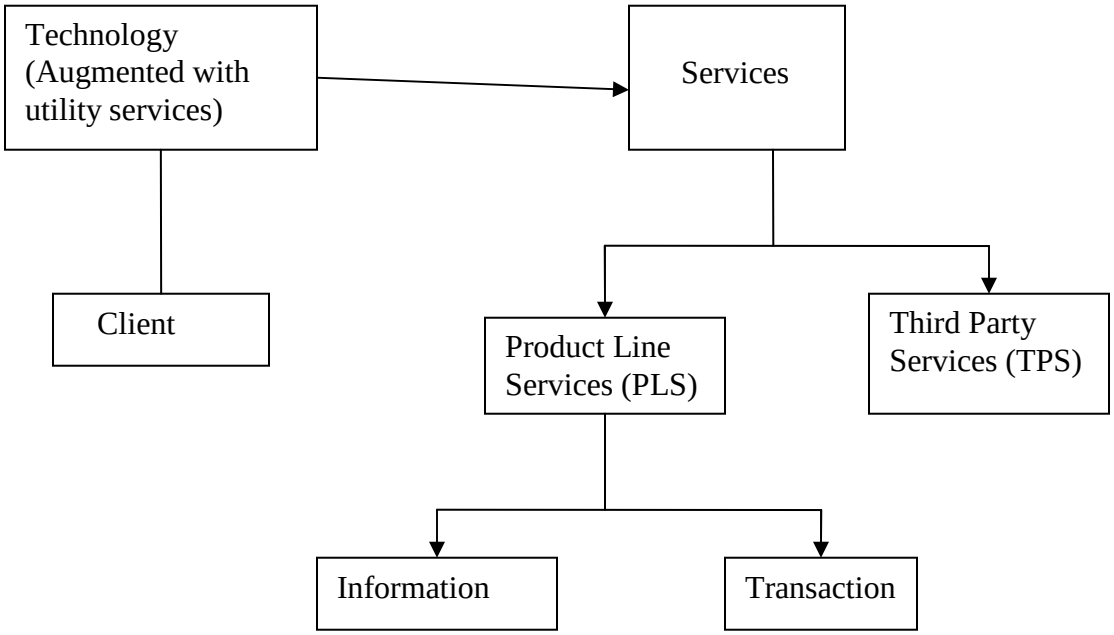

Figure 3.2: Reference Architecture for the mobile commerce Product Line

These afford us the opportunity to address the following research challenges:

3.2.3 GUISET Research Challenges

GUISET as our grid-based research focus has a number of challenges, but for the purpose of this research, we address only two of these challenges:

(1). Service *Brokerage and Context-aware Operating Environment*

We assume here that clients will own contexts either for themselves (device context) or on behalf of others as user contexts. The 'management of context awareness' role is then allocated to the Technology archetype. The Technology archetype performs, amongst other roles, the provisioning of necessary support for billing, quality of service, fault tolerance and some other basic middleware functionalities. Including this research, various research challenges are being addressed. An example is on how to handle operations that need to be carried out when the network is down? [69]. The fault and failure assumption model for this research plays a vital role to support quality of service and fault tolerance as the best communication style for event notification and handling operations that needs to be carried out when the network is down.

(2). *Service Provisioning*

The reference model's main goal is to achieve a loosely coupled event-driven architecture (interacting with one another dynamically) which is easily adaptable and facilitates scalable implementation of networked services. Services of the utility type deal with context event notifications, they are dispatched by a network of brokers

(Fig.3.3) that convey and filter notifications.

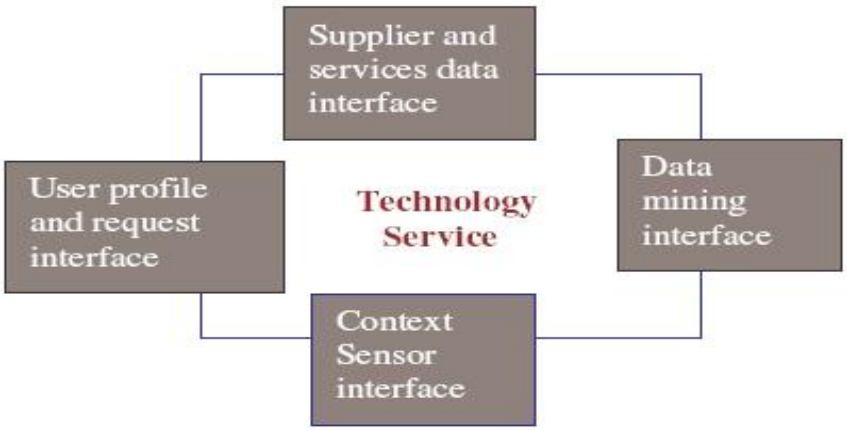

Figure 3.3: Technology Archetype as a Network of Brokers of the Four Utility Services [64]

As services are composed dynamically, performance problems might arise that were not anticipated and this will make it difficult to guarantee the quality of service during service provisioning. The change propagation framework and the associated models proposed by this research enhance quality service provisioning by validating, analyzing and monitoring service change propagation in the GUISET grid service provisioning environment. A number of investigative activities are ongoing. There are two other types of services, first is the product-line (PL) type and second is the third-party (TP) or non product-line type of service

3.2.4 Gaps Left by the Reference Model

However, while the reference model enables us to understand what is required to put together a family of applications meant for supporting SMMEs, it is inadequate with

respect to enabling the deployment of such applications. First, the brokerage and context aware operating environment still leaves us wondering how service components will be coordinated. What sort of protocol will be used? How will the service components interface and how will quality of service be guaranteed?

The service composition model supported under the reference model is not detailed enough and therefore still leaves some questions without answers. While we know that a Service Oriented Architecture provides the general framework for creating product family members, however we are yet to come up with definite answers to: What sorts of applications will the members be? What interfaces will they expose to users? Can composite application be composed from other applications? How will sharing of applications be supported? What sort of billing approach will be well-suited to the need of the envisaged infrastructure? [64].

This research contributes by considering how service component will be coordinated, interfaced and how quality of service will be guaranteed. The framework for this research enhances the monitoring of the service component to avoid costly breakdown of workflow due to maintenance work for change in application. Once more, changes to component interaction and implementation model is enhanced for quality of service provisioning

3.2.5 Service Models

The utilization of services in the real world brought about a classification based on the nature of the application logic they provide, as well as their business-related roles within

the overall solution. We refer to these classifications as service models. Thus, this research employs two types of service models [70]:

(i) **Business service model.** This characterizes the most fundamental building block within an SOA. Although it is completely autonomous, yet not executed in isolation because business services are frequently expected to participate in service compositions. They are used within an SOA as follows:

   a. As fundamental building blocks for the representation of business logic.
   b. to represent a corporate entity or information set
   c. to represent business process logic
   d. as service composition members

(ii) **Utility service model.** This stands for any generic service or service agent designed for potential reuse. The key to achieving this classification is that the reusable functionality should be completely generic and non-application specific in nature. They are used within SOA as follows:

   a. As services that enable the characteristics of reuse within SOA
   b. As solution agnostic intermediary services
   c. As services that promote the intrinsic interoperability characteristic of SOA
   d. As the services with the highest degree of autonomy

Utility services are mostly associated with the application service layer, and are often referred to as utility application services.

3.3 Background Information

In discussing this research work, it will be necessary to position it with respect to a wider context, or frame of reference. This will make it easier to delimit the research, and to relate it to other relevant areas of coverage. The context for CIA is *change control for measurement and maintenance*, which is part of *service change management for our GUISET service provisioning environment*.

We justify this research connection with GUISET for two reasons:

(1) It adds to existing knowledge about performing impact analysis in some specific business concepts so as to improve business value and maximize profit and output per time.

(2) Information obtained from performing impact analysis can be used for planning changes, making changes, accommodating certain types of service changes, and tracing through the effects of changes. It makes the potential effects of changes visible before the changes are implemented to make it easier to perform changes more accurately during service provisioning and maintenance.

3.4 Contextual Definitions

To better understand our GUISET context for which this research discusses service change management in lieu of software change management, it is worthwhile to briefly comment on some terminologies as used along the context of this research.

i. ***Proposed changes*** referring to a change that is merely a suggestion, and is not necessarily carefully thought through or even realistic. A proposed change can stem from, for example, needs for improvement, a failure need resulting from a fault and a need for adaptation. This is included to help develop the fault and failure assumption model discussed in chapter 5.

ii. **Service *change request*** is used to refer to the artefact used in the change control process to formally describe a request for changing the service. Note that defects typically are handled separately from change requests and proposals, through problem reports or trouble reports, even though fixing a defect usually involves making changes to the service. Because service change affect code changes, it is necessary to allow a change to be adequately thought of considering the cost benefit analysis before any such request will be considered.

iii. ***Service fault*** is used to signify a breakdown of a service due to unavailable infrastructure, client crash, service failure, server crash, session failure and component failure or the expiry of a waranty

iv. ***Activity Checker*** (AC) is responsible for the specification of the constraints that a well-formed service design should satisfy in order to check whether the application's design is in conformance to the main host design. Violation of the rules governing the activity checker will trigger a constraint violation event from the *Constraint Activator* (CA) to be returned to the *Change Propagation Mechanism* (CPM). This informs the *Service Repairer* (SR) of a triggered event calling for a way of fixing the violated constraint by performing actions, which

change the application's design and keeps record of the propagated change (ripple effect).

v. **Mechanism *Validator*** (V) is responsible for checking the consistency of the change to the design (through the AC), which can result in further actions.

Details of the above explanations are incorporated into Fig.4.1 of chapter 4.

When *change* is used by itself, it refers to a change in general, such as a proposed one, a requested one, one necessary to fix a defect, or one required to implement a new requirement. In other words, our use of the word change in GUISET and in this research connotes a very general concept.

3.5 Change Management

Change management (CM) as used in this research refers to the collective name for those activities that are associated with the change control function of CM (though *change management process* and *change control process* are used interchangeably throughout the thesis to denote the process of managing change). The change management activities that this research considers for GUISET is as follows:

- ✓ Change identification,
- ✓ Change analysis,
- ✓ Change Cost Evaluation
- ✓ Change approval or disapproval, and
- ✓ Change implementation and verification.

These research findings recommend that for effective change management, the first thing is to identify the need for change and justify it. Change can be syntactic and our

models concentrate on changes that have a syntactic impact, therefore, appropriate measures are based on impact that are dependent on the static nature of the provisioning system. This implies that impacts have a likelihood of propagation .

The next step is change analysis where the change is analyzed with respect to impact on system functionality, non-functional qualities (e.g., performance, reliability, and maintainability), user interfaces, etc., but also cost and schedule. This we referred to as service change impact analysis. The step following analysis is cost evaluation for full justification of the next step. Cost evaluation will provide certain implications during the cost analysis. Management will then need to consider the cost implication to either approve or disapprove which is the next step in the change process. When the impact of the change is known, it is up to the selected authority to decide if the change should be accepted or rejected, by weighing how desirable the change is against how much it would cost to realize the change. The cyclic nature of the change management process is visible earlier as well; if the change request is not written properly, it will be returned to the author for elaboration and correction. It follows that the lead time of the change control process increases if a change request has to go back to a previous step, so high change request quality and careful analysis must be strived for. If a change is approved, it is passed on for implementation and verification.

Change Implementation is the last step and it involves modifying documentation and service item, but also performing unit testing and verifying the correctness of the change. This is called service checking which is performed by the Activity Checker (AC) of our change propagation framework. That means once the service change request is complete it is passed on for implementation.

3.6 Managing Configuration in Services

Service change management (SCM) is a critical element of software engineering. Unfortunately, in practice it is often ignored until absolutely necessary. It may be introduced at first customer release, possibly through customer pressure. Frameworks for SCM has never been considered except frameworks for software change management where only certain aspects of software development and maintenance are accommodated. Software Change Management methods and tools are often viewed as intrusive by developers, a management tool that imposes additional work with little perceived benefit to the tasks of the developer. SCM is addressing issues on how to control change evolution in services for our GUISET research. A standard definition taken from IEEE standard 830-1998 highlights the following operational aspects of Change Management [71]: Identification, Control, Status Accounting and Audit and Review. Therefore for the purpose of our GUISET research, we define Service Change Management as:

A technique for identifying likely faulty service components or functionality to enhance change control and status reporting for appropriate service auditing.

The above definition therefore covers four key areas (earlier mentioned in section 3.5) which we shall be discussing:

3.6.1 Service Change Identification

The identification of a service requiring change is a fundamental part of Service Change Management. The purpose of this activity is to identify and characterize

components or *dependencies or functionality items*, and to determine how they relate to the service environment and its dependencies. A component or functionality is a single entity that is subjected to change during the project and therefore needs to be under control. An artefact outside of change management is not subject to the formal protection provided by the change management process. When an artefact is changed, there is a risk that changes to the artefact may go unnoticed. But artefacts that are not very critical for the successful change management process of the project might be excluded.

3.6.2 Change Control

Component control is synonymous with Change control because what is changed is the component. Change control consists of activities that allow management of changes by service change request analysis, service change identification, service change approval/disapproval, and service change implementation. The change management activities are governed by formal control processes and routines to enable changes to components to be handled in a consistent and repeatable manner. The change control function is tightly associated with service *change requests*, as a service change request initiates the execution of the change management activities. Since changes are so frequent in service provisioning environments, change control becomes the CM function that is most frequently conducted.

Change control is not only about controlling service change requests, but also about controlling the implemented changes. It should be possible to trace a change back to the component item it targeted, and for any component, it should be possible to

identify the change relative to the previous version. This requires that *traceability* practices are in place and are used, so that links among components and other functional artefacts can be created and maintained.

The mechanism *Validator* (V) is responsible for checking the consistency of the change to the design through the Activity Checker(AC), which can result in further actions.

3.6.3 Service Status Reporting

The Change Propagation Mechanism (CPM) is responsible for informing the *Service Repairer* (SR) of a triggered event calling for a way of fixing the violated constraint by performing actions, which change the application's design and keeps record of the propagated change (ripple effect).

The purpose of status accounting, or status reporting, is to make available data and reports from the change management process. The reported data can include the identified component items, the status of service change requests, and the implementation status of changes that have been approved from the CPM.

A status accounting system should be able to provide insight into the change management process. This makes it possible to track individual service change requests, but also to get an idea of the efficiency of the process. For example, the number of service change requests handled over a certain time period reflects the performance of the project or the process. Status accounting is therefore an important tool both in project evaluation and process improvement.

3.6.4 Service Change Auditing

The *Activity Checker* (AC) is responsible for the specification of the constraints that a well-formed service design should satisfy in order to check whether the application's design is in conformance to the main host design. Violation of the rules governing the activity checker will trigger a constraint violation event from the *Constraint Activator* (CA) to be returned to the *Change Propagation Mechanism* (CPM).

The purpose of change auditing is to ensure that the delivered product contains what has been promised and that it behaves as specified. This is accomplished using two different types of audits: functional and physical. The functional change audit verifies that the change items function as they should, in other words that their behaviour corresponds to that specified in requirements and design documentation. The physical change audit, on the other hand, verifies that the configuration items are fully contained in the product.

3.7 Software Development and Change Evolution

We see software development as the set of activities whereby a software system is built to satisfy the needs of its users. Typically, a software system is created because it automates mundane tasks, facilitates complex and time-consuming tasks, or makes otherwise unmanageable tasks possible, by utilizing the processing power and storage capability of computers. In many cases, it is essential that software systems behave correctly and generate accurate output, for example in life-support equipment, in cars, and in airplanes. We rely on software systems to perform their intended tasks

flawlessly, still they are (for the most part) developed by humans who have limited cognitive capability and can (and do) make mistakes.

A software process allows software development to proceed in a controlled and defined fashion, although its quality limits the quality of the produced system [72, 73]. Sommerville [73] defines software process as ". . . *a set of activities that leads to the production of a software product*." This definition opens up for much variation, and there are numerous different software processes that are suitable in different situations and for different types of systems. Ruhe and Saliu [74] noted in its article that sufficiently small computer programs can be produced in two steps: *analysis* and *coding*, but that more steps are necessary in a more substantial development effort.

According to Sommerville [73], the fundamental activities of any software process are:

- ✓ **Specification**: Defining the functionality of the system.
- ✓ **Design and implementation**: Producing the system according to the specification.
- ✓ **Validation**: Making sure that the system behaves in accordance with the customer's expectations.
- ✓ **Evolution**: Continually changing the system to better suit changing needs.

Different processes have different characteristics that make them suitable in different contexts. Hampered's Personal Software Process (PSP) is geared towards the individual engineer, and can help her improve planning and scheduling, and reduce the number of defects introduced in the code. The Team Software Process (TSP) helps organizations obtain leverage on their team work by building on and making use of the

disciplines dictated by PSP [75]. According to Humphrey [75], TSP is suitable for teams of 2 to 20 engineers. Agile processes are light and sufficient, meaning that they are not burdened by heavy plan-driven thinking; instead they capture what is necessary and leave out the rest. For example, the *agile manifesto* states that an agile process emphasizes interaction over documentation [76]. Consequently, agile processes work better where good interactions can be upheld, such as in smaller teams. Clean room is an example of a non agile development process that focuses on formal specification and formal verification using inspections and statistical testing, with the goal of producing zero-defect software [73]. However, the high level of formality makes changes costly.

The waterfall model is the first published model of a software process, and even today development processes are based on this model [73]. In a waterfall-based process, the phases of software development are sequentially arranged and do not overlap. For example, a set of needs is first captured in requirements, which then are built into a system, which subsequently is tested against the requirements. This assumes that we can learn everything about a system initially, and then do everything based on that. Already Ruhe and Saliu [74] pointed out that if testing occurs last, we cannot verify what we created until it becomes difficult, and thus expensive, to change something if we made mistakes. Also, only on rare occasions can everything about a system be learnt initially. Most of the time a model where learning is interleaved with doing is advantageous. Such models are called evolutionary, and examples include Boehm's spiral model [73] and Gilb's Evo model [77]. However, Sommerville notes that evolutionary development might be problematic for large-scale software development

due to difficulties in technical coordination, and recommends a process with some waterfall characteristics in that case.

Evolutionary development is different from iterative and incremental development, in that the latter two dictate how the system is built and can be released. Pfleeger [78] differentiates between *iterative* as building everything from the beginning but with reduced capability, and *incremental* as building certain parts first, and then adding new parts later on. However, also iterative and incremental development (and in particular the latter) support interleaved learning and doing, as some learning can be postponed to later iterations/increments. In this respect, the three concepts are similar. An advantage of interleaved learning and doing is that changes can be incorporated more easily. A big upfront design (as in a waterfall-like process) might be sensitive to changes, but if the design is created in an evolutionary manner, it can incorporate and evolve around the changes as they arrive. Ironically, this advantage is also a disadvantage; frequent changes can lead to a poor and fragile system structure [73]. This is especially true for complex systems developed by a large number of people, in which case changes must be controlled and coordinated.

Market-driven software development is different from tailor-made (or bespoke) software development. In the latter form of development, a single customer commissions a developer to produce a software system based on some identified needs. It is the job of the developer to capture and understand these needs, which most certainly will change (evolve) during the development, in the form of requirements. In market-driven software development, there is no single customer, but the system is instead produced with the goal of making it available on a market for potential

customers to purchase. For this to succeed, the system must have features and qualities that the customers desire.

However, the requirements for these cannot initially come from customers, as there are none, but must instead be foreseen and formulated by the developer herself [92]. When the system evolves over several releases, requirements for new features and qualities can of course be elicited from customers of the old versions. This is in fact necessary to motivate the customer to upgrade to the new release. However, a new problem arises: different customers may want the system to evolve (to change) in different and conflicting directions. Thus, market-driven software development opens up for a very change-prone environment.

The Capability Maturity Model Integration for Development (CMMIDEV) is a process improvement maturity model aimed at helping organizations improve their development and maintenance processes for products and services [72]. CMMI-DEV contains no less than 22 so called *process areas*, which are clusters of best practices within certain areas, formed to allow for the assessment of a company's capability or maturity. This neatly illustrates the fact that software development is a complex task that builds on a plethora of processes in order to be successful.

It can be seen that different development processes and different contexts call for different ways of handling change [8]. In this thesis, we assume in the development of such services and dependencies that changes must be formally controlled. The process for doing this is discussed next.

3.8 Chapter Summary

As earlier mentioned, different processes have different characteristics that make them suitable in different contexts. Change as used here is in its general context, referring to, a proposed one, a requested one, one necessary to fix a defect, or one required to implement a new requirement. Software development and change evolution is discussed here to buttress the need for change in an evolving technology environment. A component or functionality is a single entity that is subjected to change during the project and therefore needs to be under control as it has the likelihood of propagation. Therefore, change control consists of activities that allow management of changes by service change request analysis, service change identification, service change approval/disapproval, and service change implementation. Change management as used in this thesis and defined by this chapter, is the collective name for those activities that are associated with the change control function denoting the process of managing change. The research, therefore, defines service change management (to differentiate it with software change management) as “a technique for identifying likely faulty service components or functionality to enhance change control and status reporting for appropriate service auditing.” By this definition, we evolve the following service change management activities - Service Change identification, Service Change analysis, Service Change Cost Evaluation, Service Change approval or disapproval, and Service Change implementation and verification.

CHAPTER 4

CHANGE PROPAGATION FRAMEWORK

4.1 Introduction

The main focus of this chapter is to provide a framework for controlling and managing change propagation in our GUISET service provisioning environment.

Many techniques have been proposed to support change impact analysis at the code level of software systems, but there is no known reported effort to support change impact analysis at the service provisioning level. In this chapter, we provide some background information on change propagation and present an approach to supporting change impact analysis at the service level based on a framework to support service change propagation in our GUISET environment. The main feature of our approach is to control and assess the effect of change by analysing service dependencies, so that the process of change can be completely automated. We introduced five steps for change propagation framework and propose a manual technique which was used as a control process to confirm the results of the automated process.

4.2 Background Information

For years, enterprise information technology (IT) organizations have focused their attention on implementing defined change management processes and putting measures in place to ensure their enforcement. Internal and external auditors have driven home the importance of process enforcement, and the efficiencies to be gained

from process automation are well understood. But definition and enforcement of change management processes is just a first step on the road to mature software change management practices.

Change propagation is one of the key parts of system maintenance and evolution. In order to explain change propagation, we have to understand that a system consists of entities, functionalities and their dependencies. The dependency between entities A and B means that entity B provides certain services, which A requires for its correct functioning. Different programming languages or software control engineering systems may consist of different entities and dependencies. The dependency is consistent if requirements of A are satisfied by what B provides [80].

When a system maintainer makes a change in a system or service, he starts by changing a specific entity of the system or service. After the change, the entity may no longer fit with the other entities of the system or service, because it may no longer provide what the other entities require, or it may now require different services from the entities it depends on.

The dependencies which no longer satisfy the required provided relationships are called inconsistent dependencies (or inconsistencies for short), and they may arise whenever a change is made in the system or service. In order to reintroduce consistency into the service or system, a change propagation process keeps track of the inconsistencies and the locations where the secondary changes are to be made. The secondary changes, however, may introduce new inconsistencies, and this will require more changes to correct newer inconsistencies. The process in which the change spreads through the system is sometimes called the ripple effect of the change. For the

system maintainer, change propagation is a key process. The maintainer must guarantee that the change is correctly propagated, and that no inconsistencies are left in the system. An unforeseen and uncorrected inconsistency is one of the most common sources of errors in system engineering processes [80, 81].

Change propagation is made easier by supporting tools and techniques which improve both the efficiency and quality of the process. However, in order to develop effective tools, the problem must be correctly understood. This understanding is best formulated in terms of a formal model, which extracts the essential properties and separates them from accidental ones. Each model is based on a certain set of assumptions, both explicit and implicit. The model creates the foundation on which the tool developer bases the tools and techniques. The process of developing a framework or a prototype tool is a first validation of the model, and all the abstractions and assumptions built into it. In the final phase, validation of the tool must be made in a practical setting, where the usability, practical importance, effectiveness, etc., are assessed. Needless to say, all three phases are important for progress in the area of system support tools, and there is no real conflict between them [82, 83]. Change propagation also requires the development of mathematical models to predict the risk of change propagation in terms of likelihood and impact of change; and the development of a prototype computer support framework [84]. This research presents both the models and a framework tool based on it. Furthermore, it presents an example scenario of the use of the framework and the results obtained.

Modern development languages, tools, and techniques put IT organizations in a position to meet business demands faster by dramatically increasing the productivity of

the development organization. But increased productivity simultaneously tempts IT shops to roll software into production without applying appropriate controls, which can ultimately waste as much as or more money than was saved through those same productivity increases [85].

Increased adoption of componentized software architectures is one of the main technical drivers for better change management. Examples of greater componentization include such familiar architectures as the client/server model, the Web, and cross-platforms, as well as more recent innovations like service-oriented architecture. Enterprises that want to take advantage of the greater flexibility afforded by these architectures inevitably find that they must put controls in place to manage the risk that comes from having more moving software parts [85, 86].

Today's challenges [25, 87] of managing change across platforms are significant though relatively well understood, but the challenges associated with software change management in an SOA environment are new or still in the future for most enterprises. Conversely:

1. IT organizations that adopt SOA will find that the importance of mature change management grows exponentially. To start with, services are explicitly built for reuse, and changes to services thus have the potential to affect a large number of applications that consume them.

2. In addition, business services are embodiments of business processes and are, therefore, mission-critical software assets. Any defect in business services is likely to be a high-severity defect, and because the service is built for reuse, it is also a far reaching defect.

Business applications assist users in performing their daily work more efficiently and effectively and automate business processes. Business processes are a set of logically related tasks that are performed to achieve business objectives. For example, an on-line money order transferring business process consists of a sequence of tasks, such as, checking account balance, specifying transfer amount, and approving the transfer. Business process specification languages, such as BPEL4WS (Business Process Execution Language for Web Services) can be used to integrate various services, such as credit card authorization, into a service oriented business application.

Due to marketing strategies, organizations tend to reengineer their business processes in order to improve performance indices, such as cost and quality of services. When a business process is changed (e.g., adding a task, and removing a task), the source code implementing the business application needs to be updated in order to support the new business requirements. However, determining the impact of a business process change is not trivial and is challenging since business analysts are not experts in the source code and the spread of the changes across services may be too complex [25, 85, 87].

Business process changes usually result in code changes. These code changes may cause unexpected side-effects to other business processes. It is important for business analysts to evaluate the impact of business process changes especially when considering various alternatives. For example, a simple change to a business process such as the addition of a welcome screen or an order summary may seem trivial but may require a substantial amount of code changes. The code changes may be too costly in particular when the business value of such a summary page is considered.

4.3 Change Propagation Framework

Changes are endermic to software artefacts and the services provided by these artefacts. When a change is effected in a particular service connected to GUISET grid, it will be difficult to determine the propagation of this service changes. Modern distributed applications are grid-based and support flexible business collaborations. Most applications developed today rely on a given middleware platform which governs the interaction between components and the access to resources. To decide, which platform is suitable for monitoring changes to a given service, we propose a change propagation framework shown in Figure 4.1 to support change automation in our GUISET research.

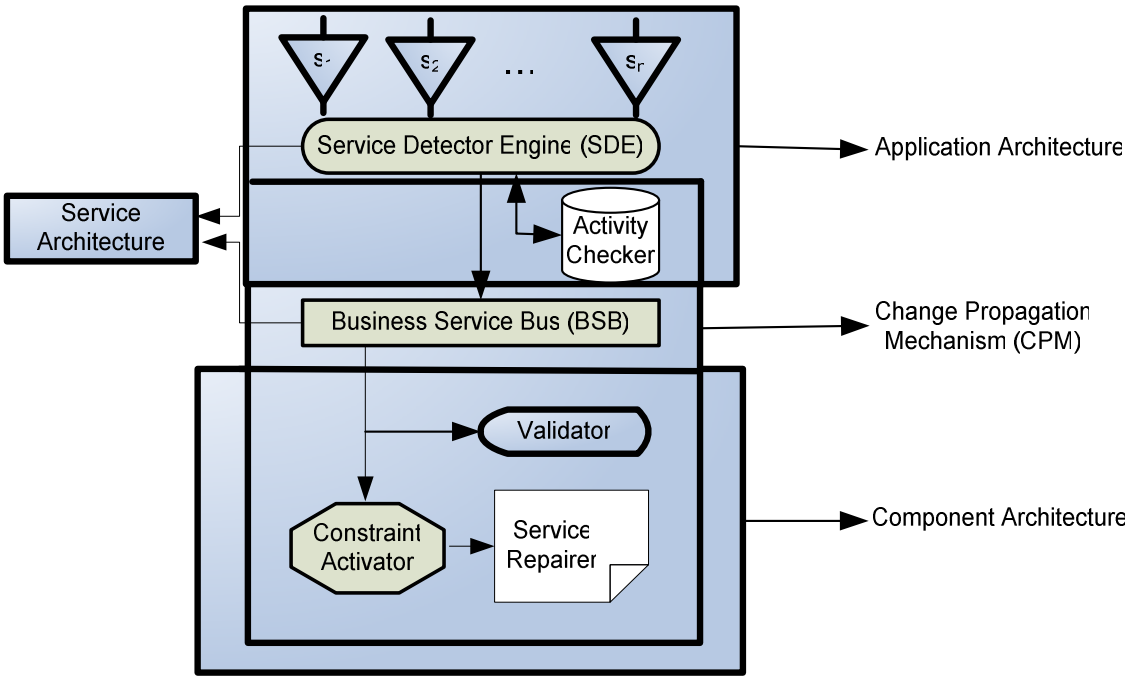

Figure 4.1: Change Propagation Framework

The *Service Detector Engine* (SDE) contains all consumer made available set of GUISET grid services (s_1, s_2, . . ., s_n) under utilization. The services contained in SDE are known to be interconnected to other services, therefore a service failure may affect any

other service that is dependent on it. In this case, the change can begin from any of the services and propagates according to the dependencies. The service maintainer is free to choose any of the services for the first modification. Whichever of the services he chooses to start with becomes the starting impact set (SIS) as defined in chapter 2. This processes take place at the application architecture level. SDE liases with *Business Service Bus* (BSB), a concept developed under Component Based Development and Integration (CBDI) [88] and incorporated into our framework to form *Service Architecture* (SA) responsible for providing a bridge between the implementation and the consumer, creating a logical view of a set of services, which are available for use and invoked by a common interface and management architecture. BSB is the infrastructure that enhances SDE to connect services and virtualize the services that are made available through the SDE. A core tenet of SOA is that service requestors are independent of the services they invoke. Therefore, the BSB is invisible to the service requestors and providers that use it. The need for the service change must first be checked by the Activity Checker(AC). The *A*C is responsible for the specification of the constraints that a well-formed service design should satisfy in order to check whether the application's design is in conformance to the main host design. Violation of the rules governing the activity checker will trigger a constraint violation event from the *Constraint Activator* (CA) to be returned to the *Change Propagation Mechanism* (CPM). This informs the *Service Repairer* (SR) of a triggered event calling for a way of fixing the violated constraint by performing actions, which change the application's design and keeps record of the propagated change (ripple effect). The mechanism *Validator* (V) is responsible for checking the consistency of the change to the design (through the AC),

which can result in further actions. The change propagation process can be reduced into the following steps:

1. Connect all known services to SDE in order for it to detect the available services, their interconnection and dependencies.
2. The AC will specify the required constraint and match each connected service with its host design.
3. Once there is a violation of the host design, the connected service or its dependencies, it means a service connection is broken or there is an immediate call notifying a breakdown in a particular service.
4. The triggered information which is first checked by the AC will be passed for validation by the mechanism Validator and the ripple effect will be recorded.
5. CA activates the relevant constraint immediately, and determines the level of propagation. Then the service will be fixed by the service repairer which is within the Change Propagation Mechanism.

The above steps take place in quick succession and explain how the framework works to effect changes and to monitor change propagation.

There are three major architectural perspectives for SOA namely: Application Architecture, Service Architecture and Component Architecture and our framework has these incorporated into it. The architectural perspectives merely signify the three major actions of the framework such as – *Service change identification*, *service level of propagation* and *service fixing*. This architecture has two views: Consumer and Provider. The salient aspect of the architecture is that the consumer of a service should not be interested in implementation detail of a service, but the service provided. This is

because the implementation architecture could vary from provider to provider, but still deliver the same service. Additionally, the provider should not be interested in the application that the service is consumed in, because new unforseen application will reuse the same set of services. The consumer's main interest is in the application architecture and the services used, but not in the detail of the component architecture. The interest is in some level of details in the general business objects that are of mutual interest, for example, provider and consumer need to share a view of what is a subscription. But the consumer does not need to know how the service component and database are implemented. Also, the provider is focused on the component architecture and the service architecture, but not on the application architecture. Again, they both need to understand certain information about the basic application in order to be able to set any sequencing rules including pre and post conditions.

SOA provides the need to be able to manage services as first order deliverables. The communication key between the provider and the consumer is service. There is the need, therefore, for a service provisioning architecture in the form of generic framework, that will ensure that services are not reduced to the status of interfaces, but have an identity of their own and can be managed individually and in sets. BSB as shown in our framework is incorporated to meet this requirement by providing a logical view of the available services for any business domain. BSB answers such questions as what services do I need?, what services are available to me?, what alternative services are available?, what will operate together and what services are connected to me [89]. Our framework is generic because it can be applied to general service provisioning engineering methodology to enhance monitoring change propagation. The most

important component of the framework is the Change Propagation Mechanism (CPM), which is represented and implemented within the service provisioning architecture and the component architecture. The CPM detects any change service due to the triggering effect generated and validated. The CPM notifies the SR of the ripple effect for immediate action of fixing the service.

With this change propagation framework, this research introduces the following three methods of making changes:

(I) Up-down Method

(II) Compulsory Change-and-fix Method

(III) Unsystematic Change-and-fix Method

For clarity, we explain these methods below:

4.3.1 Up-down Method

This method requires that, on discovery of the need for a change, begin with the service that is uppermost placed in the dependency and interconnectedness chain. The uppermost service becomes the starting impact set (SIS) and is identified for change while other dependent services will be on hold. After the first changes are effected, then the next uppermost will be effected while others are kept on hold. The changes, therefore, propagate from up to down and ends when the last of the services has been checked by the activity checker and modified accordingly.

4.3.2 Compulsory Change-and-fix Method

In this method, the change can begin in any of the affected services and propagates accordingly. The maintainer is at liberty to choose any of the services for the first modification. After performing the first modification, the maintainer is forced to

work only on the remaining services. When all the services and their ripple effects have been taken care of, the system again returns to its consistent state. This method differs from the previous (up-down method) in the following assumptions:

(a) Ripple effects of changes propagates in all direction and not just downwards.

(b) The first change can begin anywhere and not just in the uppermost class as long as the first change is noted as the starting impact set (SIS).

4.3.3 Unsystematic Change-and-fix method

Unsystematic change-and-fix method is random in nature and follows the same assumptions as the compulsory change-and-fix method. Here changes begin anywhere within the services as change always propagate in all directions. The difference is that the service maintainer is free to modify any service at any time, even if there is no identifiable fault. This is a preventive maintenance method. As a recommendation for an unfamiliar project environment and for a novice service maintainer, the compulsory change-and-fix method is best because errors are more likely when there is not tight control.

However, the expert maintainer may feel uncomfortable with the level of control exerted by the compulsory change-and-fix method. For the expert service maintainer, the unsystematic change-and-fix method acts as a guide, helping him to be organized and not to forget any ripple effect, while allowing him the complete freedom to update any service.

In view of the main goal of CIA to determine what would be affected by a change to a particular service, this research also proposes the following manual logical steps to serve as a control process for performing CIA during service provisioning:

Step:

1. Identify the service to be modified
2. Trace back through any relationships that indicate a dependency on that service. These relationships will have the selected service as a target of the relationships. This impact analysis thus results in a list of “dependent services” that depend directly on the selected service.
3. If a change affects these “dependent services”, then other services that depend on these “dependent service” will also be affected. The impact analysis must, therefore, act recursively looking for relationships that have any of the dependent service as target.
4. This process continues until a complete graph is obtained starting at the selected service and finishing with service on which nothing else depends. Each service in the graph could therefore be affected by the change to the selected service.
5. Each service change must be carried out using any of the methods described in section 4.3.

4.4 Existing Change Propagation Frameworks

One relevant question that any researcher will like to ask is: what is wrong with existing frameworks and why can we not use them? Although they are different proposals on frameworks for change propagation, there are mainly handling issues concerning propagation in source code design. This is the primary motivation for the framework in this research.

Business process changes usually result in code changes, and code changes may cause unexpected side-effects to other business processes. This makes it important for service provider to evaluate the impact of service changes especially when considering various alternatives. This change propagation framework becomes a necessary solution for service providers as there is no known existing framework which is mainly targeted for monitoring and controlling service change propagation.

Researchers have proposed various approaches to control the amount of change propagation and avoid hidden dependencies. These proposals include, most notably, a framework for understanding conceptual changes on evolving code [90], where the proposed framework is to characterise types of concept changes during evolution. The framework describes transformations in the geometry and interpretation of regions of source code. Another framework [91] focuses on grouping related entities in the same structure to ease code modification and understanding. This was to enhance predicting change propagation in software systems.

Moreover, various researchers have proposed tools, models and algorithms to assist developers in understanding their software and determining the extent of propagation for each change to the source code. For example, dependency analysis algorithms such as slicing [91, 92] in association with entity dependency browsers [93], software understanding tools and architecture visualization tools [94] are used by developers and researchers to maintain and evolve large complex long lived industrial systems. Also proposed, is a model for change propagation based on graph rewriting [95] during software maintenance and evolution. In this model, change propagation is modelled as a sequence of snapshot, where each snapshot represent one particular moment in the process, with some software dependencies being consistent and others being inconsistent.

From the above proposals, we can easily note that much concern has been on source code. This has created the need for evolving a framework that can help in change propagation control during service provisioning. This is what the framework for this research achieves.

4.5 Change Propagation Analysis

Changes to product design occur very often during the various phases of the product development lifecycle, from concept through definition and development, to manufacture, and then into service. Products are continuously modified and changed, as a result, most product development involves the steady evolution of an initial design. This is necessary, both to eliminate mistakes that have been made during the initial design and manufacturing process, and to adapt the design to new requirement [96].

Irrespective of the cost of these changes, the challenges of implementing them have not changed. Changes are required to fix problems or to improve or update products or to change a requirement. It is often the case that new products are variants or derivatives of existing ones. Hence, changes can be of different origins or nature, and often not all the consequences of a given change are expected or wanted. The effectiveness and efficiency with which an organization can predict or control these changes could have a significant impact on the competitiveness [80].

At first glance, design changes often seem deceptively simple. A required change is first identified, a new solution for the change is generated and the change is effected. Frankly, the reality is far more complex than just that explanation. As all parts of a design are connected to each other, design changes can also be connected. Consider an interconnectedness of A, B, C and D. If part A is changed then part B might need changing. A change to part B leads to a change in C, which in turn might be connected to part D. A single design change might propagate to others, transforming the initial changes through a series of other changes in the whole design [84].

The degree to which a change propagates through a product depends on the product complexity. Simon [85] defines product complexity with regards to the connections between its parts, where connections between parts can never be totally avoided. In a complex product, where the constituent parts and systems are closely dependent, changes to one configuration item of a system are highly likely to result in a change to another item, which in turn can propagate further. As widely acknowledged [86], change propagation analysis (CPA) is seemingly important for predicting the impact of change, in order to improve the capacity to manage time, cost, resources and quality.

Current practices for analyzing inconsistencies due to engineering changes often use configuration management procedures, relying heavily on human communication, the knowledge and experience of individuals in a specific system area, as well as common sense. Due to the globalization of industrial sectors, there appears to be a need for a more integrated and shared CPA approach within organisations and across their supply chains [96]. Change propagation can only be made easier by supporting tools and techniques which improve both the efficiency and quality of the process. But to develop effective tools, the understanding must be formulated in a formal model, which extracts the essential properties and separate them from accidental ones. The model must be based on a set of foundation assumptions on which the tool developer bases the tool and techniques [81]. This research, therefore, presents a change propagation framework to control, monitor, detect the need for change and ensure the consistency of assumptions amongst interdependent service entities in our GUISET environment.

4.7 Chapter Summary

We identify change propagation to be necessary in system maintenance and evolution. But to understand change propagation, we need to also understand that a system consists of entities, functionalities and their dependencies. The dependency between entities A and B means that entity B provides certain services, which A requires for its correct function. We say that a dependency is consistent if requirements of A are satisfied by what B provides. In making a change, a system maintainer starts by changing a specific entity of the system or service. After the change, the entity may no longer fit with the other entities of the system or service, because it may no longer provide what

the other entities require, or it may now require different services from the entities it depends on.

The dependencies which no longer satisfy the required provided relationships are called inconsistent dependencies, and they may arise whenever a change is made in the system or service. To maintain a state of consistency in the service, a change propagation process keeps track of the inconsistencies and the locations where the secondary changes are to be made. The secondary changes, however, may introduce new inconsistencies, and this will require more changes to correct newer inconsistencies. When change propagates through the system, we called that the ripple effect of the change. For the system maintainer, change propagation is a key process as he or she must guarantee that the change is correctly propagated, and that no inconsistency is left in the system. An unforeseen and uncorrected inconsistency is one of the most common sources of errors in system engineering process. The framework described in this chapter is to assist service maintainers to monitor and control change in any typical grid service provisioning environment like GUISET. The research aspect describe in this chapter therefore propose three methods for change propagation as: Up-down Method, Compulsory Change-and-fix Method and Unsystematic Change-and-fix Method.

CHAPTER 5

FORMAL MODELS FOR CHANGE PROPAGATION

5.1 Introduction

The main focus of this chapter is to present two formal models as a set of change impact analysis metrics to work alongside the framework and help maintainers to quantatively measure susceptibility to change and the effects of change during service provisioning in GUISET as our typical grid environment. We demonstrated this process in an example scenario and discuss how Bayesian statistics would form a useful approach for change impact prediction. We validate our concepts using the service structure shown in our example scenario and generate values for a hypothetical period of 5 years upon which we calculate their dependencies, fault propagation and number of changes and show their relationships graphically. Results obtained show that the higher the dependencies of services to the changed service, the higher the fault propagation and invariably, the higher the impact of change.

As this research proposes our models as metrics suites for change propagation, the chapter extends our proposal of the current metrics suite for object oriented programming to the new paradigm of aspect oriented programming. It also discusses existing metrics used for change impact analysis.

5.2 Change Impact Analysis Factor Adaptation Model (CIAFAM)

When a change of service is considered during service provisioning, it is necessary to identify service components that may be impacted after such a change. This enhances the services to keep running after a change implementation. Services absorb a change easily if the impacted components are small in number. One effective method to account for changes in services is to perform CIA and our framework is accessed by the impact model described. Our main concern is pivoted on how other services react to changes that lead to propagation.

For a given change K in a service P, we describe a set of impacted services as a boolean expression. The Impact Analysis Factor (IAF) for such hypothetical change is given by [116, 117]:

$$IAF\ (K,P) = A*(\sim \rho) + A'$$

Where *, +, ~ denotes the usual boolean operators: conjunction, disjunction and negation respectively.

K = a given change,

P = a given service,

A = there is an association between K and P,

ρ = K is derived from the change service,

A' = there is an occurence of aggregation link between K and P and

IAF = Impact Analysis Factor.

This expression implies that a service in association (A) with K and not derived (~ ρ) from the change service K or a service that is in aggregation link (A') with K is

impacted. It is important to state that this impact model only predicts, which services would be impacted if a change was really made. If a service is really impacted, it means there is the propensity of propagation in which case the IAF becomes 1. We concentrate on changes that have a syntactic impact, therefore, appropriate measures are based on the impacts that are dependent on the static nature of the provisioning system. This implies that impacts have a likelihood of propagation .

5.3 Fault and Failure Assumption Model

Depending on the architectural level, time phased and other specific service parameters, SOA failure modes may change. In modern SOA, common failures are due to unavailable infrastructure, client crash, service failure, server crash, session failure and component failure. Therefore, a generic failure F_k can be defined as a function of:

$$F_k = f(al, t_p, ss_p) \quad (1)$$

Where;

al = the architectural level of the faulty components

t_p = the time phase during which the fault occur

ss_p = the set of specific service parameters identifying the state of the particular service involved in the failure.

If each failure F_k is identified, the system failure modes can be represented as:

$$F = F_1 \mid F_2 \mid .. \mid F_n = \sum_{k=0}^{k=n} F_k \quad (2)$$

where n = no. of possible failure of nodes.

Meaning that the system fails if at least one of the identified failure occurs. Our failure is recorded as a boolean value (0,1) with respect to t_p.

Increasing the redundancy degree may lead to an increase in possible sources of failure resulting in potential decrease in dependability. To understand the impact of redundancy, dependability and interoperability on our framework, the failure model is necessary. Our fault assumption is based on a fail-silent assumption where either a service is actively operating or does not answer at all. This assumption is justified on the basis of our CIAFAM whose IAF is a boolean (0,1). When the value is 1, it indicates a fault (requiring change), but when the value is 0, it is in its active state.

To analyze the error type that a faulty service may induce in a grid environment, we formulate the concept of failure that will enhance change prediction as:

$$F = f\ (al, t_p, t_m, i, d) \qquad (3)$$

where al and t_p are as previously defined and ssp = t_m, i, d

where

t_m = time (in months) when a fault is detected

$$t_m = x, y\ \ \{0 \leq x \leq 6; 6 < y \leq 12\} \text{ months}$$

i = the particular service item involved in failure

$$i \in I = \{a, b, c, \ldots, z\}$$ where I is the set of available services.

d = the descriptor of the faulty session [116].

5.4 Example Scenario with Explanation

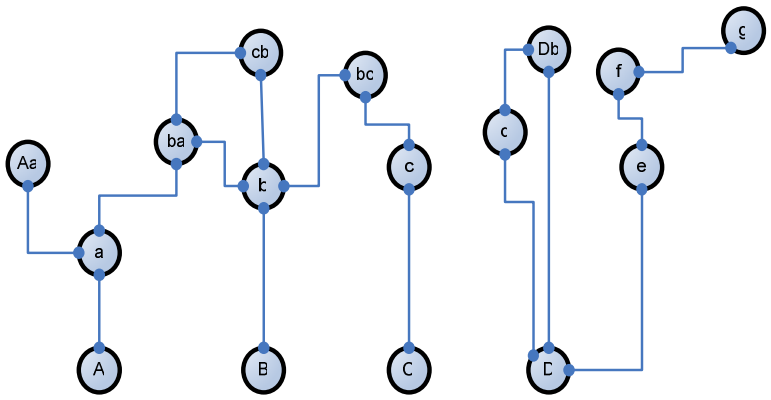

Figure 5.1: Causally related set of services connected to the framework

Figure 5.1 defines a set of A, B, C, D services which are connected to our framework at the point of s_1, s_2, s_3, and s_4 of the SDE respectively. All service interconnection is defined by a causal relationship. This causal relationship can be affected by a failure need of a change in service resulting from either unavailable infrastructure, client crash, service failure, server crash, session failure and component failure at any point in time. Services a, ba and Aa are causally related to service A, while services b, ba, cb and bc are causally related to service B. Also service c, and bc are causally related to C while d, e, Db, f, g are causally related to D.

The adaptation model defines expected results to be obtained as boolean, where a value of 0 signifies no changes made, while a value of 1 signifies that there was a synthetic impact, meaning a change was effected. Values are expected to be recorded within a period of 6 months (x) and 12 months (y) respectively. Therefore, for each year, one finds the first boolean value representing changes that have either been made or not, within the first 6 months (x) and the second value representing changes that have either been made or not within 12 months (y) of the year. We have previously explained

that changes propagates, hence it is possible to use the recorded values over certain period of years, through Bayesian statistics, to predict the effect of this propagation that will require changes for the next year and so on.

On a general note, the consequence of obtaining the value 1 is indicative of low comprehensibility, hence low reliability. This value also serves as quality measure, as the value of 1 actually indicates low value of the quality attribute. The attribute that is being measured here is a service change for productivity, hence quality. Productivity is an external attribute of the service, which is clearly dependent on many aspects of the process and the quality of service delivered. Service change is a maintenance issue with a great impact on productivity and hence affecting service quality. Our mention of reliability is due to the need for prediction. This is because the values to be obtained are on the basis of observing times (half-yearly) between faults leading to failures during service provisioning operation, and are used as parameter estimates to make statements about future reliability. Of particular note, is the fact that reliability requires collection of inter-failure data during service provisioning operation.

Service maintenance is costly and difficult. It is not always clear what the impact of any type of change to service will have across all the services. This CIA technique shows the maintainer what the effect of any change will be on the system. Our generic framework has proven to offer the potential to improve the stability and efficiency of service provisioning and cut the cost of maintenance.

5.5 Change Impact Prediction using Bayesian Statistics

The Bayesian statistical approach has proven useful for both inferential exploration of previously undetermined relationships among services as well as descriptions of these relationships upon discovery. The process of service change prediction in a service provisioning environment can be computationally intensive and NP-hard in its algorithmic implications. Predicting a change of service in an SOA service provisioning environment for a solution to a problem is usually an NP-hard problem resulting in a combinatorial explosion of possible solutions to investigate. This problem is often ameliorated through the used of heuristics, or sub-routines to make worthwhile choices along the SOA decision tree. We have used the Bayesian approach to replace heuristic methods by introducing a method where the probabilities of SOA decision tree are updated continually during predictive decision making.

We express the Bayesian approach as

$$P(H \setminus q, r) = \frac{P(H / r)P(q / H, r)}{P(q / r)} \qquad (1)$$

The term $P(H \setminus q, r)$ is defined as the "posterior probability" which is being continuously updated. It is the probability of H after considering the effect of r on q. The term $P(q / H, r)$ is the marginal probability known generally as the likelihood, and gives the probability of the evidence assuming the hypothesis H and the background information r is true. The term $P(H / r)$ is called the prior probability which measures the strength of belief probabilistically of the services, prior to any execution of experiment and may depend on the strength r service having the assumption of being true. For computational exigency of discrete nodes for SOA services, the updated services

marginal probability density function $P(H \setminus q, r)$ is calculated using the chain rule of probability

$$p(x_1, x_2, ..., x_n / \eta) = \prod_{i=1}^{n} p(x_i / x_1, x_2, ...x_{i-1}, \eta) \qquad (2)$$

Thus for the product rule of probability, equation (2) for each x_i, $\prod_i \subseteq \{x_1, ..., x_{i-1}\}$ are set of services that renders $x_i \ and \ \{x_1, x_2, ..., x_{n-1}\}$ conditionally independent in an SOA system. Therefore we have:

$$p(x_1, x_2, ..., x_n / \eta) = (p(x_i / \prod_{\kappa}, \eta) \qquad (3)$$

Where i = independent variable set of services and k = no of possible services in i.

With equation (3), the Bayesian environment structure then encodes the assertion of conditional independence in equation (2). Thus by this assumption, a Bayesian structure is a directed acyclic graph such that (i) each variable in the domain of the environment corresponds to a node in the SOA and (ii) the parents of the node corresponding to x_i are the nodes formulated from binomial distributions:

$$p(x) = \frac{n\,!}{x!(n-x)\,!} p^n (1-p)^{n-x} \qquad (4)$$

Where n = no. of possible failure of nodes and x represents the live nodes

Substituting equation (4) into equation (1), we obtain the updated predictive a posteriori P (H\q, r). With this computational updating, the service provisioning environment is predictable.

5.6 Limitation of Using the Traditional Approach

Traditionally, heuristic methods are used to solve combinatorial optimization problems which are known to be NP-hard. Admittedly, good solutions could be found efficiently for some real optimization problems of realistic sizes and less complexity. The existing exact heuristics algorithms and the application scope for solutions of this NP-hard problem are broader, yet the situation cannot be called satisfactory [118]. In the studies of algorithms, two key challenges have been to find algorithms with (i) provably good runtime and (ii) provably good or optimal solution quality [119]. It is necessary to exploit multiple problem solvers in hard combinatorial domains because large-scale combinatorial optimization problems pose difficult challenges and the algorithms guaranteed to find optimal solution are often too costly [120].

The main focus of this research is to use model-based approach instead of heuristic methods. The use of model-based approach as presented in our fault and failure assumption model would help to make predictions in the event of a failure in service. The Bayesian statistical approach for solving similar NP-hard problems [121, 122] has proven to be satisfactory for this research with respect to our GUISET research focus. Additionally, using heuristic algorithms always lead to the construction of a solution with a randomized process, which for example, randomly selects among remaining services to add to the end of the service sequence. This again would not satisfy our aim of predicting which services would be impacted if a change is really made. We concentrated on changes that have a syntactic impact, therefore, appropriate measures are based on impact that are dependent on the static nature of the provisioning system. This again is achieved by our CIAFAM model.

5.7 Empirical Validation of the Models on the Change process

Our formal models define expected results to be obtained as boolean, where a value of 0 signifies no changes made, while a value of 1 signifies that there was a syntactic impact, meaning a change was effected. Values are recorded within a period of 6 months (x) and 12 months (y) respectively. Unfortunately, this is a time consuming process and would require service maintainers to attempt implementation of our framework in their maintenance process to automate and efficiently generate data for empirical analysis.

From the example scenario in section 5.4 and our models, we generate hypothetical values and represent them in three years (1st, 2nd and 3rd). With data generated for the three years and with the understanding of our previous explanation that changes propagate, we use the recorded values over the period of three years, through the Bayesian statistics, to predict the effect of this propagation that will require changes for the next two years (4th and 5th). Although values for the last two years were obtained by prediction, yet we were not prepared to take chances, but to be sure that they represent the real values, we extended our rigorous time consuming method to confirm their validity before using them.

For each year, you find the first boolean value representing changes that have either been made or not, within the first 6 months (x) and the second value representing changes that have either been made or not within 12 months (y) of the year.

Table 5.1: Experimentally Obtained and Predicted Results

Main Services	Linked Services	1st (x, y)	2nd (x, y)	3rd (x, y)	4th (x, y)	5th (x, y)
	a	0,0	0,0	1,0	0,0	0,1
A	ba	0,0	0,0	1,0	0,0	0,1
	Aa	0,0	0,0	1,0	0,0	0,0
	b	0,1	0,0	1,0	0,1	0,0
	ba	0,1	0,0	1,0	0,1	0,0
B	cb	0,1	0,0	1,0	0,1	0,0
	bc	0,1	0,0	1,0	0,1	0,0
	c	0,1	0,0	1,0	0,1	1,0
C	bc	0,1	0,0	1,0	0,1	1,0
	d	0,0	0,0	0,0	1,0	1,0
	e	0,0	0,0	0,0	0,0	1,0
D	Db	0,0	0,0	0,0	1,0	0,1
	f	0,0	0,0	0,0	0,0	0,1
	g	0,0	0,0	0,0	0,0	0,1

Generally, as earlier explained, obtaining the value 1 shows low comprehensibility, invariably low reliability. This is indicative of the quality measure, since the value of 1 actually indicates low value of the quality attribute. Recall that, the attribute that is being measured here is a service change for productivity, hence quality. Productivity is an external attribute of the service, which is clearly dependent on many aspects of the process and the quality of service delivered. Service change is a maintenance issue with great impact on productivity and service quality. Our mention of reliability is due to the need for prediction. This is because the values are obtained on the basis of observing times between faults leading to failures during service provisioning operation, and are used as parameter estimates to make statements about future reliability. Of particular note, is the fact that reliability requires collection of inter-failure data during service provisioning operation [1].

From results obtained in the scenario example and recorded in table 5.1, we extract the characteristics of services where faults occurs, we calculate their dependencies and the

corresponding number of faults propagated. Recall that in our CIAFAM, we mention that our interest is where syntactic impact is involved. That is, we concentrate on changes that have a syntactic impact; therefore, appropriate measures are based on impacts that are dependent on the static nature of the provisioning system. We, therefore, represent the details in the following table 5.2 showing the actual impact set.

Table 5.2: Example Scenario Dependency – Fault Propagation Charateristics

Linked Service Fault Source	No. of dependency	No. of fault Propagated
B	4	5
A	3	2
C	2	1
D	1	0
Db	0	0

Representing the details of table 5.2 in a dependency – fault propagation relationship, we have the following figure 5.2.

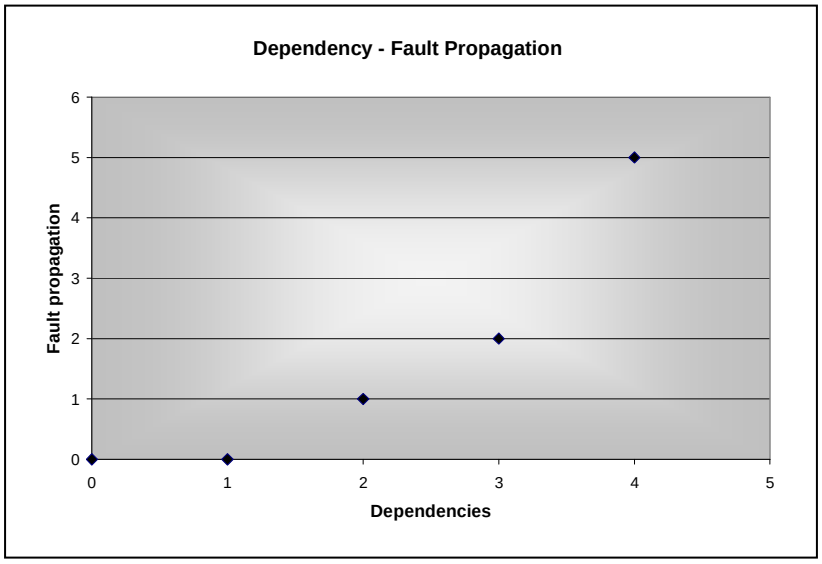

Figure 5.2: Dependency-Fault Propagation Characteristics

The propagation process is necessary because if not carefully controlled after a change, it might result into avalanche of faults which may increase the rate of inconsistencies knowing that the goal of change propagation is to ensure consistency after a change. As it is not enough to check the level of propagation only, it is also necessary to check what impact this change has on the overall service which will give the actual impact set. Therefore, from the actual impact set, we obtain the number of changes and compute their corresponding impact as expressed in table 5.3 below:

Table 5.3: Example Scenario Change – Impact Charateristics

Sources of Fault	No. of changes	No. of impacted Services
B	4	5
A	3	4
C	2	3
D	1	2
Db	0	1

Figure 5.3 below shows the change – impact Analysis characteristics derive from table 5.3

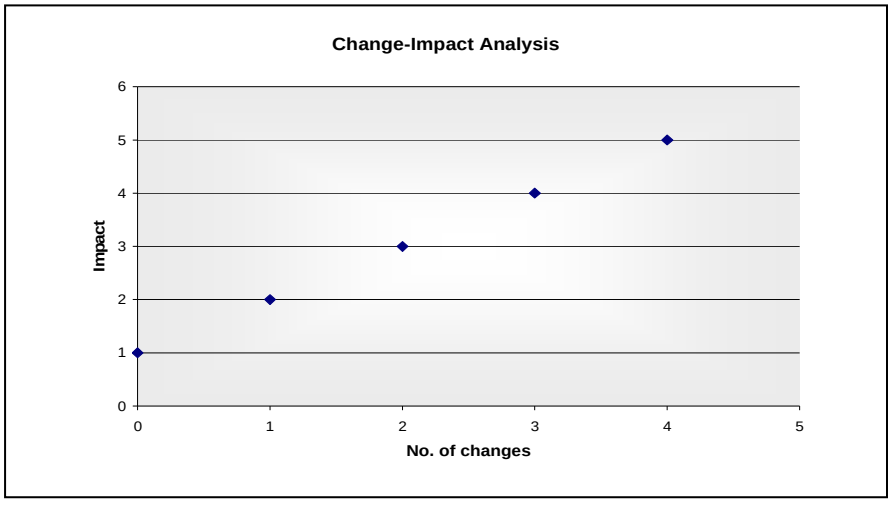

Figure 5.3: Change – Impact Analysis Characteristics

5.7.1 Discussion

We measure change impact based on the set of services that are affected by the change. Consequently, we concentrate on the number of services that are affected and their dependencies to describe the level of fault propagation. This is why we extracted only services that have a fault and their dependencies, and express this in Table 5.2 and consequently determined the characteristics of these dependencies and the corresponding fault propagation graphically as shown in figure 5.2. The general understanding obtained from this graph is that, the higher the dependencies, the higher the rate of fault propagated. Thus the greater the number of changes requires to keeping the main service in a consistent state. The affected services' complexity often determines how severe the change is. The higher the number of service dependencies, the higher the level of complexity and invariably the more severe the change. Considering our example scenario, changes that do not affect any other services because their number of dependency is either one or zero are limited in scope (e.g. linked services d and Db) and, therefore, have zero fault propagation.

Determining the severity of a change is a function of the impact the change has on the other services. We, therefore, obtain the number of changes required and compute their corresponding impact as describe in table 5.3. The relationship between the change and impact is describe in the graph represented in figure 5.3, indicating that, as the number of changes increase, the impact also increases. As noted earlier, the number of changes is also affected by the number of dependent services. Therefore, if the dependencies are high, the number of changes will be high, and consequently the impact will be high.

5.8 Proposal for New Service Development Platform Metrics

During the course of this research, we uncovered the fact that service interface development is currently been achieved in an Object Oriented (OO) environment. But we noticed that technology is drifting towards Aspect Oriented programming (AOP), and this will gradually replace existing service development platforms, due mainly to the kind of evolution in software engineering discipline. This evolution drift is anchored on an attempt to address issues of crosscutting and tangling effects observed in Object Oriented programming (OOP). Therefore, we decided to also contribute in proposing the extension of relevant existing metrics that will support the concern that is addressed in Aspect Oriented (AO) paradigm.

The general observation is that, software measurement is an empirical means of determining the complexity of the overall code with a view to producing effective and quality software products as well as predicting the overall cost. This means that evaluating software is a function of the metrics obtained from determining its complexity.

AOP is an entirely new technique proposed to handle issues of separation of concerns and cross-cutting in software design and implementation [97, 98]. AOP achieves its aim through providing explicit mechanisms in capturing the crosscutting concern structure in software systems. Synchronization, exception handling, resource sharing and performance optimization which are very difficult to cleanly express in source code can now receive a solution through aspect oriented programming languages which can be used to modularize the crosscutting structure of those concerns. These issues were referred to as code tangling and AOP can now assist in making them more apparent thus

leading to designing and maintaining code easily [99].

AO software contains code whose basic unit comprise of an aspect with an advice instead of a class for the OO software. The inclusion of join points which is a possible execution point that triggers advice further complicates the static relations amongst aspects and classes [99].

A large number of metrics are now available for software quality engineers to select from. The question is no more lack of metrics, but the selection of those metrics which meet the specific needs of the software project under consideration. OO metric were exploited from a number of metrics used in structural programming with little adjustments to satisfy the needs at that time. Yet, some were specifically developed to meet object oriented software and would be needless to implement them in structural programming [100]. The need for metrics which would reasonably and effectively give detailed meaning to software cannot be overemphasized. Properties that software should possess to increase their usefulness have been recommended by several researchers. As an example, Basili and Reiter [101] proposed that metrics should be sensitive to externally observable differences in the development environment, and must also corresponds to intuitive notions about the characteristics differences between the software artifacts being measured. A greater number of the properties recommended are qualitative in nature, therefore most proposals tends to be informal in their evaluation of metrics [102]. To cleanly follow the reasons for our proposal, we would like to consider defining some of the key terminology associated with this new drift.

5.8.1 AOP Key Terminology

To create a good understanding about the subject matter, it is imperative at this point to describe and explain the key terminology and fundamental concepts associated with this new software paradigm:

1 *Crosscutting*: Concern behavior that is triggered in multiple situations.

2 *Joinpoint*: A possible execution point that triggers *advice*.

3 *Aspects*: an aspect is defined as a special kind of concern or non-functional element of code arising from a post object implementation. An aspect is a concern whose functionality is triggered by other concerns usually in multiple forms. It packages *advice* and *point_cuts* into functional units in much the same way that Object Oriented Programming (OOP) uses classes to package fields and methods into cohesive units. A *concern* is any code related to a goal, feature, concept, or "kind" of functionality. For instance, there may be a logging aspect that contains advice and pointcuts for applying logging code to all setter and getter methods on objects. Aspects are not just a neat trick or careful technique for adding logging or synchronization or other simple functionality [103] to source code. Aspects are a natural evolution of the object-oriented paradigm which provides a solution to some difficulties encountered while modularizing object-oriented code. In most cases, functionality does not work, and the same lines of source code are repeated in many different object-oriented classes because those classes each need that functionality. It cannot easily be wrapped up in a single place. Instances of this kind of code are found in audit trails, transaction handling, and concurrency management, such code can now be modularized using aspects.

4 *Point_cuts*: Point_cuts typically define the points in a model where advice will be applied. For example, point_cuts define where in a class, code should be introduced or which methods should be intercepted before they are executed [103]. Point_cuts are also known as join_points. A method call join point is the point in the flow when a method is called, and when that method call returns. Each method that calls itself is a join_point. The lifetime of the join_point is the entire time from when the call begins to when it returns (normally or abruptly), but execution is at the join_point only at the moment the call begins and the moment it returns. A pointcut therefore describes a point in the execution of a program where crosscutting behaviour is required [104].

5 *Advice*: Advice is code that crosscuts or is applied to the existing model and is commonly referred to as a mix-in. Advice code [103] modifies the behaviour or properties of an existing object. In AspectJ, advice is the implementation of behaviour that crosscuts the set of execution points defined by the poincut. This makes advice an obvious construct to use to implement the sequence of behaviors defined in sequence diagrams in the crosscutting themes [103]. Advice can be defined to execute before, after, or around the execution points defined by pointcuts. It is possible for the programmer to decide to add advice before a join_point runs or after it finishes running, and can also force advice to run instead of the join_point. Whenever the point at which an aspect adds behaviour is defined, the behavior to be added must also be defined [105]. An advice is the triggered behavior.

6 *Weavers*: The aspect weaver accepts the component and aspect programs as input, and emits in some cases, a C program as output. The weavers' job is integration,

rather than inspiration [106]. Aspect weavers work by generating a join_point representation of the component program, and then executing (or compiling) the aspect programs with respect to it.

5.8.2 Reasons for the Extension of OO Metrics to AO

It is an interesting thing to note here, however, that the signs of a new age are everywhere and the economic and social well-being of nations depends increasingly on their ability to generate and access new knowledge by building on the previous to create an understanding of the present. The potential possed by software metrics is very great thus making their use in managing software development important. Many software metrics have been suggested, but selecting which to use is a difficult task since most of the articles on metrics promote the author's favourite metric as the best. There is no objective approach which can be use in selecting the best metric for a specific purpose [107]. It is, therefore, important at this point to state the following reasons for proposing the extension of some OO metrics to AO metrics by means of possible upgrade:

1. The impact that reuse has upon a metric is the key reason for suggesting that the "successful" metrics which has undergone validation and has been applied to related designs such as OOP metrics should be commended to still be useful for the current research on the metric for AOP.
2. Software metric reuse would reduce the quantity of software that must be developed from scratch thus allowing a greater focus on quality as it would help to improve productivity and resulting in the reliability of the reuse metrics for our

programs, designs and specifications.

3. The extension would serve as a capital investment as the high cost of verification in reused modules would be spread out over many products.
4. It would save development or coding effort as the effort for each product can be applied in building new software infrastructure which can be utilized in subsequent products.

To be able to access improvement, it is proper to analyze current level of practices and determine the usefulness of a new technology by simply comparing the need in relation to time, effort and cost. When the practice of a current metrics is accessed, the rate of improvement increases. When accessed and analysed, the rate of improvement will accelerate [117].

5.8.3 Towards a Metrics Suite for AO Software

The proposal from this research considers C-K metrics [108] which have an industrial approval, to be adopted as part of metrics suit for AOP. We also suggest that since program dependencies are dependence relationships existing between program elements in a program that are determined by control flows and data flows in the program, they can then be regarded as one of intrinsic attributes of programs. Hence, in measuring program complexity, it is reasonable to take program dependencies as one of the objects of measurement. Following this, we proposed that the set of complexity metrics [99] defined in terms of program dependence relations be adopted to measure the complexity of aspect-oriented programs. These metrics are:

1 *Module-level Metrics* – where some metrics are defined at the module level based

on the Advice Dependence Graph (ADG), Introduction Dependence Graph (IDG) and Method Dependence Graph (MDG). These metrics are said to be used to define from a general point of view, measures of various complexities of a piece of advice, an introduction or a method. If α is a module (an advice, an introduction or a method), then generally the larger the value of a metric of α, the more complex α is.

2 *Aspect-level Metrics* – Here some aspect-level metrics for an individual aspect based on its Aspect Interprocedural Dependence Graph are defined. The metrics can be used to measure various complexities from different viewpoints. If β is an aspect, the larger the value of the metric of β, the more complex β is.

3 *System-level Metrics* – finally some metrics at the whole system level based on the Aspect-oriented System Dependence Graph (ASDG) are defined. The metrics is used to measure various total complexities of an aspect oriented program from various viewpoints. If ρ is an aspect-oriented program, then in general the larger the value of a metric of ρ, the more complex ρ is.

5.9 Existing Impact Analysis Metrics

As previously mentioned in this thesis, several metrics for impact analysis exist, but they all measure impact analysis at the software code level. This is mainly why we proposed our models to be adopted as metric suit for measuring impact analysis during service provisioning. It is obvious that metrics used to address impact analysis at code level cannot be applied to address issues of impact analysis at service provisioning level. Metrics are useful in impact analysis for various reasons. They can, for example, be used to measure and quantify change caused by a new or changed requirement at the point

of the impact analysis activity. Metrics can also be used to evaluate the impact analysis process itself once the changes have been implemented. This is illustrated in Figure 5.4, in which two measure points are depicted; one after the requirements phase has ended and design is about to start and the other when testing has been completed. Using these measure points, one can capture the predicted impact (the first point) and compare it to the actual impact (the second point). This kind of measurement is crucial for being able to do an analysis and learn from experiences in order to continuously improve the impact analysis capability. The figure is simplified and illustrates a learning cycle based on a waterfall-like model. Impact analysis can be used throughout the life cycle in order to analyze new requirements and the measure points can be applied accordingly i.e. whenever a prediction has been conducted and whenever an implementation has been completed [123].

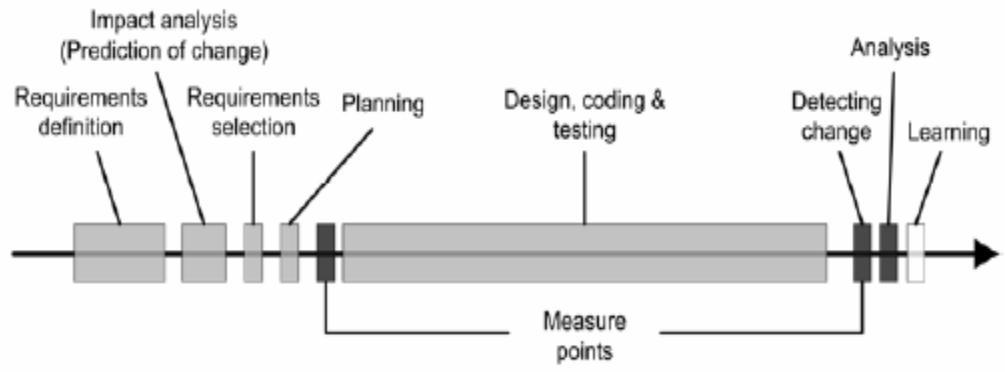

Figure 5.4: Measuring impact using metrics [123].

5.9.1 Metrics for quantifying change impact

Metrics for quantifying change impact are based on the SLOs that are predicted to be changed as an effect of new or changed requirements. In addition, indicators of how severe the change is can be used. Such measures of the predicted impact can be used to estimate the cost of a proposed change or a new requirement. The more requirements and other SLOs that are affected, the more widespread they are and the more complex the proposed change are, the more expensive the new or changed requirement would be. Requirements that are costly in this sense but provide little value can, for example, be filtered out for the benefit of requirements that provide more value but to a smaller cost [109].

Change impact can be measured based on the set of requirements that is affected by the change. For example, the number of requirements affected by a change can be counted based on this set. The affected requirements' complexity often determines how severe the change is and can be measured in various ways. Examples are the size of each requirement in terms of function points and the dependencies of each requirement on other requirements. For other SLOs, the metrics are similar. For architecture and design, measures of impact include the number of affected components, the number of affected classes or modules, and number of affected methods or functions. For source code, low-level items such as affected lines of code can be measured and the level of complexity for components, classes, and methods can be measured using standard metrics such as cyclomatic complexity and regular object-oriented metrics [116, 123].

In determining how severe or costly a change to a code is, it is useful to define the *impact factor*. Lindvall [111] defined the impact factors in Table 5.4 to measure the impact of a suggested change. The impact factor is based on empirical findings in which it was determined that changes to different types of SLOs can be used as an indicator of the extent of the change. The higher the impact factor, the more severe the change. For example, changes that do not affect any other type of SLO but the design object model are relatively limited in scope. Changes that affect the use-case model are instead likely to require changes that are related to the fundamentals of the system and are therefore larger in scope. In addition, changes to the use-case model most likely also involve changes of all other SLOs making this kind of changes even more severe.

Table 5.4: Impact factors [123]

Impact Factor	Impact	Description
M1	Change of the design object model.	These changes regard the real or physical description of the system and may generate change in the software architecture about the size of the change in the model.
M2	Change of the analysis object model.	These changes regard the ideal or logical description of the system. A small change here may generate change in the software architecture larger than the change in this model.
M3	Change the domain object model.	These changes regard the vocabulary needed in the system. A small change here may generate large change in the software architecture.
M4	Change the use-case model.	These changes require additions and deletions to the use-case model. Small changes here may require large change in the software architecture.

5.9.2 Metrics for evaluation of impact analysis

Bohner and Arnold [17] proposed a number of metrics with their introduction of impact sets. These metrics are relations between the cardinalities of the impact sets, and can

be seen as indicators of the effectiveness of the impact analysis approach employed (# denotes the cardinality of the set):

1. #SIS / #EIS, i.e. the number of SLOs initially thought to be affected over the number of SLOs estimated to be affected (primary change and secondary change). A ratio close to 1 is desired, as it indicates that the impact is restricted to the SLOs in SIS. A ratio much less than 1 indicates that many SLOs are targeted for indirect impact, which means that it will be time-consuming to check them.
2. #EIS / #System, i.e. the number of SLOs estimated to be affected over the number of SLOs in the system. The desired ratio is much less than 1, as it indicates that the changes are restricted to a small part of the system. A ratio close to 1 would indicate either a faulty impact analysis approach or a system with extreme ripple effects.
3. #EIS / #AIS, i.e. the number of SLOs estimated to be affected over the number of SLOs actually affected. The desired ratio is 1, as it indicates that the impact was perfectly estimated. In reality, it is likely that the ratio is smaller than 1, indicating that the approach failed to estimate all impacts. Two special cases are if AIS and EIS only partly overlap or do not overlap at all, which also would indicate a failure of the impact analysis approach.

Fasolino and Visaggio [112] also define metrics based on the cardinalities of the impact sets. They tie the metrics to properties and characteristics of the impact analysis approach, as per the tree in Figure 5.5.

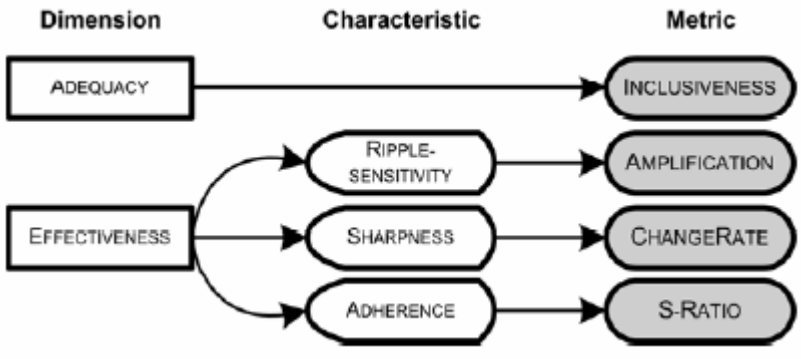

Figure 5.5: Tree of impact analysis metrics [112].

Adequacy is the ability of the impact analysis approach to estimate the impact set. It is measured by means of the binary metric *Inclusiveness*, which is strictly defined to be 1 if all SLOs in AIS are also in EIS and 0 otherwise. *Effectiveness* is the ability of the approach to provide beneficial results. It is refined into *Ripple-sensitivity* (the ability to identify ripple effects), *Sharpness* (the ability not to over-estimate the impact) and *Adherence* (the ability to estimate the correct impact).

Ripple-sensitivity is measured by *Amplification*, which is defined as (#EIS - #SIS) / #SIS, i.e. the ratio between the number of indirectly impacted SLOs and the number of directly impacted SLOs. This ratio should preferably not be much larger than 1, which would indicate much more indirect impact than direct impact. Sharpness is measured by *ChangeRate*, which is defined as #EIS / #System. This is the same metric as the second of Arnold and Bohner's metrics presented previously. Adherence is measured by *S-Ratio*, which is defined as #AIS / #EIS. S-Ratio is the converse of the third of Arnold and Bohner's metrics presented previously.

Lam and Shankararaman [113] propose metrics that are not related to the impact sets. These metrics are more loosely defined and lack consequently recommended values such as:

i. *Quality deviation*, i.e. the difference in some quality attribute (for example, performance) before and after the changes have been implemented, or between actual and simulated values. A larger than expected difference could indicate that the impact analysis approach failed to identify all impact.

ii. *Defect count*, i.e. the number of defects that arise after the changes have been implemented. A large number of defects could indicate that some impact was overlooked by the impact analysis approach.

iii. *Dependency count*, i.e. the number of requirements that depend on a particular requirement. Requirements with high dependency count should be carefully examined when being subjected to change.

Lindvall [124] defined and used metrics in a study at the Swedish telecom company Ericsson AB in order to answer a number of questions related to the result (prediction) of impact analysis as conducted in a commercial software project and performed by the project developers as part of the regular project work. The study was based on impact analysis conducted in the requirements phase, and the term *requirements-driven impact analysis* was coined to capture this fact. The results from the impact analysis were used by the Ericsson project to estimate implementation cost and to select requirements for implementation based on the estimated cost versus perceived benefit. The study first looked at the collected set of requirements' predicted and actual impact by answering the following questions: "How good was the prediction of the change caused by new and changed requirements in terms of predicting the *number* of C++ classes to be changed?" and "How good was this prediction in terms of predicting *which* classes were

to be changed?" The last question was broken down into the two sub questions: "Were changed classes predicted?" and "Were the predicted classes changed?"

There was a total of 136 C++ classes in the software system. 30 of these were predicted to be changed. The analysis of the source code edits showed that 94 classes were actually changed. Thus, only 31.0% (30 / 94) of the number of changed classes were predicted to be changed.

In order to analyze the data further, the classes were divided into the two groups *Predictive group* and *Actual group*. In addition, each group was divided into two subgroups: *Unchanged* and *Changed*. The 136 classes were distributed among these four groups as shown in Table 5.5.

Table 5.5: Predicted vs. actual changes [123].

		Predictive group		
		Unchanged	**Changed**	
Actual Group	**Unchanged**	**A: 42** (30.9%)	**B: 0** (0.0%)	**A+B: 42** (30.9%)
	Changed	**C: 64** (47.1%)	**D: 30** (22.1%)	**C+D: 94** (69.1%)
		A+C: 106 (77.9%)	**B+D: 30** (22.1%)	**N: 136** (100.0%)

Cell A represents the 42 classes that were not predicted to change and that also remained unchanged. The prediction was correct as these classes were predicted to remain unchanged, which also turned out to be true. The prediction was implicit as these classes were indirectly identified – they resulted as a side effect as complement of predicting changed classes.

Cell B represents the zero classes that were predicted to change, but actually remained unchanged. A large number here would indicate a large deviation from the prediction.

Cell C represents the 64 classes that were not predicted to change, but turned out to be changed, after all. As with cell B, a large number in this cell indicates a large deviation from the prediction.

Cell D, finally, represents the 30 classes that were predicted to be changed and were, in fact, changed. This is a correct prediction. A large number in this cell indicates a good prediction.

There are several ways to analyze the goodness of the prediction. One way is to calculate the percentage of correct predictions, which was (42 + 30) / 136 = 52.9%. Thus, the prediction was correct in about half of the cases. Another way is to use Cohen's Kappa value, which measures the agreement between two groups ranging from -1.0 to 1.0. The -1.0 figure means total incompliance between the two groups, 1.0 means total compliance and 0.0 means that the result is no better than pure chance [114]. The kappa value in this case is 0.22, which indicates a fair prediction. We refer to [111] for full details on the Kappa calculations for the example. A third way to evaluate the prediction is to compare the number of classes predicted to be changed with the number of classes actually changed. The number of classes predicted to be changed in this case turned out to be largely under predicted by a factor of 3. Thus, only about one third of the set of changed classes was identified. It is, however, worth noticing that all of the classes that were predicted to be changed were in fact changed.

The study then analyzed the predicted and actual impact of each requirement by answering similar questions for each requirement. The requirements and the classes that were affected by these requirements were organized in the following manner: For each requirement, the set of classes predicted to be changed, the set of changed classes and the intersection of the two sets, i.e. classes that were both predicted and changed. In addition, the sets of classes that were predicted but not changed and the set of classes that were changed but not predicted were identified.

The analysis showed that in almost all cases, there was an under-prediction in terms of number of classes. In summary, the analysis showed that the number of changed classes divided by the number of predicted classes ranged from 1.0 to 7.0. Thus, up to 7 times more classes than predicted were actually changed [115].

Estimating cost in requirements selection is often based on the prediction like it was in the Ericsson case, which means that requirements predicted to cause change in only a few entities are regarded as less expensive, while requirements predicted to cause change in many entities are regarded as more expensive. This makes the rank-order of requirements selection equal to a requirements list sorted by the number of items predicted. By comparing the relative order based on the number of predicted classes with the relative order based on the number of actual changed classes, it was possible to judge the goodness of the prediction from yet another point of view. The analysis on the requirements level showed that a majority of the requirements were under-predicted. It was also clear that it is relatively common that some classes predicted for one requirement are not changed because of this particular requirement, but because of some other requirement [110, 115]. This is probably because the developers were

not required to implement the changed requirements exactly as was specified in the implementation proposal resulting from the impact analysis. The analysis of the order of requirements based on the number of predicted classes showed that the order was not kept entirely intact. Some requirements that were predicted to be small proved to have a large change impact, and vice versa.

In order to try to understand the requirements-driven impact analysis process and how to improve it, an analysis of the various characteristics of changed and unchanged classes was undertaken. One such characteristic was size, and the questions were: “Were large classes changed?”; “Were large classes predicted?” and “Were large classes predicted compared to changed classes?”

The analysis indicated that large classes were changed, while small classes remained unchanged. The analysis also indicated that large classes were predicted to change, which leads to the conclusion that class size may be one of the ingredients used by developers, maybe unconsciously, when searching for candidates for a new or changed requirement [110, 115].

5.10 Chapter Summary

Metrics are useful and important in impact analysis for various reasons. Metrics can, for example, be used to measure and quantify change caused by a new or changed requirement at the point of the impact analysis activity. They can also be used to evaluate the impact analysis process itself once the changes have been implemented. In determining how severe or costly a change is, it is useful to determine the impact or the impact factor as it indicates the likely extent of a change to a certain type of SLO.

In this chapter, we have proposed two formal models as metrics for supporting our framework to monitor or control change propagation and determine level of fault propagation and consequently the impact of change in a typical grid environment. The chapter has also proposed the need for OO metrics extension for use in developing the evolving AO platform for service provisioning.

In conclusion, impact analysis is a crucial activity supporting requirements engineering and service orientation. The results from impact analysis feed into many activities including estimation of requirements' cost and prioritizing of requirements. These activities feed directly into project planning, making impact analysis a central activity in a successful project.

CHAPTER 6

SUMMARY, CONCLUSIONS AND FURTHER WORK

6.1 Summary

Our motivational goal for this research was to evolve a change impact analysis model-based framework for validating, analyzing and monitoring change propagation in a typical grid service provisioning environment given our GUISET research focus as a typical grid service provisioning environment.

In line with this goal and to address the research question (MRQ1) in section 1.3, we develop a change propagation measurement framework as an approach to assist the system maintainer to track changes and determine apriori the need for change performance. In using the framework developed in this research to identify potential impacts before making a change, it is possible to minimally reduce the risks of embarking on a costly change because the later the problem is identified the more its cost of solving it. The framework can provide visibility into the potential effects of changes before changes are implemented, and identify the consequences or propagation effects of proposed system changes. On this basis, it can help maintainers to plan changes, make changes more accurately, accommodate certain types of changes, trace through the effects of changes. The framework can also be used to evaluate the appropriateness of a proposed modification. Assuming a proposed change has the propensity of impacting large, disjoint sections of a service, the change might need re-examination to ascertain whether a safer change is possible.

Managers can use this framework to run “what if” analysis on different change proposals, and choose the one that is most cost effective. Service maintainers can also use this framework to indicate the vulnerability of critical sections of the service functionality during utility service provisioning. If a service functionality is dependent on a different part of a service, its functionality is susceptible to changes made in these parts. The maintainers can use it to locate the areas that are impacted by the changes, enabling them to focus particularly on those areas and still feel confident about the quality of the service provided.

During the course of this research, we proposed a review of Object Oriented Metrics for the new Aspect Oriented Paradigm [117]. This became paramount as new service platforms are now being developed using AOP and serves as enhancement for our model metrics development.

Also, to address the second research (MRQ2), we crafted two formal models to serve as metrics that are expected to work along side the framework and help maintainers to quantitatively measure the system in regards to its susceptibility to change. Measurement of change effect has been incorporated into several service maintenance models to give maintainers valuable information about the system they are maintaining. Maintenance is difficult because it is not clear where modifications have to be made or what the impact will be on the rest of the service once those changes are made. The framework and associated models can be used to help maintainers with assessing that impact. Along with many other metrics, the model metrics is not the answer to all maintainer's problems, but used as part of a suite of metrics that can give maintainers useful information to make their task easier. The framework is not only useful during

service maintenance, but can also be used to ascertain whether service stability is increasing or decreasing and changes made accordingly.

6.2 Concluding Remarks

Our automatic evaluation of change impact analysis has been at some cost, since we have not practically utilized the idea in an existing grid environment as our GUISET research is still on-going. But the idea behind the research is vividly clear, although requiring little human effort as the part of identifying, monitoring and calculating the change is a human activity.

Impact analysis is a crucial and salient part of requirements engineering since service changes are often initiated by changes due to fault and failure of service or the need for preventive maintenance. The need for service providers to determine what to change in order to implement changes has always been present. Classical methods and strategies used in conducting impact analysis were dependency analysis, traceability analysis and slicing. Earlier work on impact analysis was focused on applying such methods and strategies onto source code in order to conduct program slicing and determine effects of code propagation for code changes. As software engineering discipline matures, the need to understand how change requests affect other SLOs than source code became imperative. Today's typical strategies and methods are based on analyzing traceability or dependency information, utilizing slicing techniques, consulting design specifications and other documentation, and interviewing knowledgeable developers. Interviewing knowledgeable developers is probably the most common way to acquire information about likely effects of new or changed requirements.

Metrics are useful and important in impact analysis for various reasons. Metrics can, for example, be used to measure and quantify change caused by a new or changed requirement at the point of the impact analysis activity. They are also useful to evaluate the impact analysis process itself once the changes have been implemented. In determining how severe or costly a change is, it is useful to determine the impact factor as it indicates the likely extent of a change to a certain type of SLO.

The results obtained from impact analysis are beneficial in many activities such as cost requirement estimation and prioritizing of requirements. These activities feed directly into project planning, making impact analysis a central activity in a successful project.

For this research, we have evolved a framework for monitoring service change propagation in our GUISET environment. The framework is to assist service maintainers to track changes and the affected dependencies.

We have also developed two relevant models as metrics, to be use alongside the framework, to validate and analyze the effects of service change propagation.

We measured change impact based on the set of services that are affected by a change. Therefore, our concentration was on the number of services that are affected by the change and their dependencies to describe the level of fault propagation. This is why we extracted only services that have a fault and their dependencies, and expressed this in Table 5.2 and consequently determined the characteristics of these dependencies and the corresponding fault propagation graphically as shown in figure 5.2. The general understanding obtained from this graph is that, the higher the dependencies, the higher the rate of fault propagated, and the greater the number of changes required to keeping the main service in a consistent state. The affected services' complexity often

determines how severe the change was. The higher the number of service dependencies, the higher the level of complexity and invariably the more severe the change.

6.3 Research Limitation and Future Work

Impact analysis work is very costly, as it is difficult to find resources for performing impact analysis. In an improvement effort, the research work presented here did not consider using software codes implementation to validate its approach. This is because previous research on the main subject has concentrated effort on code design impact analysis. But in other for this work to contribute to this field, we have moved our study to services which are being consumed from software code. Therefore, this work is limited to utility service provisioning environments.

Although we acknowledge the fact that the problem at hand may be viewed as an NP-hard optimization problem, for which one way solution approach is usually through the use of heuristics, this research did not consider the use of heuristics as an approach to solving the research questions. This is because, it is necessary to exploit multiple problem solvers in hard combinatorial domains, since large-scale combinatorial optimization problems pose difficult challenges and the algorithms guaranteed to find optimal solution are often too costly. Again, because our approach is model-based, we do not consider using the traditional approach of applying existing NP hard algorithms for Change propagation.

The most imminent limitation in this research is the implementation and evaluation of this idea in a large-scale industrial setting. Our Grid environment is yet to

mature for industrial usage; therefore, we are currently limited to a wider application of the proof of concept.

For the framework and its associated models to serve as a fully working measurement tool for the industry, further work needs to be carried out.

Until now automatic computation of change impact analysis factor or value has proved troublesome. We aim to make the framework in our GUISET research group robust, so that it can be used extensively in industry to compute the effect of service change in a grid environment. The foundations have been built with our work described in this thesis and our task is now to see at GUISET research implementation, the usability enhancement of our framework.

Another aspect of this research that needs to be refined is that of choosing the right constants for the metrics developed. A major undertaking would be to validate these metrics, and explore how best to apply them in a real industrial life situation.

Additionally, we observed that, syntactic impact is measured purely by information obtained from the framework through causal relationship between services, data flow and control flow. The research did not consider the semantic impact investigation. This is because semantic knowledge includes domain knowledge and system program run time knowledge. Semantic knowledge is extremely difficult to derive and verify compared to syntactic information. Against this backdrop, further research is needed in analysing change impact from the semantic perspective.

More so, the research has included a proposal for the extension of OOP metrics to AOP. This argument is predicated on adopting a set of complexity metrics defined in terms of

program dependence relationships for measuring the complexity of aspect oriented programs. We view that, research work on this proposal requires further development.

A wider application of this research to enhance change propagation prediction is also a vital future interest. For this future work, it would be interesting to explore statistical inferential approaches (e.g. the Bayesian statistics, the Regression method, the Over Relaxation Method and the Neural Networks) to enhance future prediction of the need for change effect determination.

Bibliography

[1] Mockus, A. and Votta, L.: Identifying Reasons for Software Changes Using Historic Databases. Proceedings of the International Conference on Software Maintenance, 11-14 October, 2000. San Jose, USA, pp. 120-130

[2] Leffingwell, D. and Wildrig, D.: Managing Software Requirement – A Unified Approach, Addison Wesley. 1999

[3] Lehman, M., Ramil, J., Wernick, P. Perry, D. and Turski, W.: Metrics and Laws of Software Evolution – The Nineties View. Proceedings of the 4th International Software Metrics Symposium, 5-7 November, 1997. Albuquerque, USA, pp 20-32.

[4] Maciaszek, L.: Requirement Analysis and System Design – Developing Information System with UML, Addison Wesley. 2001

[5] Bohner, S. A. and Arnold, R. S.: An Introduction to Software Change Impact Analysis – Software Change Impact Analysis, IEEE Computer Society Press 1996.

[6] Jonsson, P.: Exploring Process Aspects of Change Impact Analysis. Blekinge Institute of Technology Doctoral Dissertation Series No. 2007:13. ISSN 1653-2090. Sweden, 2007.

[7] Chapin, N.: Do we know what preventive maintenance is? *Proceedings of the 16th International Conference on Software Maintenance*, pp. 15–17. San José, CA, USA, 2000.

[8] Hass, A. M. J.: *Configuration Management Principles and Practice*. Boston, MA, USA: Addison-Wesley Professional, 2002.

[9] Godfrey L. W. and Lee, E. H. S.: Secrets from the Monster - Extracting Mozilla's Soft-ware Architecture, Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools, Limerick, Ireland, pp 15-23

[10] Wiegers, K. E.: *Software Requirement*, Microsoft Press, 2003.

[11] Weinberg, G.M.: *Kill that Code*, Infosystems, 1983. 30: 48-49

[12] Loyall, J. P. and Mathisen, S. A.: Using Dependence Analysis to Support the Software Maintenance Process. Proceedings of International Conference on Software Maintenance, 1993.

[13] Ekabua, O. O., Olugbara, O. O. and Adigun, M. O.: A Generic Change Propagation Framework to Enhance Service Provisioning in a Grid Environment. Asian Journal of Information Technology, 6(10): 1015-1019, 2007. ISSN: 1682-3915.

[14] Hao, H: *What is Service-Oriented Architecture*. CTO of SoftTouch Information Technology Pty. Sept 2003. webservices.xml.com

[15] David, S. and Lawrence, W.: *Understanding Service-Oriented Architecture*. .NET Architecture Centre. Microsoft Architect Journal, January 2004.

[16] Luciano, B., Reiko, H., Sebastian, T. and Daniel, V.: *Modeling and Validation of Service-Oriented Architecture: Application vs. Style.* FSEC/FSE, 2003. Sept. 1-5, 2003. Helsinki. Finland.

[17] Bohner, S. A. and Arnold, R. S. *Software Change Impact Analysis*. IEEE Computer Society Tutorial, IEEE Computer Society Press, 1996.

[18] Davis, A. *Software Requirements: Analysis and Specification*. Prentice-Hall, New Jersey, 1989.

[19] Boehm, B. *Improving Software Productivity*. IEEE Computer, September 1987, Pages 43-57.

[20] Bohner, S. A. *Impact Analysis in the Software Change Process: A Year 2000 Perspective.* In Proceedings International Conference on Software Maintenance ICSM'96, pages 42-51. IEEE Computer Society Press, November 1996.

[21] Arnold, R. S. and Bohner, S. A.: *An Introduction to Software Change Impact Analysis, Software Change Impact Analysis*, IEEE Computer Society Press 1996.

[22] Rajlich, V.: MSE – A Methodology for Software Evolution. Journal of Software Maintenance, Vol. 9, 1997

[23] Kabaili, H., Keller, R. K. and Lustman, R. A.: Change Impact Model Encompassing Ripple Effect and Regression Testing. In Proceedings of the 5th International Workshop on Quantitative Approaches in Object-Oriented Software Engineering, Budapest, Hungary, 2001

[24] Bauer, A. and Pizka, M.: The Contribution of free Software to Software Evolution. Proceedings of IEEE International Workshop on Principles of Software Evolution (IWPSE'03), Helsinki, Finland, Sept. 2003

[25] Xiao,H., Guo, J. And Zou, Y.: Supporting Change Impact Analysis for Service Oriented Business Applications. Proceedings of IEEE International Workshop on Systems

Development in Service-Oriented Architecture Environments, SDSOA'07: ICSE Workshop, May 2007. pp 6-11

[26] Li, W. and Henry, S. *An Empirical Study of Maintenance Activities in Two Object-oriented Systems*. Journal of Software Maintenance, Research and Practice, Volume 7, No. 2 March-April 1995, pp.131-147.

[27] Elish, M. O. and Rine, D. *Investigation of Metrics for Object Oriented Design Logical Stability*. In Proceedings of the Seventh European Conference on Software Maintenance and Reengineering. 26-28 March 2003, pp.193-200.

[28] Haney, F. M.: Module Connection Analysis - A Tool for Scheduling Software Debugging Activities, Proceedings of AFIPS Joint Computer Conference, 1972.

[29] Weiser, M.: Program Slices: Formal, Psychological, and Practical Investigations of An Automatic Program Abstraction Method, Ph.D. thesis, University of Michigan, Michigan, USA, 1979.

[30] ANSI/IEEE Std 830-1984 IEEE Guide to Software Requirements Specifications, Institute of the Electrical and Electronics Engineers, 1984.

[31] Yau S. S. and Collofello, J. S.: Some Stability Measures for Software Maintenance, IEEE Transactions on Software Engineering, 1980.

[32] Turver, R. J. and Munro, M.: An Early Impact Analysis Technique for Software Maintenance, Journal of Software Maintenance Research and Practice, 1994.

[33] Bohner, S. A. and Arnold, R. S.: Software Change Impact Analysis, IEEE Computer Society Press, 1996.

[34] Bohner, S. A.: Extending Software Change Impact Analysis into COTS Components. Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop, Greenbelt, USA, December 4-6, 2002.

[35] Bohner, S. A. and Gracanin, D.: Software Impact Analysis in a Virtual Environment. Proceedings of the 28th Annual NASA Goddard Software Engineering Workshop, December 2-4, Greenbelt, USA, 2003.

[36] Bilal, H. Z. and Black, S. E.: Computing Ripple Effect for Object Oriented Software, Proceedings 10th ECOOP Conference. QAOOSE Workshop. Nantes, France. 3rd July 2006.

[37] Wiegers, K. E.: Software Requirements, Microsoft Press, 2003.

[38] Weinberg, G. M.: Kill That Code, Infosystems, 1983.

[39] Lindvall, M. and Sandahl, K.: How well do Experienced Software Developers Predict Software Change?, Journal of Systems and Software, 1998.

[40] Eick, S. G., Graves, L., Karr, A. F. and Marron, J. S.: Does code decay? Assessing the Evidence from Change Management Data. IEEE Transactions on Software Engineering, 2001.

[41] Kotonya, G. and Sommerville, I.: Requirement Engineering – Processes and Techniques, Wiley and Sons. 1998.

[42] Leffingwell, D. and Wildrig, D.: Managing Software Requirement – A Unified Approach, Addison Wesley. 1999.

[43] Maciaszek, L.: Requirements Analysis and System Design - Developing Information Systems with UML, Addison Wesley, 2001.

[44] Robertson, S. and Robertson, J.: Mastering the Requirements Process, Addison Wesley, 1999

[45] Sommerville I, Sawyer P.: Requirements Engineering - A Good Practice Guide, John Wiley and Sons, 1997

[46] Arnold, R. S. and Bohner, S. A.: Impact Analysis – Towards a Framework for Comparison. Proceedings of the conference on Software Maintenance, Los Alamitos, CA, September 1993, pp 292-301.

[47] Pfleeger, S. and Bohner, S.: A Framework for Software Maintenance Metrics. IEEE Transactions on Software Engineering May 1990. pp 320-327

[48] Turver, R. J. and Munro, J.: An Early Impact Analysis Technique for Software Maintenance. *Software Maintenance: Research and Practice*, 6:35{52, 1994.

[49] Bilal, H. Z. and Black, S. E.: Using the Ripple Effect to Measure Software Quality. Proceedings of Software Quality Management (SQM'05), Cheltenham, Gloucestershire, UK. March 21-23, 2005.

[50] Black, S. E. and Rosner, P. E.: Measuring Ripple Effect for the Object Oriented Paradigm. Proceedings of IASTED International Conference on Software Engineering, Innsbruck, Austria, February 15th – 17th, 2005.

[51] Harrod, M. J. and Malloy, B.: A Unified Interprocedural Program Representation for a Maintenance Environment. IEEE Transactions on Software Engineering, Volume 10, No. 6. June 1993, pp 583-593.

[52] Lee, M. I.: Change Impact Analysis of Object Oriented Software. Technical Report ISE-TR-99-06, George Mason University, 1998.

[53] IEEE Standard 610-12[729]: Software Engineering Terminology. Published by the Institute of Electrical and Electronics Engineering, Inc. 345 East 47th Street, New York, NY 10017-2349, USA, 1990.

[54] Pressman, R. S.: *Software Engineering: A Practitioner's Approach, (3rd Ed.)*. European Adaptation, McGraw-Hill International (UK) Ltd., Maidenhead, UK, 1994.

[55] Moreton, R.: A Process Model for Software Maintenance. Journal of Information Technology, Volume 5, 1990. pp 100-104.

[56] Hoffmann, M., Kühn, N. and Bittner, M.: Requirements for Requirements Management Tools. Proceedings of the 12th IEEE International Requirements Engineering Conference, Kyoto, Japan, September 6-10, 2004

[57] Natt och Dag, J., Regnell, B., Carlshamre, P., Andersson, M. and Karlsson, J.: A Feasibility Study of Automated Support for Similarity Analysis of Natural Language Requirements in Market-Driven Development, Requirements Engineering 2002. 7:20-33

[58] Lee, M., Offutt, J. A. and Alexander, R. T.: Algorithmic Analysis of the Impacts of Changes to Object-Oriented Software. Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems, Santa Barbara, USA, July 30-Aug 4, 2000

[59] Adigun, M. O., Emuoyibofarhe, O. J. and Migiro, S. O.: Challenges to Access and Opportunity to use SMME enabling Technologies in Africa. A presentation at 1st all Africa Technology Diffusion Conference. June 14-16, Johannesburg, South Africa.

[60] Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., and Orchard,D.: Web Services Architecture. W3C, Working Draft http://www.w3.org/TR/2003/WD-ws-arch-20030808/, 2003.

[61] Foster, I., Kesselman, C., and Tuecke, S.: The Anatomy of the Grid: Enabling Scalable Virtual Organisations. International Journal of Supercomputer Applications. 15(3). 200-222. 2001.

[62] Foster, I., Kesselman, C., and Jennings, N.: Brain Meets Brawn: Why Grid and Agents Need Each Other. Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagents Systems, Pages 8-15, 2004.

[63] Foster, I., Kesselman, C., Nick, J.M. and Tuecke, S.: Grid Services for Distributed Systems Integration. IEEE Computer 35(6). 37-46. 2002.

[64] Adigun, M. O., Emuoyibofarhe, O. J. and Migiro, S. O.: Challenges to Access and Opportunity to use SMME enabling Technologies in Africa. A presentation at 1st all Africa Technology Diffusion Conference. June 14-16, Johannesburg, South Africa.

[65] Medvidovic, N et al,: Software Architectural Support for Handheld Computing. IEEE Computer, 2003

[66] Lee, E. A.: Embedded Software" Advances in Computers, M Zelkowitz, ed., Vol.56, Academic Press. 2002

[67] Fiege, L et al.: Publish-Subscribe Grows Up-Support for Management, Visibility Control and Heterogeneity," IEEE Internet Computing, 2006

[68] Daramola, W., Adigun, M. O. and Olugbara, O. O.: A Generic Architecture for Tourism Product-Line. In Proceedings of SAICSIT Conference, 2006.

[69] Zuma, S.M. and Adigun, M. O.: CACIP: A Pattern for Interfacing Components in a Context-Aware Mobile Environment. Proceedings of IASTED International Conference on Simulation and Modelling, Delta Hotel, Montreal, Quebec, Canada, May24-26, 2006.

[70] Erl, T.: Service-Oriented Architecture – Concepts, Technology, and Design. Pearson Education Inc., Prentice Hall Professional Technical Reference. 6th Edition, 2006. ISBN: 0-13-185858-0.

[71] IEEE Std 830-1998: IEEE Recommended Practice for Software Requirements Specifications. Technical Report IEEE 830-1998, IEEE Computer Society, Los Alamitos, CA, USA. 1998

[72] CMMI for Development, version 1.2. Tech. Rep. CMU/SEI-2006-TR-008, Software Engineering Institute 2006

[73] Sommerville, I.: *Software Engineering*. Boston, MA, USA: Addison Wesley, 7th ed. 2004

[74] Ruhe, G. and Saliu, M. O. (2005): The art and science of software release planning. *IEEE Software 2005*. ISSN 0740-7459.

[75] Humphrey, W. S.: Three process perspectives: organizations, teams, and people. *Annals of Software Engineering, 2002*

[76] Cockburn, A.: *Agile Software Development*. Boston, MA, USA: Pearson Education, Inc. 2002

[77] Larman, C. and Basili, V. R.: Iterative and Incremental Development: a Brief History. *Computer Society, 2003*.

[78] Pfleeger, S. L.: *Software engineering: theory and practice*. Upper Saddle River, NJ, USA: Prentice Hall, international ed. 1998

[79] Potts, C.: Software-engineering research revisited. *IEEE Software, 1993*.

[80] Rajlich, V.: A Model for Change Propagation Based on Graph Rewriting. Proceedings of ICSM'97, Bari, Italy, September 28th to October 2nd, 1997.

[81] Yau, S. S., Nicholl, R.A., Tsai, J. J. and Liu, S.: An Integrated Life-Cycle Model for Software Maintenance. IEEE Transactions for Software Engineering, 1998.

[82] Erckert, C., Clarkson, P. J. and Zanker, W.: Customization in Complex Engineering Domains. University of Cambridge Technical Report CUED/EH101/6/99. 2000

[83] Luqi, A.: A Graph Model for Software Evolution. IEEE Transaction on Software Engineering, 1990

[84] Clarkson, P. J., Simons, C. and Eckert, C.: Predicting Change Propagation in Complex Design. Proceedings of DETC'01 ASME 2001 Design Engineering Technical Conferences and Computer and Information in Engineering Conference, Pittsburgh, Pennsylvania, September 9 – 12, 2001

[85] Harrison, W., Ossher, H. and Tarr, P.: Software Engineering Tools and Environments: A Roadmap 2000. Future of Software Engineering, Limerick, Ireland. 2000.

[86] Brereton, P. and Budgen, D.: Component-Based Systems – A Classification of Issues. IEEE Computer, Vol. 33, No. 11. 2000

[87] Sun, C.: Empirical Reasoning about Quality of Service of Component-Based Distributed Systems. ACMSE'04, Huntsville, Alabama, 2004

[88] David, S. and Lawrence, W.: *Understanding Service-Oriented Architecture*. .NET Architecture Centre. Microsoft Architect Journal, January 2004.

[89] Hao, H: *What is Service-Oriented Architecture*. CTO of SoftTouch Information Technology Pty. Sept 2003. webservices.xml.com

[90] Gold, N. and Mohan, A.: A Framework for Understanding Conceptual Changes in Evolving Source Code. Proceedings of the IEEE International Conference on Software Maintenance (ICSM'03), 2003

[91] Hassan, A. E. and Holt, R. C.: Predicting Change Propagation in Software Systems. Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04), 2004

[92] Gallagher, K. B. and Lyle, J. R.: Using program slicing in software maintenance. IEEE Transactions on software maintenance, 1991

[93] Lee, E. H. S.: Software Comprehension across Levels of Abstraction. Master's thesis, University of Waterloo, 2000

[94] Finnigan, P.J., Holt, R.C., Kalas, I., Kerr, S., Kontongiannis, H. A., Muller, J., Mylopoulos, S. G., Perelgut, M., Stanley, M. and Wong, K.: The Software Bookshelf. IBM Systems Journal, 1997

[95] Cleland-Huang, J., Chang, C. K. and Wise, J. C.: Automating Performance-Related Impact Analysis Through Event Based Traceability. Requirements Engineering, 2003 8(3):171-182

[96] Rutka, Guenov, Lemmens, Schmidt-Schaffer, Coleman, Riviere: Methods for Engineering Change Propagation Analysis. In Proceedings of 25th International Congress of the Aeronautical Sciences (ICAS'06), 2006

[97] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J.: Aspect Oriented Programming. Proceedings of 11th ECOOP. LNCS, Vol. 1241, Springer-Verlag,June 1997

[98] Tar, T., Ossher, H., Harrison, W. and Sutton, S.: N degrees of Separation: Multi-DimensionalSeparation of Concerns, Proc. the 21st International Conference on Software Engineering, 1999

[99] Zhao, J.: "Towards a Metric Suite for Aspect-Oriented Software.Technical Report SE-136- 25,Information Processing Society of Japan (IPSJ), March 2002

[100] Xenos, M., Stavrinoudis, D., Zikouli, K., Christodoulakis, D.: Object-Oriented Metrics – A Survey. Proc. of the FESMA 2000, Mildrid, Spain, 2000

[101] Basili, V. R. and Reiter, Jr. R. W.: Evaluating Automable Measures of Software Development. *IEEE Transactions on Software Engineering*, 1979.

[102] Chidamber, S. and Kemerer, C.: A Metrics Suite for Object Oriented Design. IEEE transactions on Software Engineering. Vol. 20, No 6. June 1994

[103] Clark, S., Baniassad, E.: Aspect-Oriented Analysis and Design – The Theme Approach. Addison Wesley, march, 2005

[104] Karl L, Doug O. Johan O.: Aspect Oriented Programming with Adaptive Methods Communications of the ACM, Vol. 44, No. 10. October 2001

[105] John, V., Jeffrey, V.: Can Aspect Oriented Programming Lead to More Reliable Software?. IEEE Software Society, Nov/Dec 2000

[106] Henry, S., Kafura, D.: Software Structure Metrics Based on Information flow in IEEE Transaction on Software Engineering, Vol. SE-7: 1981

[107] David, A.: Properties of Software Measures. Proceedings of BCS-FACS workshop on Formal Aspect of Measurement, South Bank University, London, May 1991

[108] Zakaria, A. Hosny, H.: Metrics for Aspect-Oriented Software Design". AOM: Aspect Oriented Modeling with UML, AOSD, 2003

[109] Jönsson, P. and Wohlin, C.: Understanding impact analysis: an empirical study to capture knowledge on different organisational levels. In Proceedings of the International Conference on Software Engineering and Knowledge Engineering, Taipei, Taiwan, 2005

[110] Jönsson, P. and Wohlin, C.: A study on prioritisation of impact analysis issues: A comparison between perspectives. In *Proceedings of the 5th Conference on Software Engineering and Research in Sweden*, Västerås, Sweden, 2005

[111] Lindvall, M. and Sandahl, K.: How well do Experienced Software Developers Predict Software Change?, Journal of Systems and Software, 1998

[112] Fasolino, A. R. and Visaggio, G.: Improving Software Comprehension through an Automated Dependency Tracer, Proceedings of the 7th International Workshop on Program Comprehension, Pittsburgh, USA, May 5-7, 1999

[113] Lam, W. and Shankaraman, V.: Requirements Change: A Dissection of Management Issues. Proceedings of the 25th EuroMicro Conference, Milan, Italy, September 8-10, 1999

[114] Cohen, J.: A Coefficient of Agreement for Nominal Scales, Educational and Psychological Measurement, 1960

[115] Jönsson, P. and Wohlin, C.: Using Checklists to Support the Change Control Process - A Case Study. In *Proceedings of the 6th Conference on Software Engineering and Research in Sweden*, Umeå, Sweden, 2006

[116] Ekabua, O. O. and Adigun, M. O.: A Framework and Associated Models for Determining Change Impact Analysis During Utility Service Provisioning in a Grid Environment. In Proceedings of International Conference on Software Engineering

Research and Practice (SERP'08), WorldComp'08, Las Vegas, Nevada, USA. July 14 – 17, 2008.

[117] Ekabua, O. O. and Adigun, M. O.: Reviewing Object Oriented Metrics for Aspect Oriented Paradigm. Proceedings of International Conference on Software Engineering Research and Practice (SERP'07), WorldComp'07, Las Vegas, Nevada, July, 2007.

[118] Pop, P. C. and Zelina, I.: Heuristics Algorithm for the Generalized Minmum Spanning Tree Problem. Proceedings of the International Conference on Theory and Applications of Mathematics and Informatics – ICTAMI, Thessaloniki, Greece, pages 385-395, 2004.

[119] Kasperski, A. and Zielinski, P.: An Approximation Algorithm for Interval Data Minmax Regret Combinatorial Optimization Problems. Information Processing Letters, 97: 171-180, 2006.

[120] Zhou, Y., He, J. and Nie, Q.: A Comparative Runtime Analysis of Heuristics Algorithm for Satisfiability Problems. Artificial Intelligence. Vol. 173, Issue 2, pages 240-257, 2009.

[121] Stamelos, I., Angelis, L., Dimou, P. and Sakellaris, E.: On the Use of Bayesian Belief Networks for the Prediction of Software Productivity. Information and Software Technology, 45: 51-60, 2003.

[122] Stewart, B.: Predicting Project Delivery rates Using the Naive-Bayes Classifier. Journal of Software Maintenance and Evolution. Reserach and Practice, 14: 161-179, 2002.

[123] Jonsson, P. and Lindvall, M.: Engineering and Managing Software Requirements. Published in C. Wohlin and A. Aurum (Eds.), Chapter 6, Springer-Verlag, 2005

[124] Lindvall, M.: An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Systems Evolution, Ph.D. thesis no. 480, Linköping Studies in Science and Technology, Sweden, 1997