ABSTRACT

QUANTIFYING SOFTWARE ARCHITECTURE ATTRIBUTES

by
Bo Yu

Software architecture holds the promise of advancing the state of the art in software engineering. The architecture is emerging as the focal point of many modern reuse/evolutionary paradigms, such as Product Line Engineering, Component Based Software Engineering, and COTS-based software development.

The author focuses his research work on characterizing some properties of a software architecture. He tries to use software metrics to represent the error propagation probabilities, change propagation probabilities, and requirements change propagation probabilities of a software architecture. Error propagation probability reflects the probability that an error that arises in one component of the architecture will propagate to other components of the architecture at run-time. Change propagation probability reflects, for a given pair of components *A* and *B*, the probability that if *A* is changed in a corrective/perfective maintenance operation, *B* has to be changed to maintain the overall function the system. Requirements change propagation probability reflects the likelihood that a requirement change that arises in one component of the architecture propagates to other components. For each case, the author presents the analytical formulas which mainly based on statistical theory and empirical studies. Then the author studies the correlations between analytical results and empirical results.

The author also uses several metrics to quantify the properties of a Product Line Architecture, such as scoping, variability, commonality, and applicability. He presents his proposed means to measure the properties and the results of the case studies.

QUANTIFYING SOFTWARE ARCHITECTURE ATTRIBUTES

by
Bo Yu

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

January 2006

APPROVAL PAGE

QUANTIFYING SOFTWARE ARCHITECTURE ATTRIBUTES

Bo Yu

Dr. Ali Mili, Dissertation Advisor Professor of Computer Science, NJIT	Date
Dr. Joseph Leung, Committee Member Distinguished Professor of Computer Science, NJIT	Date
Dr. Frank Shih, Committee Member Professor of Computer Science, NJIT	Date
Dr. Qun Ma, Committee Member Assistant Professor of Computer Science, NJIT	Date
Dr. Robb Klashner, Committee Member Assistant Professor of Information Systems, NJIT	Date

BIOGRAPHICAL SKETCH

Author: Bo Yu

Degree: Doctor of Philosophy

Date: January 2006

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2006

- Master of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2001

- Bachelor of Science in Chemistry,
  Shandong University, Jinan, P. R. China, 1987

Major: Computer Science

Presentations and Publications:

Bo Yu and Ali Mili,
"Estimating Requirements Change Propagation from Software Architecture", WSEAS Transactions on Computer, Issue 5, Volume 3, November 2004, page: 1674.

Bo Yu and Ali Mili,
"Assessing and Quantifying Attributes of Product Line Architectures", International Conference on Computing, Communications and Control Technologies: CCCT'04, August 2004, Austin, Texas.

W. Abdelmoez, D. M. Nassar, M. Shereshevsky, N. Gradetsky, R. Gunnalan, H. H. Ammar, B. Yu, A. Mili,
"Error Propagation In Software Architectures", 10th International Symposium on Software Metrics (METRICS'04), 09 11 - 09, 2004 Chicago, Illinois, pages: 384-393.

W. Abdelmoez, M. Shereshevsky, R. Gunnalan, H. H. Ammar, B. Yu, S. Bogazzi, M. Korkmaz, A. Mili,

"Quantifying Software Architectures: An Analysis of Change Propagation Probabilities", ACS/IEEE International Conference on Computer Systems and Applications, Cairo, Egypt, 3-6 January, 2005

W. Abdelmoez, M. Shereshevsky, R. Gunnalan, H. H. Ammar, B. Yu, S. Bogazzi, M. Korkmaz, A. Mili,

"Software Architectures Change Propagation Tool (SACPT)", Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04), 09 11 - 09, 2004 Chicago, Illinois, page: 517

To my wife, Dr. Yongxia Wang, without her support this dissertation would have been impossible.

The author also wants to dedicate this dissertation to his daughter, Catherine Yu, whose birth bettered his life in every aspect.

ACKNOWLEDGMENT

The author would like to express his utmost gratefulness to Dr. Ali Mili for all the help he got from Dr. Mili over the years, and the good working environment Dr. Mili provided. Dr. Mili's support and patience are sincerely appreciated. The author also wants to express his deep appreciation to Dr. Hany Ammar (from West Virginia University) for his help and all the discussions.

The author expresses his heartfelt gratitude to all the committee members, Dr. Joseph Leung, Dr. Frank Shih, Dr. Qun Ma, and Dr. Robb Klashner, for their participation and the time spent on this dissertation.

The author is also in debt to Mr. Sergio Bogazzi who made big contributions in building the automatic tools.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES
(Continued)

PART I BACKGROUND

CHAPTER 1

SOFTWARE ARCHITECTURE

The foundation of software architecture was laid in the 1960s through the 1980s [89-91]. Software architecture did not get popular until 1990s [92]. It is not because of some masters advocate the concepts but because of the necessity. Software systems were growing larger and larger: systems of millions of lines of code are not rare anymore. The large systems were getting very difficult to develop, to manage and to understand. Then there came the software architecture to rescue.

There are three main reasons why software architecture is important to large, complex software system [1]:

- It is a vehicle for communication among stakeholders
- It is the manifestation of the earliest design decisions
- It is a reusable, transferable abstraction of a system

It can be seen that software architecture forms the backbone of building large, complex software systems.

1.1 Software Architecture Definition

In this section, the author tries to provide a definition to software architecture. There are many definitions but one is more popular than the others. The most popular one is given by Bass et al. [4] which states as: *"The Software architecture of a program or computing system is the structure or structures of the system, which comprise software components,*

the externally visible properties of those components, and the relationships between them".

It is common to distinguish between five broad classes of architectures, called *architectural styles,* where each style is defined/characterized by: component types; communication patterns/protocols between the components; semantic constraints; and a vocabulary of connectors. The five architectural styles are:

- *Independent Components.* In this style, an architecture is an aggregate of independent processes/objects that communicate through data or control messages.
- *Virtual Machines.* In this style, an architecture is an aggregate of virtual machines arranged in layers, where each layer invokes the layer below it and provides the vocabulary to define the layer above it.
- *Dataflow Architectures.* In this style, an architecture is an aggregate of processing nodes whose activation is driven by the flow of data streams.
- *Data Centered Architectures.* In this style, an architecture is an aggregate of interacting components that communicate through a shared data repository.
- *Call and Return Architectures.* In this style, an architecture is an aggregate of components that are defined in programming terms (procedures, functions routines) and whose interactions are restricted to programming language supported interactions (call and return, parameter passing, etc).

Perhaps with some loss of generality, the author focuses his attention in this study on the first architectural style, i.e. independent components. This style is characterized by its relative genericity: its topology is an arbitrary graph; its messages can be data or control signals; and its interactions can take place through a central message medium, or through one-to-one links. This style is very close to the definition given by Software Engineering Institute (SEI) [3] which is stated as:

> *The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements,*

the externally visible properties of those elements, and the relationships among them.

From the definition, it can be seen that a software architecture is an abstraction of a software system. It omits some less important information, such as purely local information and private component details. In order to reduce the complexity of the problem at hand in most of this dissertation, the author uses a simplified view of a software architecture. He views software architecture as a system blueprint which comprises components and connectors. Connectors are the relationships among components.

1.2 Software Component Definition

In this section, the author tries to provide a definition to a software component. Since the author uses the following definition of software architecture, component is a pivotal part of a software architecture.

The Software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships between them.

What is a component? Bachman et al. [22] state that "all software systems comprise components"; "phrase component-based system has about as much inherent meaning as 'part-based whole'". To the author, a component is mainly the part(s) of the software system that are the result of the system decomposition. It also could be a COTS component. This definition is very close to Szyperski's definition [23] which is:

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party.

1.3 Architectural Structures and Views

A lot of modern systems are large and very complex. It is very difficult to describe the whole system using just one structure or view. Obviously it takes multiple structures or views to describe all the aspects of a software system. According to reference [4], the architectural structures can be divided into three groups:

- Module Structures
  In this group elements are modules. An element is a unit of implementation.

- Component-and-Connector Structures
  In this group elements are runtime components and connectors. Components are the major executing units; and connectors describe how components interact with each other.

- Allocation Structures
  In this group, elements are resource allocation structures. These structures show the relationship between the elements of a software system and outside world of the system. Examples could like, on what processor a particular element runs on.

1.4 Measuring Software Architecture

Using software architecture has a lot of benefits. Different architects could design the same system in different ways. If there are multiple architectures for a same system which should a user pick over the others. Even there is a single architecture for a system people need to know what the quality of the architecture is. Is this system reliable or is it

easy to maintain? People can see the necessity to evaluate architectures. There are a lot of benefits of measuring software system. The author lists several of them.

- Software measurement can lead to process improvement which leads to lower development cost. Lower development cost means higher productivity, shorter development cycle.

- Based on the measurement of some historical architectures an organization can improve its ability to develop new architectures. After measuring some architectures, the organization should have better understanding about the domain and its systems.

- An organization can increase its customer's confidence by showing its customers the measurements. Measurements show the knowledge of an organization about its systems.

1.5 General Design Rule for Good Architecture Design

Different architects will design different architectures for the same software system. There are no absolute measurements to rank an architecture. It is very difficult to differentiate given architectures in terms of which one is better than another because it depends on how one priorities the non-functional attributes. Although it is hard to tell which architecture is better there are some rules of thumb to follow to make a good software architecture. According to reference [4], these rules can be put into two categories: process recommendation and product (or structural) recommendation. A few of them are listed below.

Process Recommendation

- There should be one architect or a small group of architects.

- The architect or architect team should have all the functional requirements available to them and a list of quality attributes.

- The architecture should be well-documented and easy to understand.

- All the stakeholders should be actively involved in the design of the architecture.

Product Recommendation

- The architecture should be divided into well-defined modules. The design of the modules should base on the principles of information hiding and separation of concerns.
- Each module should have well-defined interfaces that encapsulated or hide changeable aspects.
- Modules that consume data should be separated from the modules that produce data.
- The architecture should do the same thing in the same way throughout the system. This rule shall increase the understandability and productivity of developers. It also shall enhance modifiability.

1.6 Software Architecture Styles

Architectures can be partitioned into architectural styles, which are characterized by topological properties, message data types, and interaction protocols. An architectural style or pattern is analogous to a conventional architectural style in buildings. It consists of a few features and rules for combining them so that architectural integrity is preserved. Architecture styles are determined by the following factors [31] as stated previously:

- A set of element types, such as the data repository
- A set of semantic constraints, such as the properties of a filter
- A topological layout of the elements
- A set of interaction mechanism, such as event-subscriber

Buschmann et al. [72] group architectural patterns into four categories:

- From Mud to Structure
  Patterns in this category support the controlled decomposition of an overall system into components. This category includes the following architecture

patterns: the *Layers pattern* [73], the *Pipes and Filters pattern* [74] and the *Blackboard pattern* [75].

- Distributed Systems
  This category has one pattern, *Broker pattern* [76]. Broker pattern provides a complete infrastructure for distributed system.

- Interactive Systems
  This category has two patterns, the *Model-View-Control pattern* [77], and the *Presentation-Abstraction-Control* pattern [78]. These two patterns support the structuring of software systems that feature human-computer interaction.

- Adaptable Systems
  This category has two patterns inside, the *Reflection pattern* [79] and the *Microkernel pattern* [80]. Both of these patterns support extension of applications and their adaptation to evolving technology and changing functional requirements.

Bass et al. distinguish between at least five distinct styles, the most prominent of which is the Independent Components style. This style is characterized by its:

1. relative genericity
2. topology as an arbitrary graph
3. messages can be data or control signals
4. interaction can take place through a central message medium or through one-to-one links.

1.7 Software Product Line Architecture

In the previous several sections, the author discusses the characteristics of general software architecture. In this section, the author discusses the architecture for a group of systems which is called Product Line Architecture (PLA). This architecture is heavily promoted by Software Engineering Institute (SEI) [15].

Most of the software organization today develop and maintain a set of software products rather than a single product only. These products typically are similar products

in the same application domain and thus significantly share common characteristics. In order to make software-related tasks as efficient as possible, these commonalities among products should be exploited systematically.

Product line engineering is one such approach that tackles the problem by making components systematically as generic as needed for a particular product family and thus allows components to be reused easily within a family context. The architecture shared by multiple products is called product line architecture. A software product line is also referred to as software product family. It originates from domain-specific software architectures [49].

1.7.1 Software Product Line Definition

A Software Product Line (SPL) is defined as:

A set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [15].

Core assets often include, but are not limited to, the architecture, reusable software components, domain models, requirements statements, documentation and specifications, performance models, schedules, budgets, test plans, test cases, work plans, and process descriptions. The architecture is the key among the collection of core assets.

1.7.2 Essential Activities of Product Line Architecture

SEI defines three essential and highly iterative activities that blend technology and business practices. Fielding a product line involves *core asset development* and *product*

development using the core assets under the aegis of technical and organizational *management*, which is illustrated in Figure 1.1 [27].

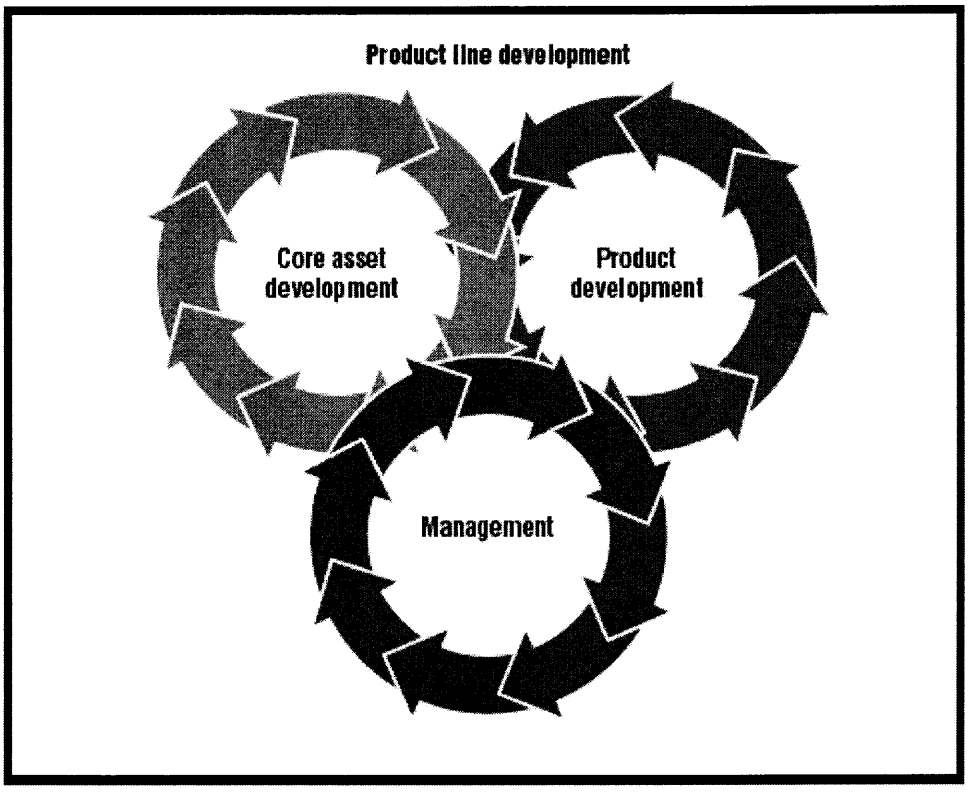


Figure 1.1 Essential product line activities.

- *Core Asset Development*: Establish the production capability for product.
  Outputs: core assets, production scope, and production plan.

- *Product Development*: build products.
  Output: products.

- *Management*: oversees the core asset development and the product development activities, ensuring that the groups building core assets and those building products engage in the required activities, follow the processes defined for the product line, and collect data sufficient to track progress.

1.8 Architecture Description Language

It is necessary to describe software architectures using natural (i.e., common) language so that others can understand their basic functionality and benefit in analysis. An Architecture Description Language (ADL) is a language used to describe software architectures. There are several ADLs developed so far: ACME (developed by CMU), Rapide (Stanford), Wright (CMU), and Unified Markup Language (UML) (OMG). Although some people dispute that UML is not an ADL [88], many people use it as an ADL. In the following several paragraphs, the author gives a short description to a couple of the ADLs.

1.8.1 UML

The author uses UML as the main architecture description language (ADL) throughout this dissertation. UML is becoming a *de facto* language for modeling software system throughout industry and academic world. The history of UML's evolution is described in detail by Kobryn [21]. UML 0.9 unified the modeling notations of Booch [50], Jacobson [51], and Rumbaugh [52]. In 1997, OMG 53 adopted UML 1.1 as an object modeling standard.

Model-Driven Architecture Approach:

OMG is currently promoting Model-Driven Architecture [54] approach for software development. In this approach, UML models are developed prior to implementation. According to OMG, UML is methodology-independent. UML is a notation for describing the results of an object-oriented analysis and design developed via the methodology of

choice. There are several UML tools available. For this dissertation the author uses IBM Rational Rose® [55, 56] as the main tool to draw the UML diagrams.

1.8.2 ACME

ACME was developed by a group at Carnegie Mellon University [86, 87]. It was originally developed as a common interchange format for architecture design tools. Later on it becomes an architecture definition language too. It has the following main features:

- Architectural Ontology, six basic elements: Components, Connectors, Systems, Properties, Constraints, and Styles.
- Flexible Annotation, supporting non structural information.
- Type mechanism, abstracting common, reusable, architectural idioms and styles.
- Open Semantic Framework, reasoning about architectural descriptions.

CHAPTER 2

SOFTWARE METRICS

In this chapter, the author gives the definition of software metrics and discusses the uses of software metrics. Then, he focuses on the discussion of one category of metrics that are architecture metrics.

2.1 Definitions and Classifications of Software Metrics

The groundwork of measuring software systems using software metrics was established in the 1970s [42]. There were four primary technology trends that occurred at that time that have evolved into the metrics used today.

1. Code Complexity Measures
   Examples include McCabe's Cyclomatic Complexity Measure [6], Halstead's Software Science [45].

2. Software Project Cost Estimation
   Examples include Barry Boehm's COCOMO Model 46 and Larry Putnam's SLIM Model [47].

3. Software Quality Assurance

4. Software Development Process

Metrics have proven to be an effective technique for improving software system quality and productivity [43]. Effective management of any process requires quantification, measurement, and modeling. Software metrics provide a quantitative basis

for the software development and validation of models of the software development. Metrics can be used to improve software productivity and quality.

A metric quantifies a characteristic of a process or product. Metrics can be directly observable quantities or can be derived from one or more directly observable quantities. Examples of raw metrics include the number of source lines of code, number of documentation pages, number of staff-hours, number of tests, number of requirements, etc. Examples of derived metrics include source lines of code per staff-hour, defects per thousand lines of code, or a cost performance index. Software metrics can be classified into three categories: process metrics, product metrics, and resource metrics [71].

- Process metrics: are collections of software-related activities which include the following metrics:
  - Maturity metrics
  - Management metrics
  - Lifecycle metrics
- Product metrics: are any artifacts, deliverables or documents that result from a process activity which may include the following metrics:
  - Size metrics
  - Architecture metrics
  - Structure metrics
  - Quality metrics
  - Complexity metrics
- Resource metrics: are entities required by a process activity which include the following metrics:
  - Personal metrics
  - Software metrics

- Hardware metrics

2.2 Software Architecture Metrics

All of these metrics are important to software development. Since this dissertation is to quantify architectural attributes the author focuses on architecture metrics. According to reference [2], architecture metrics can be further divided into three classes:

- Components metrics
  - Number of (language) paradigms
  - Part of standard software
  - Quality level
- Architecture characteristics
  - Open system level
  - Integration level
- Architecture standard metrics
  - Used standards metrics
  - Part of standardization

Today's practices of software metrics utilize global indicators which provide insights into improving the software development and maintenance process. In the future, software metrics may be applied more frequently for the prevention of faults within a feedback mechanism to analyze where problems have occurred so that the development process can be improved.

The author uses software metrics to quantify a variety of attributes of a software architecture. These metrics include: error propagation, change propagation, and

requirements change propagation probability metrics. The author also uses metrics to quantify some attributes of a Product Line Architecture, such as scope, commonality, variability, and applicability.

2.3 Premises of Quantitative Approaches

As stated before, software development requires measurements of evaluations or feedbacks. To measure the quality of a software system or the quality of software development processes, measurement models or mechanisms are required. There are several models that have been proposed, such as the Quality Function Deployment approach [81], the Software Quality Metrics approach, and the Goal Question Metrics approach [82, 83]. Researches done by Forrest Research show that the Goal-Question-Metric model is still the most programmatic approach [84].

2.3.1 The Goal-Question-Metric Approach

The Goal-Question-Metrics (GQM) approach [19] to process metrics, first suggested by Basili and his colleagues, has proven to be a particularly effective approach to select and implement metrics. This approach is based on the assumption that if an organization wants to measure software development or a process it must specify the goal of the measurement. The GQM measurement model has three levels:

1. Conceptual level (Goal). A goal is defined for an object with respect to some attributes of the software systems or processes.

2. Operational level (Question). A set of questions is used to characterize the way the assessment/achievement of a specific goal is achieved.

3. Quantitative level (Metric). A set of data is associated with every question in order to answer it in a quantitative way

The structure of the GQM model is shown in Figure 3.1.

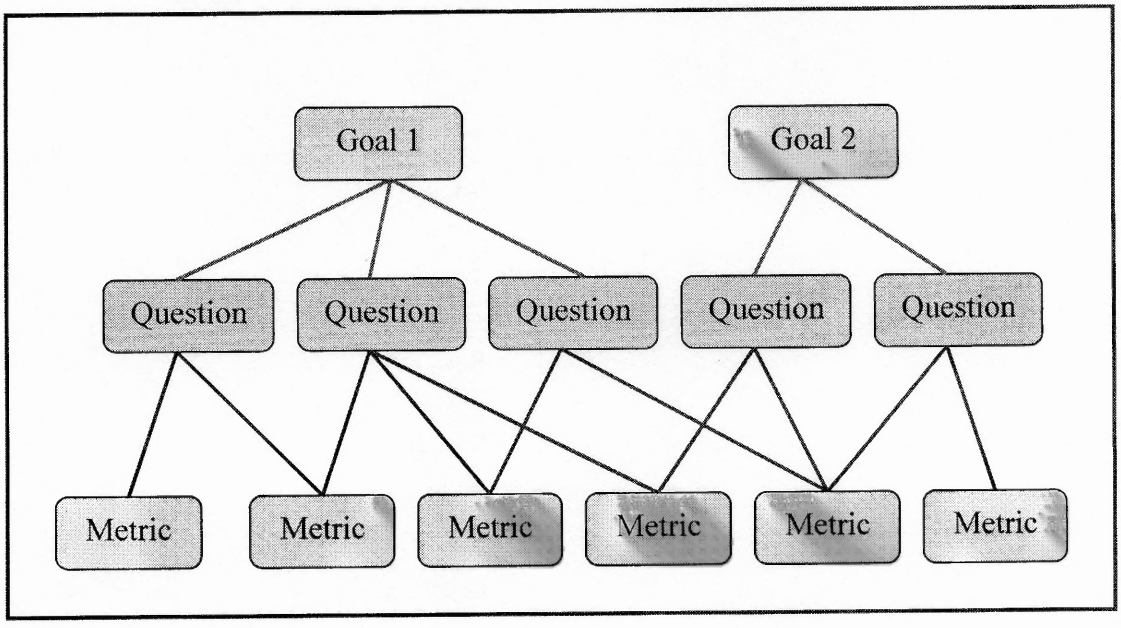


Figure 2.1 Goal-Question-Metrics model structure.

2.3.2 Goal-Function-Metric Paradigm

The GQM model is for general software development measurement. It requires knowledge of the systems or processes to be measured. This dissertation work is to study software architectures. There is no system specific information available at the measurement time. The GQM model can not be directly applied to this study. The model has to be modified or extended to suit this study.

The focus of this study on software architectures (rather than source code, for example) has a direct impact on what attributes the author may wish to define, characterize and quantify. Traditional software metrics that characterize source code (e.g. complexity [6], fault density, size) or depend on the executable/operational nature of source code for their definition (e.g. reliability, dependability) are not meaningful at the

architectural level. Architectural quality attributes can be divided into two distinct classes:

- Attributes that view the software architecture as an intrinsic product, and characterize it as such.
- Attributes that view the software architecture as a blueprint for operational software systems, and characterize it by the properties of these systems.

In this study, the author focuses on the latter class, so that when he states that an architecture has some attributes, he actually means that operational software systems that are derived from this architecture have these attributes (or are likely to).

As a matter of separation of concerns, and in order to facilitate the discussions, the author defines a three-tier hierarchy of attributes [17]:

- *Qualitative Attributes,* which represent relevant features of an architecture that the author wants to define and characterize. These are typically complex, multi-dimensional attributes, for which people have some intuitive understanding, but not necessarily a simple quantitative definition. Examples include: dependability, maintainability, evolvability.
- *Quantitative Functions,* which represent formally defined functions that may be related to the qualitative attributes or may represent some aspect of a qualitative attribute. These are typically easy to define but not necessarily easy to compute/estimate in practice. Examples include: error propagation probabilities (related to dependability); change propagation probabilities (related to maintainability); requirements propagation probability (related to evolvability).
- *Computable Metrics,* which represent quantitative functions that people can compute by analyzing the architecture [8, 17]. These are typically used as approximations for quantitative functions. Example: the entropy of random variables representing the flow of information through an architecture [18].

In the spirit of the *Goal/Question/Metric* of Basili and Rombach [19], the author maps the attribute of interest onto a computable metric that can be evaluated on the basis of information that is available at the architectural level. At this level, usually the wealth of structural and semantic information that is available at the code level is not available,

but the available information is the information about the flow of control and data within components and between components. This precludes using traditional software metrics, which are based on such code-level features as tokens [20], flow graphs [6], data dependencies [7] and control dependencies (to mention a few). The author's approach can further be characterized by a combined *Bottom-Up/Top Down* discipline, whereby he complements the top down approach advocated by Basili and Rombach's *Goal/Quality/Metric* paradigm with a bottom up approach that analyzes the architecture and derives a matrix that quantifies architectural attributes. In the absence of structural and semantic information, the author cannot analyze the flow of information within an architecture deterministically; hence he resorts to a stochastic approach.

The link between qualitative attributes and quantitative functions is usually an informal, intuition-based relation, such as: if one is interested in this qualitative attribute, he ought to look at this quantitative function, as it quantifies/reflects one aspect/dimension of the attribute, or is otherwise related to it. By contrast, the link between a quantitative function and a computable metrics is more formal: the quantitative function is approximated by means of computable metrics. Along with the formula for the quantitative function, the author usually aims to provide: the approximations that are made in deriving the formula; the rationale for the approximations; the conditions under which the approximations are most legitimate; and the scope of the approximation.

Part II ERROR PROPAGATION PROBABILITIES

CHAPTER 3

ERROR PROPAGATION ANALYTICAL STUDY

In this part, the author focuses his attention on the study of error propagation probabilities in an architecture. Given an architecture made up of N components, the author derives an N × N matrix whose entry at row A and column B contains the probability that an error in component A propagates to component B at run time. If the author claims that this quantitative function is related to dependability, he does not mean that he represents dependability by this function; he merely means that if he is interested in the attribute of dependability, then he may want to look at this function as it reflects some relevant aspect of dependability.

The contributions of this part are outlined as follows:

- The author presents a simple but formal definition of the error propagation probability between two components based on an error in the message from the sending component and the behavior of the receiving component.

- The author derives an analytical expression for estimating error propagation probabilities based on information theoretic entropy measure for the information exchanged between components and the impact of this information on the dynamic behavior of the receiving components.

- The author derives an upper bound on the error propagation probability which can be calculated easily based on the number of messages and the number of states of the receiving component.

- The author derives an expression for the cumulative error propagation measuring the probability that an error propagates from a sending component to a receiving component through any number of intermediate components.

- The author defines a single scalar measure of an architecture called the *Error Insulation Coefficient that* reflects, using a value between 0 and 1, how close they are to the ideal architecture in which errors does not propagate to other components.

- The author presents an empirical approach for measuring error propagation probabilities and uses it to validate the analytical measures.

Given a matrix of error propagation probabilities, one may want to use it in the following manner:

- If row A has high values, it means that an error in A is likely to propagate widely through the architecture; this calls for taking special steps to minimize faults in A at design/development time, so as to minimize errors at run-time.

- If column B has high values, it means that B tends to be affected by errors arising throughout the architecture; this calls for taking special steps to immunize B against errors, e.g. by providing it with fault tolerance capabilities (error detection, damage assessment, error recovery).

- If one is considering two candidate architectures and dependability is an important consideration, he may want to compute their error propagation matrices and rank the candidates accordingly.
- An ideal error propagation matrix is an identity matrix, which has one's on the diagonal and zero's outside. One can measure how close a given error propagation matrix is to the identity matrix, and use this as a scalar measure of the fitness of an architecture with respect to error propagation.

If, in addition to the matrix of error propagation probabilities (say, EP), a vector (say, V) can be derived. V measures for each component the probability that an error originates in the component. One can multiply vector V with EP to obtain another vector (say, W) that represents for each component the probability that an error arises in the component, be it by originating there or by propagating from another component. Vector V can be established by considering the frequency with which components are expected to be invoked, and the fault density of components (which can, in turn, be estimated from

component complexity). If vector W has a high value for component A, then A must be provided with means to handle errors (e.g. error detection and error recovery).

3.1 Error Propagation

Let us assume components A and B of an architecture are to be analyzed. Let X be the connector that carries information from A to B. For the purposes of the current discussion, the specific form of connector X is not important. It will be merely modeled it as a set of values that A may transmit to B. Also, the specific form of components A and B is not important for the purposes of this discussion. The author will merely model them as functions that map an internal state and an input stimulus into a new state and an output.

Definition 1. The *Error Propagation Probability* from component A to component B is denoted by EP(A,B) and defined by:

$$EP(A, B) = Prob([B](x) \neq [B](x') \mid x \neq x'), \tag{3.1}$$

Where [B] denotes the function of component B, and x is an element of the connector X from A to B. the author interprets [B] to capture all the effects of executing component B, including the effect on the state of B as well as the effect on any outputs produced by B. The author interprets EP (A, B) as the probability that an error in A is propagated to B (as opposed to being masked by B) because the outcome of executing B will be affected by the error in A. By extension of this definition, one can let EP (A, A) be equal to 1, which is the probability that an error in A causes an error in A. Given an architecture with N components, one can let EP be an $N \times N$ matrix such that the entry at row A and column B be the error propagation probability from A to B.

Note that nothing in the definition above indicates that x' is an erroneous message; all the definition says is that x' is different from x—as far as this definition is concerned, both could be correct. While this may seem to be an anomaly, all it means is that the author is measuring error propagation probabilities by a wider property, which is the probability that different arguments are mapped by function [B] to different images (a measure of injectivity of [B]).

3.2 Cumulative Error Propagation

Note that the definition of the error propagation given above uses the concept of *conditional* probability, i.e. the author calculates the probability that an error propagates from *A* to *B under the condition that A actually transmits a message to B*. It is often useful, however, to use the *unconditional error propagation* which the author denotes simply as E (*A*, *B*), and defines as the probability that an error propagates from *A* to *B* not conditioned upon the event that *A* sends a message to *B*. Function *E (A, B)* is clearly dependent on *EP (A,B),* but it further integrates the probability that *A* does send a message to *B*. In order to bridge the gap between the original *(conditional) error propagation* and the newly introduced *unconditional error propagation*, let us consider the *transmission probability matrix T* where the entry *T*(*A*, *B*) reflects the probability that connector (*A* → *B*) will be activated during a typical/canonical execution. *T* is the $N \times N$ matrix whose entry *T*(*A*, *B*) is the probability that the component *A* sends a message to component *B* given that the *A* is expected to transmit a message to *some* component. Note that:

- It is reasonable to assume that *T*(*A*, *A*) = 0 for all components *A*,

- Clearly, T is a *stochastic* matrix, i.e. $\sum_{B} T(A,B) = 1$ for every component A.

The matrix T is used to distinguish between a connector that is invoked intensively in each execution and one that is invoked only occasionally, under exceptional circumstances. The matrix T reflects the variance in frequency of activations of different connectors during a typical execution. By virtue of simple probabilistic identities, the author finds that the *unconditional error propagation* is obtained as the product of the conditional error propagation probability with the probability that the connector over which the error propagates is activated, i.e.

$$E(A,B) = EP(A,B) \times T(A, B) \tag{3.2}$$

The concept of *unconditional error propagation* is useful when the author discusses *cumulative error propagation probabilities*, which the author does in the next subsection. Whereas so far the author has focused his attention on single step error propagation from some component A to some component B, he wants to consider, now, the probability that an error in some component A propagates to some component B in an arbitrary number of transmissions (steps) starting in A and ending in B. The author calls this the *cumulative error propagation probability* from A to B. The author submits two premises pertaining to the analysis of cumulative error propagation:

Cumulative error propagation probabilities must be derived, not from matrix EP (re: conditional error propagation probabilities) but rather from matrix E (re: unconditional error propagation probabilities). Indeed, the probability that an error propagates along some path depends first and foremost on the probability that the path is

actually taken (re: matrix *T*), combined with the probability that the error is propagated through each arc of the path (re: matrix *EP*).

Second, the matrix of cumulative error propagation probabilities cannot be derived as the traditional transitive closure of matrix E, because while matrix T is stochastic, matrix E is not (there is no basis to claim that if component *A* has an error, then it propagates to one component or another with probability 1). Hence the author needs to find a specific formula for this case, which he does in the sequel.

The author denotes the cumulative error propagation probability from A to B by E*(A, B), and lets E* be the matrix of such probabilities. The detailed mathematical developments that the author has followed to derive the formula of *E** are given in Appendix A. The author contents himself, in this part with presenting the formula:

$$E^*(A,B) = \sum_{s\geq 0} \mathrm{E}_s(A,B), \tag{3.3}$$

Where E_s is the *s-step* error propagation matrix, i.e. E_s(A, B) is the probability that an error in A propagates to B via *exactly s* connectors. The s-step error propagation matrix E_s is given by:

$$E_S(A,B) = \sum_{\substack{1\leq C_1,C_2,\ldots,C_{s-1}\leq N \\ C_r\neq B \text{ for} 1\leq r\leq s-1}} \mathrm{E}(A,C_1)\mathrm{E}(C_1,C_2)..\mathrm{E}(C_{s-1},B), \tag{3.4}$$

Where A≠*B*; and $E_s(A, A) = 0$ for all *A*.

3.3 The Error Insulation Coefficient

Whereas the two previous features are matrices (like the original EP), this feature is a scalar. Through this scalar, the author aims to reflect the potential of an architecture to insulate its components from each other's errors. Obviously, the ideal *Error Insulation Coefficient (EIC)* corresponds to a *conditional* error propagation matrix where only the diagonal cells are 1s and the rest of the matrix is 0, i.e. an identity matrix; in such a matrix, no component propagates errors to other components. At the other extreme, the *worst possible EIC* is one for which all cells of the *conditional* error propagation matrix are 1s; in such a matrix, whenever an error arises in some component, it propagates to all other components. The author wishes to define the error insulation coefficient in such a way as to reflect, using a value between 0 and 1, how close the matrices are to the ideal matrix and how far the matrices are to the worst possible matrix. Without dwelling into detailed mathematics, the author submits that the following formula satisfies the criteria, and let it be the definition of the *Error Insulation Coefficient* (*EIC*):

$$EIC = \frac{\sum_{A} \sum_{A \neq B} EP(A, \mathrm{B})}{N^2 - N} \tag{3.5}$$

Where N is the number of components in the architecture. A low EIC value indicates an architecture where errors do not easily propagate between its components.

3.4 Estimating Error Propagation Analytically

The author has found (see Appendix A) that analytically, the error propagation probability (as defined in this chapter), can be expressed in terms of the probabilities of the individual A-to-B messages and states, via the following formula:

$$EP(A \rightarrow B) = \frac{1 - \sum_{x \in S_B} P_B(x) \sum_{y \in S_B} P_{A \rightarrow B}[F_x^{-1}(y)]^2}{1 - \sum_{v \in V_{A \rightarrow B}} P_{A \rightarrow B}[v]^2}, \quad (3.6)$$

where $F_x^{-1}(y) = \{ v \in V_{A \rightarrow B} \mid F_x(v) = y \}$, and the author assumes a probability distribution P_B on the set of states S_B of component B, and a probability distribution $P_{A \rightarrow B}$ on the set of messages $V_{A \rightarrow B}$ passed from A to B.

The term $\sum_{v \in V_{A \rightarrow B}} P_{A \rightarrow B}[v]^2$ in the denominator of Equation (3.6) is an exponent of the 2nd order Renyi entropy [9], which according to the recent studies [12] is closely related to the classical Shannon entropy [10]. If one can assume that the states of B, as well the messages passing through the connector from A to B are equi-probable, the Equation (3.6) for error propagation is simplified into

$$EP(A \rightarrow B) = \frac{1 - \frac{1}{|S_B| \, |V_{A \rightarrow B}|^2} \sum_{x \in S_B} \sum_{y \in S_B} |F_x^{-1}(y)|^2}{1 - \frac{1}{|V_{A \rightarrow B}|}}, \quad (3.7)$$

Since the software practitioner cannot always extract from the available artifact the detailed information on the transition table F for the architectural components, it would be helpful to be able to estimate the right-hand side of (3.7) without using any

knowledge of function F. The following inequality (see Appendix for the proof) gives precisely such an estimate (upper bound)

$$EP(A \rightarrow B) \leq \frac{1 - \frac{1}{|S_B|}}{1 - \frac{1}{|V_{A \rightarrow B}|}}, \tag{3.8}$$

Thus, for instance, if the receiving component B has two states, and is capable of receiving five different messages from component A, the A-to-B error propagation cannot exceed (1-0.5)/(1-0.2) = 0.625.

Notice (see Appendix A for explanation) that the inequality (3.8) can be used as a close approximation of the actual $EP(A \rightarrow B)$ value whenever for every initial state of B the number of messages that trigger its transition to a new state is approximately the same, no matter what that new state is.

3.5 Example: A Command and Control System

The example case the author uses to illustrate the work is a large command and control system that is used in a life-critical, mission-critical application. This system was modeled using the Rational Rose Realtime CASE tool [13]. It is a Computer Software Configuration Item (CSCI) that provides the following functions:

- Facilitating Communication, Control, Cautions and Warnings including subsystem Configuration Management, C&DH (Communication and Data Handling), Communications Control, Processing, Memory Transfer, C&DH Failure Detection, Isolation, and Recovery and Time Management,
- Controlling a secondary electrical power system, and
- Environmental Control, which provides temperature and humidity Control.

The author concentrates on the Thermal Control part of the system, which is a rather complex system with operations setting controller, fault recovery procedures, and pump control functionalities. The system is responsible for providing overall management of pumps as well as performing the necessary monitoring and response to sensors data. Also, it is responsible for performing automated startup, and controlling Thermal System reconfigurations. During each execution cycle, a check is performed for incoming commands. Received commands are validated in the same execution cycle. Mode change commands, which will reconfigure the Internal Thermal System, are also accepted from other components of Thermal System to compensate for system component failures or coolant leaks. A failure recovery system detects failure conditions and performs recovery operations in response to the detected failures. Failure conditions include combinations of Pump failures and Shutoff Valve failures.

The system has a hierarchical architecture. The top-level software architecture of this system is shown in Figure 3.1. Also, the internal architecture of SubSystem Z is shown in Figure 3.2. Using these artifacts, one can identify the components and the connectors that describe the components-based system architecture and label the EP matrix rows and columns with the components names.

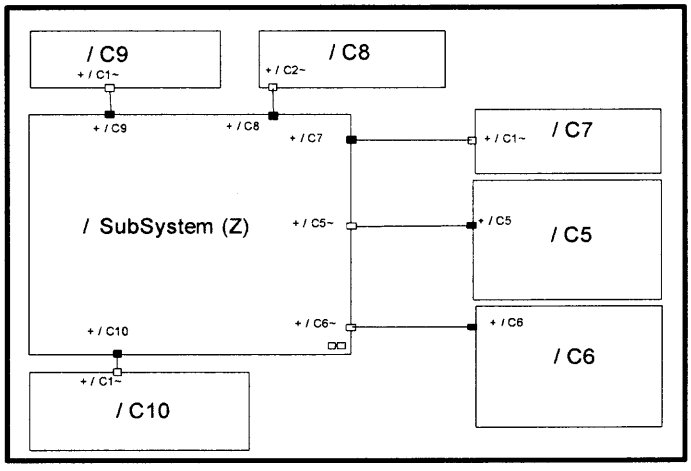


Figure 3.1 Top-level software architecture of the system.

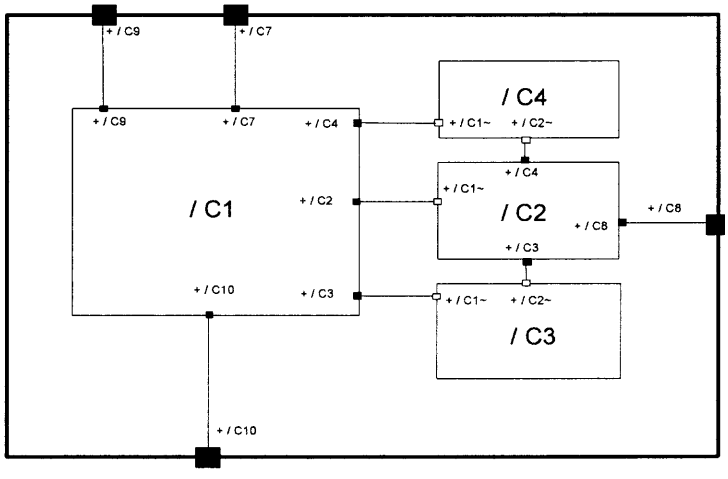


Figure 3.2 The architecture of Sub-System (Z).

Figure 3.3 shows a sample message protocol between a pair of components in the system. This artifact provides the author with the message set $V_{A\rightarrow B}$ and $V_{B\rightarrow A}$ that is going between the two components A and B. Similarly, using the Rose-RT tool the

author can get the whole sets of messages that are going on between each pair of components in the system.

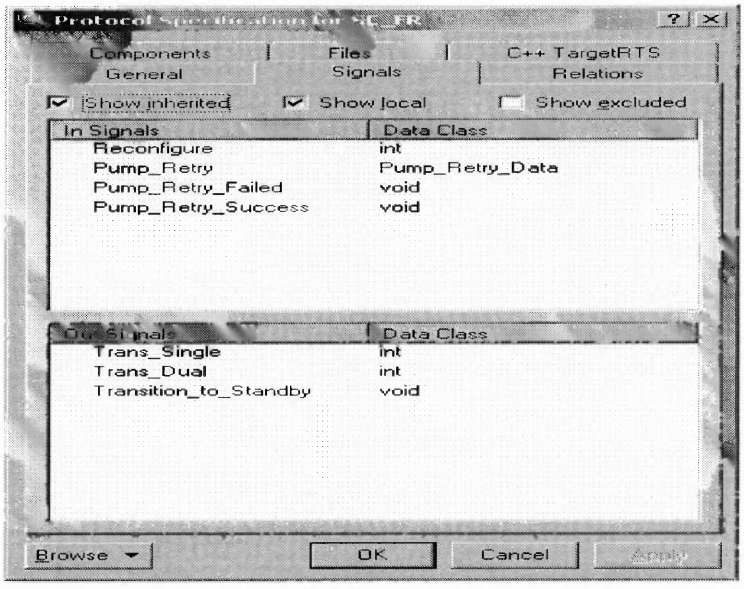


Figure 3.3 A sample of a sanitized message protocol.

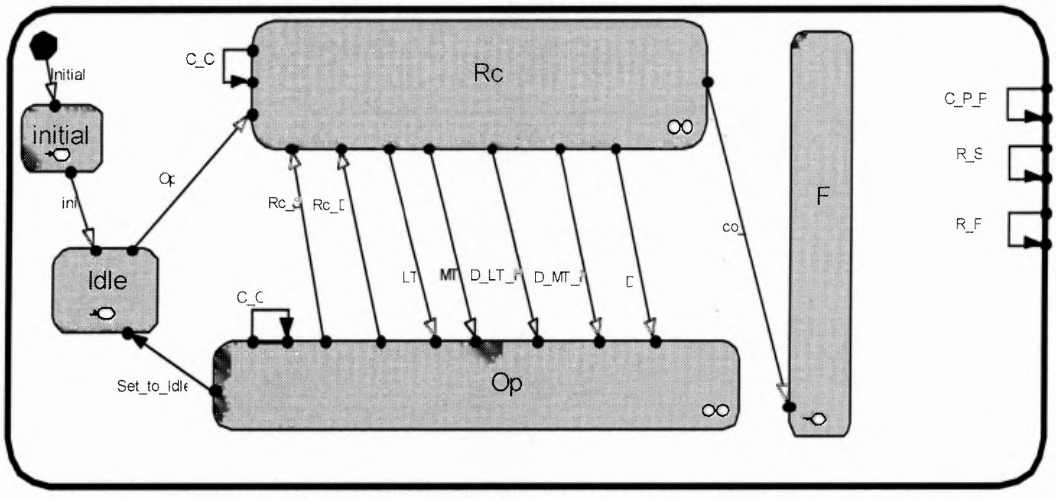


Figure 3.4 A state diagram of a component.

The state chart shown in Figure 3.4 is a sample of state chart of a component in the system. This provides us with the state set S_B for this sample component. Using the

Rose-RT tool, the author can easily identify the triggering messages from one state to another. In a similar way, one can get all the state sets for all the components.

3.6 Analytical Results

Considering the CSCI system discussed above, the author gets the set of states S_B and messages $V_{A \to B}$ from the artifacts of the system specification. The author obtains the matrix EP of (conditional) error propagation probabilities of this system (Table 3.1), using the approximation (3.8). The author assumes equi-probability of states and messages.

Table 3.1 Conditional Error Propagation Matrix - Analytical Results

		B									
		C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
A	C1	1.0000	0.1061	0.4210	0.3368	0.4472	0.4623				
	C2	0.2001	1.0000						0.5238		
	C3	0.0105	0.4722	1.0000							
	C4	0.0190	0.2332		1.0000						
	C5		0.2765			1.0000					
	C6		0.1265				1.0000				
	C7	0.3761						1.0000			
	C8								1.0000		
	C9									1.0000	
	C10	0.0014									1.0000

For this particular case study, the author has derived the connector activation matrix *T* as a stochastic matrix of probabilities that contains for each entry (A, B), the probability that connector (A, B) is activated, given that component A is broadcasting a message. Using this connector activation matrix, the author derives the *unconditional*

error propagation matrix EA, also referred to as the 1-step error propagation matrix of the system; this is given in Table 3.2. The author gets the matrix *T* through a simulation of the system representing the operational profile of the execution.

Table 3.2 Unconditional Error Propagation Matrix - Analytical Results

		B									
		C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
A	C1		0.0012	0.0132	0.0102	0.0146	0.0145				
	C2	0.1104							0.1264		
	C3	0.0060	0.2024								
	C4	0.0107	0.1026								
	C5		0.1005								
	C6		0.0506								
	C7	0.3761									
	C8										
	C9										
	C10	0.0014									

Using the unconditional error propagation matrix, say E_A, given above, the author derives the matrix of cumulative error propagation probabilities, which called E^*_A. The author finds the following matrix (Table 3.3):

Table 3.3 Cumulative Error Propagation Matrix - Analytical Results

		B									
		C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
A	C1	1.00	1.16E-03	1.32E-02	1.02E-02	1.46E-02	1.45E-02		1.46E-04		
	C2	1.10E-01	1.00	1.46E-03	1.12E-03	1.61E-03	1.60E-03		1.26E-01		
	C3	6.04E-03	2.02E-01	1.00	6.15E-05	8.82E-05	8.78E-05		2.56E-02		
	C4	1.07E-02	1.03E-01	1.41E-04	1.00	1.56E-04	1.55E-04		1.30E-02		
	C5	1.11E-02	1.01E-01	1.47E-04	1.13E-04	1.00	1.61E-04		1.27E-02		
	C6	5.59E-03	5.06E-02	7.39E-05	5.69E-05	8.15E-05	1.00		6.40E-03		
	C7	3.76E-01	4.35E-04	4.98E-03	3.83E-03	5.49E-03	5.47E-03	1.00	5.49E-05		
	C8								1.00		
	C9									1.00	
	C10	1.41E-03	1.62E-06	1.86E-05	1.43E-05	2.05E-05	2.04E-05	0.00	2.05E-07	0.00	1.00

Except for possible round-off errors, matrix E^*_A is greater than matrix E_A, entry by entry. The error isolation coefficient of this architecture is found to be:

$$EIC = 0.0447$$

CHAPTER 4

EMPIRICAL STUDY OF ERROR PROPAGATION

4.1 Fault Injection Experiment

In order to validate the analytical study, the author developed a framework for experimental error propagation analysis in which he utilizes fault injection experiments to alter architecture specifications. The author then simulates the corrupted specifications and records component traces as "faulty-run" logs. Finally he compares the faulty-run logs against a fault-free "golden-run" log obtained by simulating the uncorrupted architecture specifications [14]. The author performs the simulation-based error propagation analysis in two phases (as shown in Figure 5.1): an acquisition phase and an analysis phase. In the acquisition phase:

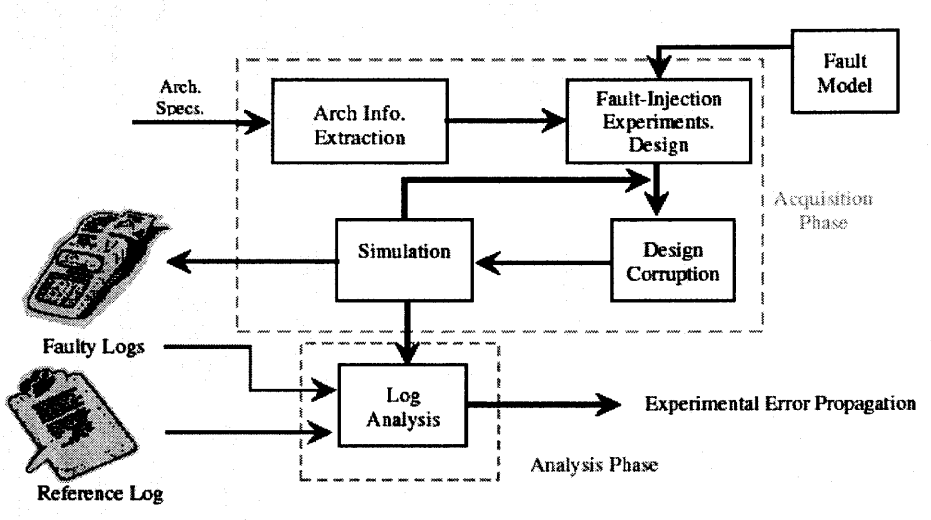


Figure 4.1 The framework of experimental error propagation analysis.

The author extracts architecture information about the components and connectors that make up the software system. The underlying finite state machines in the components, and the inter-component messages. The author uses a message swapping fault model [36] to generate fault injection experiments. In each of the fault injection experiments the author replaces all occurrences of a message nominally flowing over a connector (from component A to component B) by a different message (as a result of an error in component A) that belongs to the set of messages that A may send to B.

The author simulates the corrupted specifications and records simulation traces for the different experiments that cover all messages for the different connectors present in the architecture. The faulty-run logs also contain markers that indicate instants when faults are injected.

In the analysis phase: the author conducts post-simulation comparison between the faulty-run logs and the reference (fault-free) log. The comparisons are based on state transitions at simulation time instances following a fault injection. Immediately before injection of a fault, there is no difference between the state of component B as recorded on the reference log and its state recorded on the faulty log. After a fault is injected, any discrepancy between the two logs is due to error propagation from A to B. A faulty-message propagating (from A to B) will at most cause a single instance of error propagation (from A to B).

The author computes the experimental error propagation probability from component A to B as the percentage of the fault injections (corresponding to errors in A) that propagate to component B.

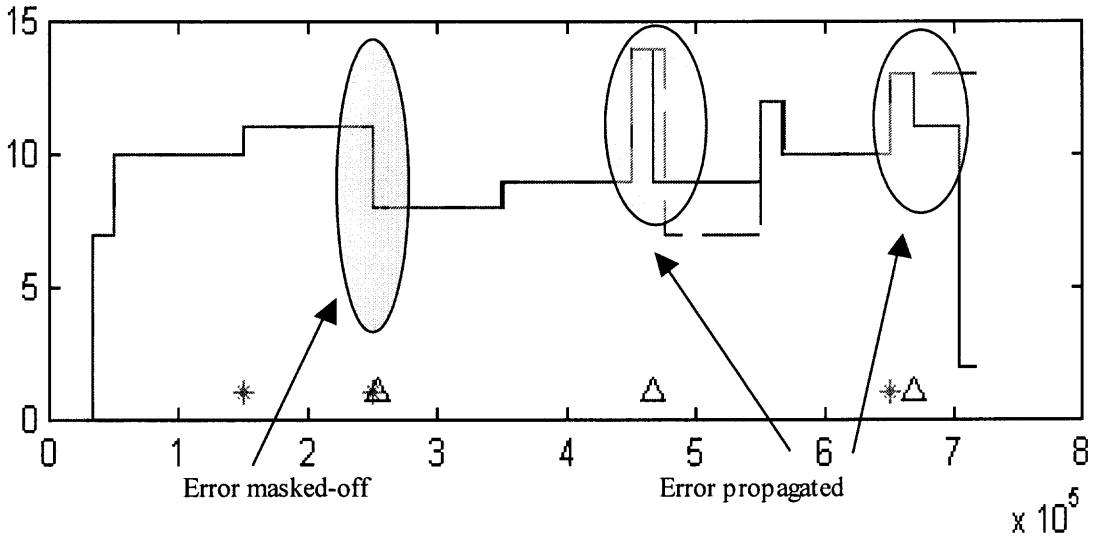


Figure 4.2 An illustration of log comparison.

Figure 4.2 shows an example of how the author compares a faulty-run log (dashed line) with a reference log (solid line). The indices on the vertical axis correspond to the states of the receiving component (B), while the horizontal axis indicates the time during the simulation (in milliseconds). The triangular marks (Δ) correspond to instants of injecting the fault. The star symbols (*) correspond to log-alignment markers, which the author uses to compensate for any timing skews between faulty logs and the reference log. In this example, three faults were injected and in two incidences of the three, error propagated (from the sending component A) to component B whose faulty state-transition log is shown in Figure 4.2.

4.2 Experimental Results

Table 4.1 shows the experimentally obtained error propagation matrix E_E from the fault injection experiment explained in section previous section. Considering the same mode of operation whose analytical error propagation matrix E_A is given in Table 4.1. (Note: for both Table 4.1 and 4.2 blank cell means zero value)

Table 4.1 Unconditional Error Propagation Matrix E_E - Empirical Results

		B									
		C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
A	C1		0.5000	0.0557	0.4912	0.1331	0.1280				
	C2	0.7838							0.4286		
	C3	0.0161	0.7083								
	C4	0.7917	0.6429								
	C5		1.0000								
	C6		1.0000								
	C7	0.7500									
	C8										
	C9										
	C10	1.0000									

Using matrix E_E, the author derives matrix E^*_E of cumulative error propagation probabilities, and finds the results shown in Table 4.2. Note that except for round-off errors, this matrix is greater than the matrix of unconditional probabilities, entry by entry.

Table 4.2 Cumulative Error Propagation - Experimental Results

		B									
		C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
A	C1	1.00	5.00E-01	5.57E-02	4.91E-01	1.33E-01	1.28E-01		2.14E-01		
	C2	7.84E-01	1.00	4.37E-02	3.85E-01	1.04E-01	1.00E-01		4.29E-01		
	C3	1.61E-02	7.08E-01	1.00	7.91E-03	2.14E-03	2.06E-03		3.04E-01		
	C4	7.92E-01	6.43E-01	4.41E-02	1.00	1.05E-01	1.01E-01		2.76E-01		
	C5	7.84E-01	1.00E+00	4.37E-02	3.85E-01	1.00	1.00E-01		4.29E-01		
	C6	7.84E-01	1.00E+00	4.37E-02	3.85E-01	1.04E-01	1.00		4.29E-01		
	C7	7.50E-01	3.75E-01	4.18E-02	3.68E-01	9.98E-02	9.60E-02	1.00	1.61E-01		
	C8								1.00		
	C9									1.00	
	C10	1.00E+00	5.00E-01	5.57E-02	4.91E-01	1.33E-01	1.28E-01		2.14E-01		1.00

4.3 Statistical Validation

In this section, the author confronts the results computed by the analytical formula from Chapter 3 against the results derived from the fault injection experiment from previous section to assess the validity of the analytical formulas. He lets E_A and E_E be (respectively) the analytical matrix and the empirical matrix of (unconditional) error propagation for the sample architecture; these are both 10×10 matrices. The author then uses a number of criteria to this effect:

The first possible criterion is simply the correlation between the entries of the two matrices; because these matrices contain 100 values each, the correlations do bear some significance. The second possible criterion is to correlate, not all the values of the matrices, but rather the non-trivial values (other than those that are either 0 or 1 by definition); the rationale behind this criterion is that trivial values do not really test the analytical results. The third criterion discriminates between empirical values that were

derived from a small number of fault injections and those that were derived from a large number of fault injections. If the analytical results are accurate, one should find empirical values that stem from large numbers of fault injections to be highly correlated to their corresponding analytical values, whereas those values than stem from small numbers of fault injections are not guaranteed to correlate to their corresponding analytical results.

Orthogonally, the author finds it useful to compare not only E_A and E_E, which represent single step propagations, but also cumulative versions of these matrices, which represent probabilities of error propagations that may have taken more than one step through the architecture. There are two reasons why the author may want to consider the cumulative matrix E* in addition to the single-step matrix E:

A person interested in the probability that an error in *A* propagates to *B* from a practical perspective does not usually care how many steps the propagation requires. Hence, E* is a better reflection of what the author wants to measure than E. Also, if there is any discrepancy between what the analytical study defines as a single step and what the empirical study does, this discrepancy will be smoothed out once the author consider propagations of arbitrary length. In the remainder of this chapter, the author applies these criteria to matrices E and E* in order to determine to what extent the values obtained empirically are consistent with those found analytically.

4.3.1 Correlating One Step Matrices

In this section, the author presents the results of the study that he conducts to explore the correlation between the analytically estimated single step error propagation matrix and its experimentally derived counterpart. The correlation coefficient between all the cells of

the analytical E_A matrix and the experimental E_E matrix is:

$$Cor(E_A, E_E) = 0.628 \qquad \text{(r value)}$$

where 'r' denotes the Pearson product-moment correlation coefficient.

The author notes, however, that there are only 15 non-trivial entries in each of the two matrices. Trivial entries correspond to self-loops from a component to itself (with error propagation probability of 1 by definition) and to the non-directly connected components (with error propagation probability of 0 by definition). It may be useful to evaluate the correlation between the set of non-trivial values of matrices EA and E_E having a significant number of fault injections(> 20). The connectors that have < 20 fault injections are shaded in Table 6.1. The author finds:

$$Cor'(E_A, E_E) = 0.5576 \qquad \text{(r value)}$$

Table 4.3 contains the 15 non-trivial entries corresponding to the 15 connectors over which faults were injected during the controlled experiment. Note that the number of injected faults over connectors varies considerably across the entries. The connectors in the table follow a descending order with respect to the number faults injected over each connector. Overall, the correlation decreases as the number of injected faults drop, although not monotonically; the disturbances in the first few rows may stem from the fact that a correlation is not necessarily meaningful when too few values are involved.

Table 4.3 Correlation between Analytical and Experimental EP Probabilities

Entry	Connector		E_A	E_E	# Injected Faults	Correlation Coefficient	
	From	To					
1	1	5	0.0146	0.1331	1067	N/A	N/A
2	1	6	0.0145	0.1280	1055	1.0000	Entries 1 through 2
3	1	3	0.0132	0.0557	592	0.9999	Entries 1 through 3
4	3	1	0.0060	0.0161	559	0.8708	Entries 1 through 4
5	7	1	0.3761	0.7500	64	0.9894	Entries 1 through 5
6	1	4	0.0102	0.4912	57	0.8160	Entries 1 through 6
7	2	1	0.1104	0.7838	37	0.7153	Entries 1 through 7
8	1	2	0.0012	0.5000	36	0.6488	Entries 1 through 8
9	4	2	0.1026	0.6429	28	0.6433	Entries 1 through 9
10	3	2	0.2024	0.7083	24	0.6829	Entries 1 through 10
11	4	1	0.0107	0.7917	24	0.5576	Entries 1 through 11
12	2	8	0.1264	0.4286	7	0.5501	Entries 1 through 12
13	5	2	0.1005	1.0000	4	0.5068	Entries 1 through 13
14	6	2	0.0506	1.0000	4	0.4291	Entries 1 through 14
15	10	1	0.0014	1.0000	4	0.3240	Entries 1 through 15

The results in this table are interesting, in that they show a fairly high correlation between experimental results and analytical results in those cases where the experimental result is based on a large number of fault injections. Also, predictably, the correlation drops (as shown in Table 4.3) as the number of fault injections drop (though not monotonically). [You haven't described why the bottom entries are shaded.]

4.3.2 Correlating Cumulative Matrices

The correlation between matrices E_A and E_E represent the one-step unconditional error propagation probabilities, estimated analytically and experimentally In addition to analyzing this correlation, the author is interested in analyzing the correlations between matrices E^*_A and E^*_E, which represent the cumulative (multi-step) versions of these

matrices. The results of the study are shown in Figure 6.2. The author finds for all elements,

$$Cor\ (E^*_A, E^*_E)=0.737 \qquad \text{(r value)}$$

Also, the author finds that the correlation for multi-step error propagation for non-trivial values, is

$$Cor'(E^*_A, E^*_E) = 0.460 \qquad \text{(r value)}$$

Hence the analytical formula can be used to predict cumulative error propagation probabilities throughout an architecture with a significant positive correlation, at least in this sample case study - all the while using nothing more than the UML-RT description of the system.

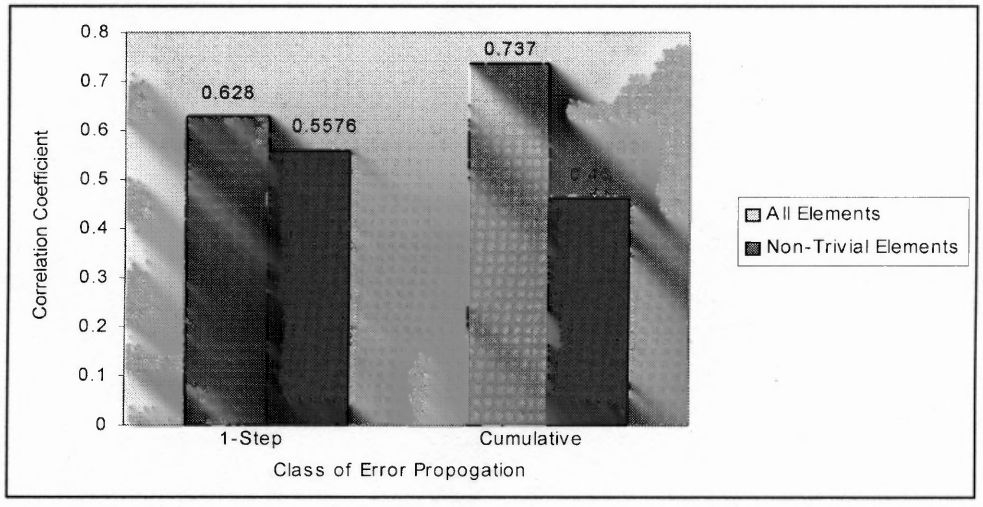


Figure 4.3 Correlation between analytical and empirical error propagation.

4.3.3 Statistical Significance of the Correlation

Now the author wants to validate the results, i.e. to make sure that the positive correlation values observed are statistically significant. To further test the relationship between

analytical and experimental error propagation hypothesis testing was done using the T-test (One-tail) [37] for the non-trivial entries using the level of significance $\alpha = 0.05$

- *H0*: $\rho = 0$ (There is no linear association between analytical change propagation values and empirical change propagation values).

- *H1*: $\rho > 0$ (There is a positive linear association between analytical change propagation values and empirical change propagation values).

The null-hypothesis is rejected when the P value is smaller than 0.05. The author computes the value of t statistic for the non-trivial values of 1-step matrices $t_{ob} = 2.015$ (n=11), and the corresponding $P < 0.05$; whence he infers that the correlation of 0.628 is statistically significant. Likewise, the value of t statistic for the non-trivial values of cumulative error propagation matrices is found to be $t_{ob} = 3.5893$ (n=50), and the corresponding $P < 0.05$ which shows that the correlation between experimental and analytical cumulative error propagation matrices is statistically significant.

The T-test results showed that there is a linear association between analytical error propagation and experimental error propagation as well as between cumulative analytical error propagation and cumulative experimental error propagation as in both cases based on the P value the null hypothesis of no linear association was rejected.

4.4 Conclusion

In this chapter, the author has derived an analytical approach to estimate the probability of error propagation between components in a software architecture. Further, he illustrates his proposed formula by means of a fault injection experiment, applies on a large command and control system, and found a fairly meaningful correlation between

the analytical estimates and the experimental observations. Given that the analytical approach is based on architecture specifications, and uses exclusively information that is typically available at an architectural level, the author submits that the results can be used to estimate the error propagation behavior of an architecture, at a time when relatively little is known about the actual execution of products that instantiate the architecture. In addition to providing the basic conditional probability of error propagation over a given connector (conditioned on the activation of the connector), the author ahs also provided analytical formulas for unconditional error propagation (which incorporate the probability of connector activation) as well as the cumulative error propagation probability, which quantifies the probability that an error propagates from one component to another in an arbitrary number of connector activations. Finally he has briefly explored ways for a software architect to analyze and use the information provided by the analytical estimates.

4.5 Comparison to Related Work

Software metrics has been a very active field for several decades, and it is impossible to do justice to all the relevant work in this area. The author briefly discuss some of the research efforts that he finds most closely related to his work, highlighting in what way his work are different from the others. Many authors have used information theory to derive software metrics. Some, such as Allen and Khoshghoftaar [93], Chapin [95], and Harrison [94] do so explicitly; others, such as Halstead [20], do not invoke information theory explicitly, but their metrics can be interpreted in terms of this theory. Whereas all these authors define metrics by interpreting the source text of the program as the message,

the author derives metrics by interpreting information flow throughout the architecture as messages whose entropy the author is estimating.

Basili and Rombach [19] introduce an analytical paradigm for the derivation of software metrics, called the *Goal/Question/Metric* paradigm. It is based on a systematic, goal-oriented, procedure for the derivation of software metrics. In [96], Fenton presents a *Representational Theory of Measurement*, and argues that software metrics work must adhere to it. Also, he evaluates various metrics efforts with respect to the guidelines of this theory, and argues that Basili's *GQM* is but an instance of it. In the author work he combines the top-down goal oriented approach advocated by Basili and Rombach and Fenton with a bottom-up approach, which analyzes the architecture and produces a matrix of information theoretic measures, where each cell in the matrix is associated either with a component (for diagonal elements) or with a connector (outside the diagonal) in the architecture. These measures are used for the purposes of estimating error propagation.

In [97], Voas analyzes error propagations between COTS components and presents an automated tool to simulate error propagation, which is used to deploy a fault injection experiment. Michael et al. [98] present an empirical study of data state error propagation behavior. They argue that at a given location either all data state errors injected tend to propagate to the output, or else none of them does. In [14] Hiller et al. analyze error propagation conceptually, introducing the concept of *error permeability* and discussing means to measure it using fault injection techniques.

In [99], Geoghegan and Aversky propose a formal approach for adding fault detection to software. An assertion-based formalism is used to represent algorithm

specifications. This representation is then used to generate a flowgraph or ddgraph [?], which is used to construct an execution path tree. The information gained from this algorithm representation is used to aid in the design of software based fault tolerance techniques. [99]'s approach provide a software developer the opportunity of placing checks in an application, of determining the coverage, and possibly of adding more checks to the application until the desired level of coverage has been achieved. The approach of Geoghegan and Aversky can be used in conjunction with the author's work to make provisions for components with high error propagation values.

Using whitebox knowledge of a software artifact, [100] aims to reduce inter-modular error propagation by design. The main aims of [100] are to be able to calculate the influence values between a source module and a target module in the system, and to use these values to determine candidate modules for replication or to equip with error detection and error recovery mechanisms. That paper first analyzes the error propagation process, which provides the relevant information needed when performing fault injection experiments, i.e., which metrics to evaluate to allow calculation of influence. In this case, the metrics are error transmission probabilities and error transparency along a certain input set. That paper shows that the analytical framework can predict to a very high degree of accuracy the influence value between a pair of modules. Using this influence value (and associated metrics such as error transparency or error transmission probability), the authors are then able to derive probability estimates. Furthermore, the influence values provide insights into whether a system has been built too defensively. The work of [100] is close to the authors in terms of its goals, but quite distinct in its means (deterministic analysis) and its focus (finished software products).

In [101], Shin and Lin use a direct graph to represent a multi-module computing system and error propagation in the system is modeled by general distributions of error propagation times between all pairs of modules. Two algorithms are developed to systematically and efficiently compute the distributions of error propagation times. Experiments are also conducted to measure the distributions of error propagation times within the fault-tolerant multiprocessor (FTMP). Statistical analysis of experimental data shows that the error propagation times in FTMP do not follow a well-known distribution, the Weibull distribution, thus justifying the use of general distributions in their model. This paper has similar goals to ours, but use a different metric (propagation time rather than propagation probability) and a different focus (system structure rather than software architecture).

In [102], Iyer and Tang discuss methodologies and advances in the area of the experimental analysis of computer system dependability. Because the focus of the chapter is system dependability, the chapter covers several fault injection techniques, such as hardware-implemented fault injection, software-implemented fault injection, and radiation-induced fault injection. In this dissertation, the author's main concern is the error propagation probabilities; fault injection is merely a tool in performing his empirical experiment.

Powell et al. [103] address the problem of estimating the coverage of a fault tolerance mechanism through statistical processing of observations collected in fault injection experiments. A formal definition of coverage is given in terms of the fault and system activity sets that characterize the input space. Two categories of sampling techniques are considered for coverage estimation: sampling in the whole space and

sampling in a space partitioned into classes. The estimators for each technique are compared by means of hypothetical examples. Techniques for early estimations of coverage are then studied. These techniques allow unbiased estimations of coverage to be made before all classes of the sampling space have been tested. Then, the 'no-reply" problem that hampers most practical fault-injection experiments is discussed and a posteriori stratification techniques are proposed that allows the scope of incomplete tests to be widened by accounting for available structural information about the target system. This reference uses various sampling methods to estimate coverage factor (which is defined as: the probability of system recovery given that a fault exists). The author uses fault injection technique in his experiment but fault coverage is not a big concern for him.

PART III CHANGE PROPAGATION PROBABILITIES

CHAPTER 5

ANALYTICAL STUDY OF CHANGE PROPAGATION

In this chapter, the author studies a specific architecture-level attribute *Change Propagation Probability,* which reflects the probability that a change that arises in one component of the architecture (in the context of corrective/adaptive maintenance) requires changes to be done to other components. As stated before, this study is part of a larger project that investigates a wide range of architecture-level attributes, including *Error Propagation Probabilities, Requirements Propagation Probabilities, Diagonality, etc.*

In the spirit of the *Goal/Question/Metric* of Basili and Rombach [19], the author maps the attributes of interest onto a computable metric that can be evaluated on the basis of information that is available at the architectural level. At this level, one does not usually have the wealth of structural and semantic information that is available at the code level; instead, he only has information about the flow of control and data within components and between components. This precludes using traditional software metrics, which are based on such code-level features as tokens, flow graphs, data dependencies [16], and control dependencies. The approach can further be characterized by a combined *Bottom-Up/Top Down* discipline, whereby the author complements the top-down approach advocated by Basili and Rombach (in their Goal/Quality/Metric paradigm) with a bottom-up approach that analyzes the architecture and derives a matrix that quantifies the flow of information within and between components of the architecture. In the absence of detailed functional /behavioral information, one cannot analyze the flow of information within an architecture deterministically; hence, the author resorts to a

stochastic approach. This stochastic approach captures information flow within the architecture by means of random variables, and quantifies this flow by means of entropy functions applied to these random variables.

5.1 Background and Definition

If one is given a software architecture, modeled by components and connectors, and he is interested in the maintainability (or, conversely, the maintenance costs) of the products that can be instantiated from it, he may want to consider the *change propagation probability* matrix for this architecture, which reflects, for a given pair of components A and B, the probability that if A is changed in a corrective/perfective maintenance operation, he has to also change B. Whence the author propose the following definition for change propagation.

Definition 1. Given components *A* and *B* of a system *S*, the *change propagation probability* from *A* to *B* is denoted by CP (*A*, *B*) and defined as the following conditional probability:

$$CP(A, B) = Pr(([B] \neq [B']) \mid ([A] \neq [A']) \wedge ([S] = [S'])), \quad (5.1)$$

Where S' is the system obtained by changing *A* into *A'* (and, possibly, *B* into *B'* as a consequence).

In practice it is useful to add some qualifications to the above definition and distinguish between the *1-step* and *multi-step change propagation.* By the *1-step change propagation,* the author means the change propagation pertaining to individual connectors in the architecture, i.e., the *1-step change propagation* accounts for the change

propagating from one component to another directly as a result of one component using services (information) provided by another. The author denotes the *1-step change propagation* from *A* to *B* by $CP_1(A, B)$. For the *1-step change propagation* from *A* to *B* to be non-zero, it is necessary (but not sufficient) that they are adjacent nodes in the architecture graph, i.e., a connector must exist between them.

However, it is obvious that in some architectures a change may propagate between a pair of components even when they are not directly linked by a connecter: the change transmission may occur via a chain of changes in intermediate components. In order to account for this phenomenon, the author introduces the notion of *n-step change propagations*, which collectively (for all $n>2$) are referred to as *multi-step change propagations*.

The term *n-step change propagation* ($n>2$) refers to the probability of a change propagating from one component to another as a result of *n* consecutive acts of 1-step change propagation. The author denotes the *n-step change propagation* from *A* to *B* by $CP_n(A, B)$. For the *n-step change propagation* from *A* to *B* to be non-zero, there must exist in the architecture graph a *simple* directed path of length *n* that begins at *A* and ends at *B*. This implies in particular that for every $n \geq |S|$ (where $|S|$ is the number of components in the system *S*), the *n*-step change propagation between any pair of components is always zero, since every simple path in a graph has a length less than the number of nodes of the graph.

The author finds it both convenient and useful to represent (for any $n\geq1$) the *n*-th step change propagation probabilities for various pairs of components in a system *S*, in the form of a square ($|S| \times |S|$) matrix in which the entry at the intersection of the *i*-th

column and j-th column is $CP_n(C_i,C_j)$ $(1 \leq i,j \leq |S|)$, where C_i denotes the i-th component of the system.

5.2 Change Propagation: Usage

In addition to the many applications the author has briefly discussed in the introduction, he finds that the availability of change propagation probabilities of an architecture allows him to quantify an important classification of change propagations, first proposed by Clarkson et al. Clarkson et al. present a three-tiered classification of changes as follows:

1. Ripples of change: the introduced changes will result in an acceptable behavior to be observed for the maintenance process. Changes are controlled and limited.

2. Waves of change: the introduced changes will still result in an acceptable behavior to be observed for the maintenance process. Although there are many changes, they are under control.

3. Avalanches of changes: the introduced changes will result in an unacceptable behavior to be observed in the maintenance process. There are many changes and they are uncontrolled.

The avalanche type of change propagation is the one that software maintainers worry about most. Such avalanche changes would make managing the software maintenance very difficult and very costly. It would be a great benefit to be able to predict in advance if a certain change could cause this kind of avalanche of changes.

Because of its highly heuristic and qualitative nature, the above classification, while being conceptually useful, cannot be applied directly for an analytical study. In order to make *Clarkson's classification* more usable for the purposes of the quantitative analysis of change propagation, the author gives it a more rigorous quantitative

interpretation. Whereas Clarkson et al. present this classification to characterize individual changes; he uses it to characterize components. Specifically,

- The author considers that a component belongs to the *Ripple* class if, on average, the changes initiated in this component produce a ripple effect.
- He considers that a component belongs to the *Wave* class if, on average, the changes initiated in this component produce a wave effect.
- He considers that a component belongs to the *Avalanche* class if, on average, the changes initiated in this component produce an avalanche effect.

In order to give meanings to these concepts, the author must introduce some numeric parameters. The classification of any given change (or component) depends on what value the author gives to each parameter (these parameters can be "tuned" by the analyst according to the degree of change tolerance or sensitivity that he or she considers appropriate for the system under consideration):

- The *negligibility threshold* δ ($0< \delta <1$), indicates the level, below which the change propagation probability is considered negligible.
- The *ripple area coverage* ρ ($0< \rho <1$), determines the fraction of the total number of components affected by a ripple change propagation; ρ can be expressed as a fraction or as a percentile.
- The *avalanche area coverage* α ($0< \alpha <1$), determines the fraction of the total number of components that must be affected by avalanche change propagation; α could be expressed as a fraction.

Having chosen a value of δ, the author can, for each integer $n>0$ define the n-th step CP-graph $CPG_{n,\delta}$ of the architecture to be the subgraph of the original architecture graph G obtained by erasing in G all the edges (A,B) for which $CP_n(A,B)< \delta$. Notice that

the graph $CPG_{n,\delta}$ monotonically decreases as δ increases (for δ sufficiently close to 0, $CPG_{n,\delta} = G$, while increases for δ sufficiently close to 1, it is empty).

For any value of δ and n, one can associate with each component in the system an integer-valued CP-based metric. Since, as noted above, $CP_n(A,B) = 0$ for all $n \geq |S|$, the graph $CPG_{n,\delta}$ is empty for any $n \geq |S|$. Thus, henceforth, when speaking of the steps of change propagation, "for all $n \geq 0$" actually means "for all $n < |S|$".

Definition 2. The *n-th step CP range* of A (with sensitivity threshold δ), denoted by $M_{n,\delta}(A)$, is the out-degree of the node A in the graph $CPG_{n,\delta}$, i.e.,

$$M_{n,\delta}(A) = |\{B \in S \mid CP_n(A,B) > \delta\}|, \tag{5.2}$$

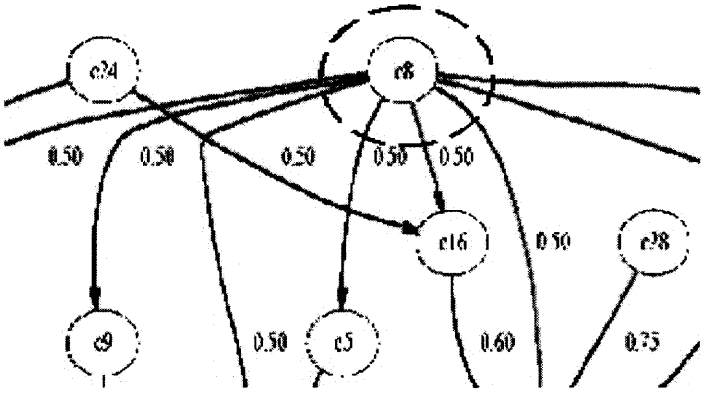


Figure 5.1 An example on how to calculate $Mn(C_8) = 8$.

If the author applies the definition on the part of the graph for a single-step change propagation for component C_8 presented in Figure 1 , he finds that $Mn(C_8) = 8$. Based on these metrics, one can interprets Clarkson's classification of CP behavioral patterns of the system according to the dynamics of $M_{n,\delta}(A)$ considered as a function of the step n. (Here, the author interprets the step of the unfolding process of change

propagation, as an analogue of the time into the maintenance cycle in Clarkson's classification.)

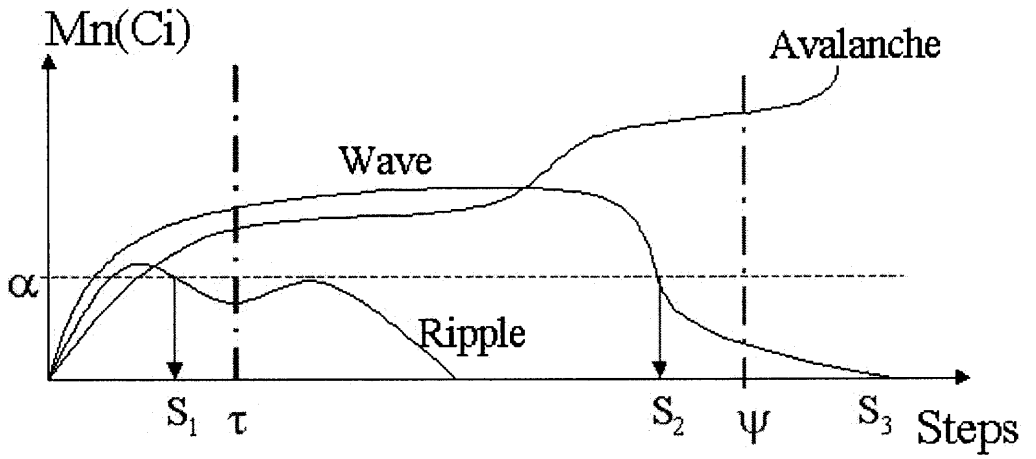
Definition 3. For any component A in the architecture S, one can say that A has a potential for generating

- *A ripple of changes, if* $M_{n,\delta}(A) < \rho |S|$ *for* $n=1,2$; *and* $M_{n,\delta}(A) = 0$ *for all* $n > 2$, *i.e., the first and second steps of change emanating from A affect not more than* $\rho|S|$ *other components, and all steps beyond the second do not affect (have negligible affect on) any other component;*

- *An avalanche of changes, if for some* n_a *such that* $M_{n_a,\delta}(A) \geq \alpha |S|$; *and* $M_{n,\delta}(A) \geq \rho |S|$ *for all* $n \geq 1$, *i.e., all steps of change emanating from A affect at least* $\rho |S|$ *of other components, and (at least) one step affects no less than* $\alpha |S|$ *of them (as noted above, it is enough to verify this condition for all* $n < |S|$ *only);*

- *A wave of changes, if neither of the two conditions above are satisfied (i.e., it is neither ripple nor avalanche). This case has intermediate severity between the ripple and the avalanche.*

Note that the type (severity) of change propagation behavior as defined above is relative to the choice of parameters: a component may be classified as having a potential for generating an avalanche of changes for a low value of δ, while it shows a potential for generating only a wave or a ripple of changes for a higher value of δ. An illustrative description for change propagation behavior is presented in Figure 5.2.



where:

α ($0 < \alpha < 1$) is the propagation area significance,

τ ($0 < \tau < 1$) is the ripple threshold, and

ψ ($0 < \psi < 1$) is the avalanche threshold

Figure 5.2 Parameterization of the categorization of the change behavior.

Using these parameters, the author can now characterize Clarkson's classification in terms of change propagation probabilities.

5.3 Analytical Approach

The analytical (as well as experimental) study of change propagation in its most general form, i.e., encompassing all kinds of changes that may be made in software components appears to be extremely difficult. The author suggests a rather simplified but viable classification of the types of changes into two major classes:

- *Interface changes*, i.e., those changes that affect the signature of the interface variables of the affected component; and
- *Functionality changes*, i.e., those changes that affect the "mapping" of the input/output data processing performed by the component, within fixed interfaces.

Of course, there may also exist more complex changes that involve a *combination* of both interface and functionality types of changes. In this study, the author addresses only the first class of changes – the *interface* changes. It is fairly obvious to the author that much (thought not all) of the methodology he has developed for the treatment of this case can also be used to deal with the propagation of functionality change. A similar study of functionality change propagation will be the subject of the future work.

5.3.1 Context and Assumptions

The author uses the unified modeling language UML to represent architectures. He selects UML because of its widespread use in industry.

In the study of change propagation, the author limits his scope to interface changes between components as the cause of changes. This assumption is justifiable since changes in any component, as long as they do not reach the interfaces, will remain local to that component.

5.3.2 Analytical Formula

In this section, the author first discusses single step change propagation, and then moves on to multi-step change propagation.

5.3.2.1 Single Step Change Propagation. In this section, the author introduces some formal notations and describes an analytical procedure for estimating the change propagation probabilities. The author views an architecture as a collection of components

C_i, i=1,…,N. With every component C_i he associates the set V_i of all the variables of all the provided functions of C_i. For every variable $v \in V_i$ and every other component C_j, $j \neq i$, he determines the *variable usage coefficient* value π_{vj}, which is a binary value: $\pi_{vj} = 1$, if the variable v (provided by C_i) is used by (required by) C_j; otherwise, he sets $\pi_{vj} = 0$. It is easy to see that any signature change in component C_i (no matter what particular "old" type is being replaced by what "new" one) associated with variable v will propagate to component C_j if $\pi_{vj} = 1$.

From this observation, it follows that for every pair of components C_i and C_j, $i \neq j$, *the (interface) change propagation* probability ICP(C_i, C_j) can be determined based on the table of the *variable usage coefficients* π_{vj} by the formula (5.3): i.e. by averaging all the v-to-j variable usage coefficients over all the variables v provided by C_i. The author remarks here that formula (5.3) is based on the assumption that an interface in C_i whose propagation he is trying to trace is equally likely to affect any of its interface variables, as shown in Figure 5.3.

$$ICP\ (C_i, C_j) = \frac{1}{|V_i|} \sum_{v \in V_i} \pi_{vj}\ , \qquad (5.3)$$

The method described above allows us to evaluate the (1-step) change propagation probabilities, given the information on the interface specification of the architecture. Once the 1-step CP values are obtained, the author can get upper-bound estimates on the multi-step CP.

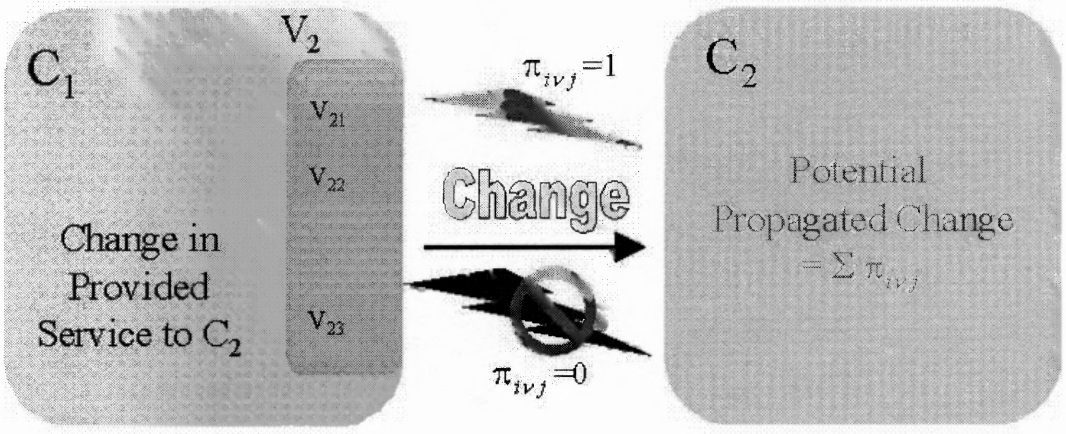


Figure 5.3 Single-step change propagation estimation.

5.3.2.2 Multi-Step Change Propagation. Suppose one has obtained (say, using the formula (7-3)) the 1-step interface change propagation matrix ICP he can derive multi-step change propagation matrix, for convenience, here the author uses shorter notations:

$$\Lambda(i, j) := \mathrm{ICP}(C_i, C_j), \quad i, j = 1,\ldots,N$$

In the directed graph G representing the architecture, let $\Pi_G(i, j)$ be the set of all simple (directed) paths leading from node (i.e. component) i to node j ($i \neq j$). For a path $\pi = (i, i_1, i_2,\ldots, i_{n-1}, j) \in \Pi_G(i, j)$, where $n = |\pi|$ is the length of π, let us denote by $\Lambda(\pi)$ the probability of a change in i propagating to j via the path π, i.e., the probability that a change in i causes a change in i_1, which in turn causes a change in i_2, etc., finally causing a change in j. If one makes the simplifying assumption that change propagation events in different connectors are independent, he obtains:

$$\Lambda(\pi) = \Lambda(i, i_1)\,\Lambda(i_1, i_2)\ldots\Lambda(i_{n-1}, j)$$

Let $\Lambda_k(i, j)$ be the probability of a change in i propagating to j in k steps ($k \geq 1$). It is easy to see that $\Lambda_k(i, j)$ is the probability of the union of the events that consist in the change propagating from i to j along particular simple paths of length k in G. Since the probability of a union is never greater than the sum of the probabilities of the constituent events, the author gets:

$$\Lambda_k(i,j) \leq \sum_{\substack{\pi \in \Pi_G(i,j) \\ |\pi|=k}} \Lambda(\pi), \tag{7.4}$$

The reason the author only considers *simple* paths is that he is interested in the *first* propagation of change to its destination.

Proposition 1:

$$\Lambda_k(i,j) \leq \tilde{\Lambda}^k(i,j),$$

where $\tilde{\Lambda}^k(i,j)$ is the kth power of matrix $\tilde{\Lambda}$, and $\tilde{\Lambda}$ is obtained from matrix Λ by setting its diagonal elements to zero, i.e., formally, $\tilde{\Lambda} = \Lambda - I$,

Proof: Using (7.4), one has

$$\Lambda_k(i,j) \leq \sum_{\substack{\pi \text{ is a simple path in } G, \\ \text{that starts at } i \text{ and ends at } j \\ |\pi|=k}} \Lambda(\pi) \leq \sum_{\substack{\pi \text{ is any path in } G, \\ \text{that starts at } i \text{ and ends at } j \\ |\pi|=k}} \Lambda(\pi) = \tilde{\Lambda}^k(i,j),$$

Note that if the author uses the original matrix Λ instead of $\tilde{\Lambda}$, the entries of its k-th power would also incorporate the paths of length *less than k*. Indeed, for example, the sum that defines $\Lambda^3(i,j)$ may include such members as $\Lambda(i, m)\,\Lambda(m, m)\,\Lambda(m, j)$ (where m

is some other component in G), which actually equals $\Lambda(i, m)\ \Lambda(m, j)$ (because all diagonal elements $\Lambda(m, m)$ are 1), and thus describes a simple path of length 2 from i to j.

5.3.3 Estimation Procedure

To compute the change probabilities, one needs the internal architecture of the system under investigation, as shown in a structure diagram such as in Figure 5.4. Using these artifacts, one can identify the components and the connectors that describe the component-based system architecture and can label the CP matrix rows and columns with the component names.

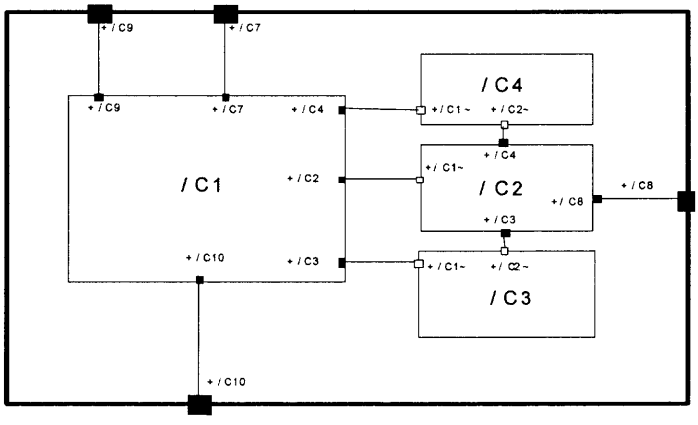


Figure 5.4 The architecture of a sample System.

Figure 5.5 shows a sample message protocol between a pair of components in a sample system. This artifact provides us with the sets $V_{A\to B}$ and $V_{B\to A}$ of messages between the two components A and B. Similarly, using the case tools the author can get, for any pair of components in the system, the sets of messages between them. The analyst is required to decide which of these sets of messages might propagate a change from component A to component B, according to the domain information and the design of the interfaces between the components.

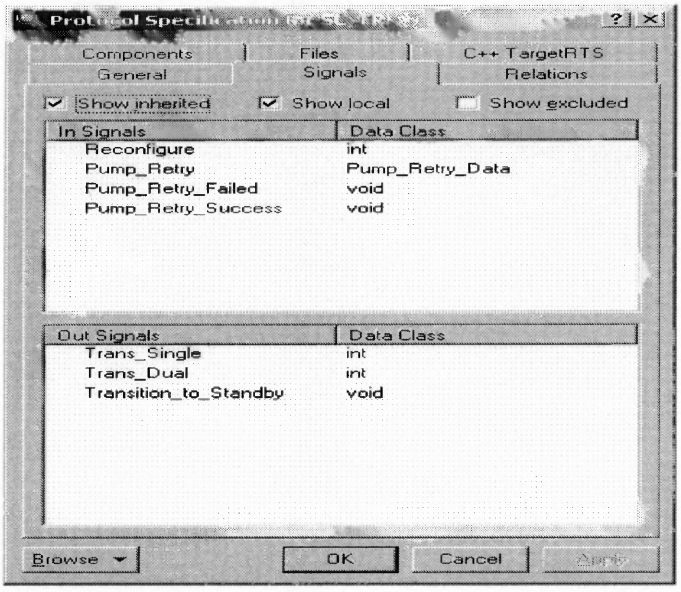


Figure 5.5 A sample of a message protocol.

5.4 Analytical Result

Using the interface specification of the automatic tool (see Chapter 6 for the tool details), the author does his analytical study on a sample system. The sample system is described in detail in Chapter 6. The author first determines the interface variables that have an effect on the neighboring components of the architecture of the system. Then, he analytically computes an estimated change propagation matrix for the system. This gives him an estimate of the probability that an interface change will propagate to a neighboring component due to that change.

The analytical result is presented in Table 5.1. To have a better understanding of the resulting matrix, the author produces a graphical representation of it. One can see that there is a significant amount of change propagation between pairs of components that have a small value. Due to this fact, the author sets a relatively high threshold value to have a better focus on more critical components that have larger values of change

propagation. This gives him a graphical representation of the critical change propagation of the system, as shown in Figure 5.7. In this example, the author lets the significance threshold be 0.4 in order to identify the more critical components of the system.

Table 5.1 Analytical Change Propagation Result for Sharp Tool

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16
C1	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.40	0.00
C2	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C3	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.25	0.00	0.00	0.25	0.00	0.00	0.00	0.00
C4	0.00	0.00	0.00	1.00	0.00	0.00	0.40	0.00	0.60	0.40	0.00	0.00	0.00	0.00	0.40	0.00
C5	0.00	0.00	0.00	0.00	1.00	0.40	0.00	0.00	0.00	0.40	0.00	0.00	0.00	0.00	0.00	0.00
C6	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C7	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
C8	0.00	0.00	0.00	0.00	0.50	0.50	0.00	1.00	0.50	0.00	0.00	0.50	0.00	0.00	0.00	0.50
C9	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
C11	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00
C12	0.04	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.15	0.04	0.00	0.00
C13	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	0.00
C14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
C15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00
C16	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.27	0.00	0.00	0.00	0.00	0.00	0.00	1.00
C17	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.00	0.00	0.00	0.00	0.00	0.00
C18	0.00	0.00	0.06	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.55	0.03	0.00	0.00	0.00
C19	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.30	0.00
C20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.50	0.00	0.00	0.00
C21	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
C22	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C23	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C24	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50
C25	0.00	0.00	0.00	0.00	0.00	0.33	0.00	0.00	0.00	0.00	0.33	0.00	0.00	0.00	0.33	0.00
C26	0.00	0.00	0.00	0.00	0.00	0.14	0.32	0.00	0.32	0.32	0.00	0.00	0.00	0.14	0.14	0.00
C27	0.00	0.00	0.00	0.00	0.00	0.00	0.07	0.00	0.22	0.12	0.00	0.05	0.03	0.00	0.00	0.08
C28	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C29	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C30	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C31	0.00	0.00	0.00	0.00	0.00	0.11	0.33	0.00	0.22	0.33	0.11	0.00	0.00	0.11	0.11	0.00
C32	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 5.1 Analytical Change Propagation Result for Sharp Tool (Continued)

	C17	C18	C19	C20	C21	C22	C23	C24	C25	C26	C27	C28	C29	C30	C31	C32
C1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.63	0.00	0.00	0.00	0.00	0.00
C3	0.00	0.25	0.00	0.00	0.00	0.00	0.00	0.25	0.00	0.00	0.25	0.00	0.00	0.25	0.00	0.00
C4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.20	0.00	0.00	0.60	0.00	0.00	0.40	0.00	0.20
C5	0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.30	0.60	0.00
C6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C7	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.00	0.00	0.50	0.50	0.00
C9	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.00
C10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.15	0.00
C11	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C12	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.54	0.00	0.00	0.00	0.00	0.00
C13	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.93	0.00	0.00	0.00
C16	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.13	0.00	0.00	0.20	0.07	0.00
C17	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C18	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.00	0.00	0.06	0.00	0.00
C19	0.30	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C20	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.00	0.00	0.00	0.00	0.00
C21	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C22	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00
C23	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C24	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.50	0.00	0.00	0.00	0.00	0.00
C25	0.33	0.00	0.00	0.00	0.33	0.00	0.00	0.00	1.00	0.38	0.00	0.33	0.00	0.00	0.00	0.00
C26	0.11	0.00	0.00	0.00	0.14	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.46	0.14	0.00
C27	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.10	0.00	0.00	1.00	0.00	0.03	0.16	0.08	0.00
C28	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.75	0.00	0.00
C29	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00
C30	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
C31	0.11	0.00	0.00	0.00	0.11	0.00	0.00	0.00	0.11	0.00	0.00	0.11	0.11	0.33	1.00	0.22
C32	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00

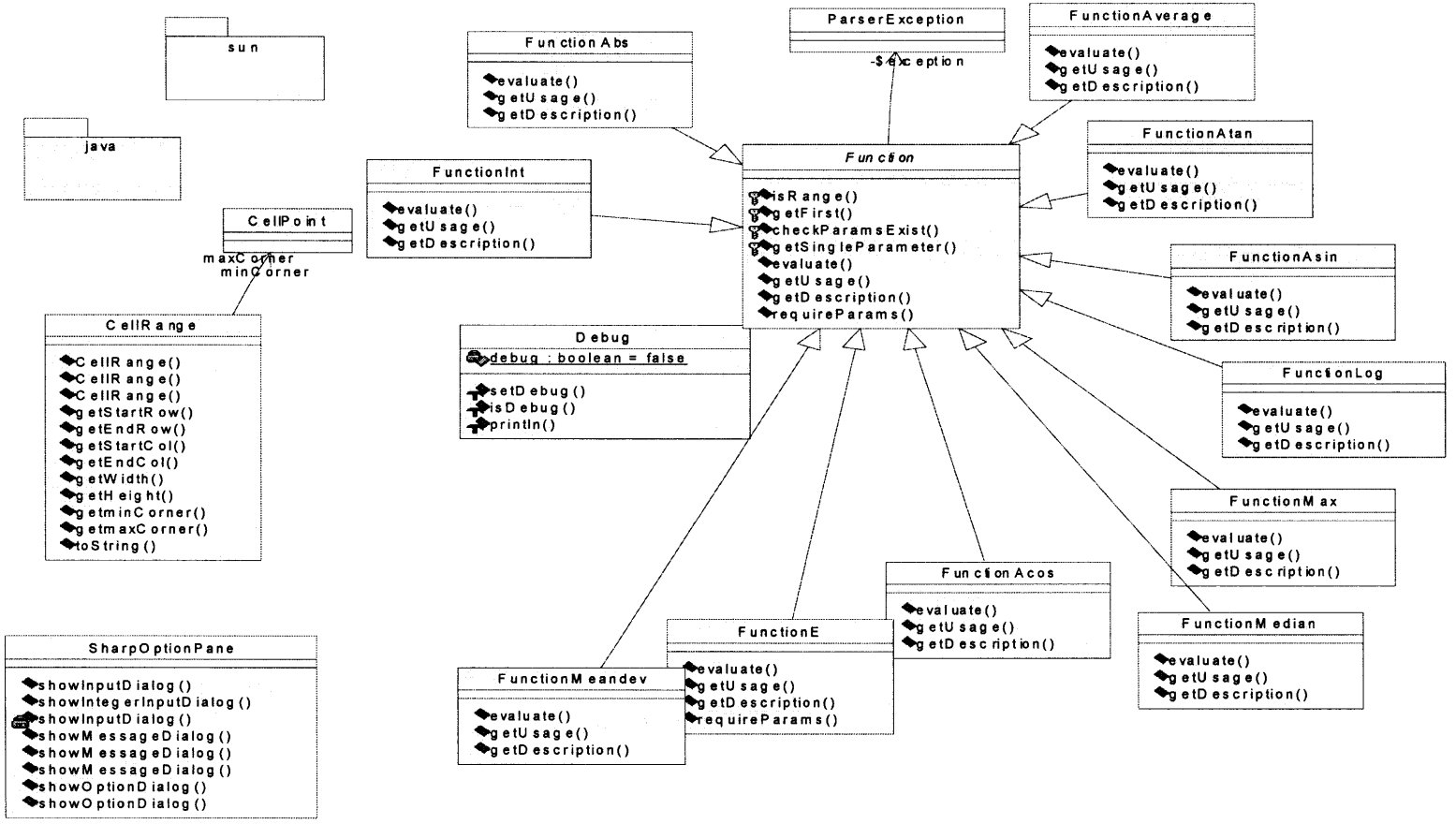


Figure 5.6 A reversed-engineered partial class diagram of Sharp tool.

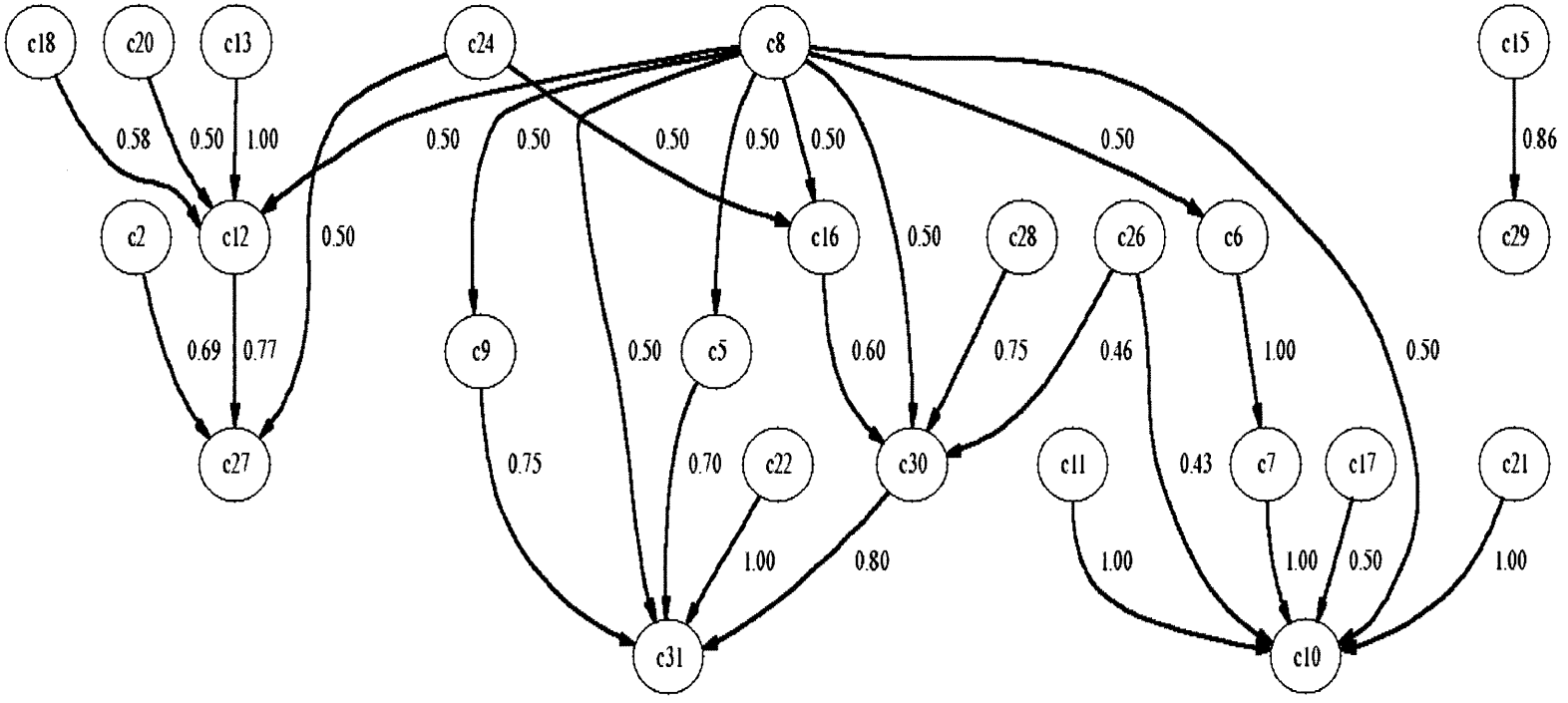


Figure 5.7 Graphical representation of the critical change propagation.

CHAPTER 6

CHANGE PROPAGATION EMPIRICAL EXPERIMENT

In this chapter, the author applies his view of change propagation to a sample case study. First, a brief description of the system under consideration is presented. Then, the aforementioned single-step change propagation on the system is applied. And then, the correlations between the analytical results and the observations from empirical study are examined. Next, the author expands the single step change propagation results to get a multi-step change propagation view of the same system. Using these results, the author tries to categorize the change propagation behavior of this system into ripple, wave, or avalanche.

6.1 Sample System

The system selected for the experiment is a spreadsheet application written in Java. It features full formula support (nested functions, auto-updating, and relative/absolute addressing), a file format compatible with other spreadsheets, printing support, undo/redo, a clipboard, sorting, data exchange with Excel, histogram generation, and a built-in help system. A partial class diagram of this tool was reverse-engineered to get a better understanding of the system (Figure 5.6).

6.2 Change Propagation Probabilities: Empirical Observations

In order to validate the analytical results, a controlled experiment on this system is performed. The author introduces interface changes on the interfaces of the components of the system. And with the help of the Java compiler and Understand for Java [85], the author tries to identify where the interface changes propagate in the system. That gives him experimental results for single-step change propagation in the system. From the single-step change propagation results, the author can get an estimate of the multi-step change propagation according to Section 5.2. The empirical observation is list in Table 6.1.

From Table 6.1, the author can see the rows corresponding to component C8, C26, and C31 have high values. This information means a change in these three components has a higher probability to require changes in other components to maintain the overall functionality of the system.

Table 6.1 also shows that columns corresponding to components C6, C9, and C10 have high values. This information means if a change has to make in the system these three components has high probability to get changed. These three components should be designed for easy modification.

Table 6.1 Experimental Change Propagation Result For Sharp Tool

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16
C1	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.20	0.00
C2	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C3	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C4	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C5	0.00	0.00	0.00	0.00	1.00	0.40	0.00	0.00	0.00	0.40	0.00	0.00	0.00	0.00	0.00	0.00
C6	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C7	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
C8	0.00	0.00	0.00	0.00	0.50	0.50	0.00	1.00	0.50	0.50	0.00	0.50	0.00	0.00	0.00	0.50
C9	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
C11	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00
C12	0.04	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.19	0.04	0.00	0.00
C13	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	0.00
C14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
C15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00
C16	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.27	0.00	0.00	0.00	0.00	0.00	0.00	1.00
C17	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.00	0.00	0.00	0.00	0.00	0.00
C18	0.00	0.00	0.06	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.58	0.06	0.00	0.00	0.00
C19	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.40	0.00
C20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.00	0.00	0.00	0.00
C21	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
C22	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C23	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C24	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50
C25	0.00	0.00	0.00	0.00	0.00	0.14	0.00	0.00	0.00	0.00	0.14	0.00	0.00	0.00	0.14	0.00
C26	0.00	0.00	0.00	0.00	0.00	0.22	0.38	0.00	0.30	0.43	0.00	0.00	0.00	0.11	0.11	0.00
C27	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.00	0.16	0.08	0.00	0.05	0.03	0.00	0.00	0.08
C28	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C29	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C30	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C31	0.00	0.00	0.00	0.00	0.00	0.11	0.33	0.00	0.22	0.33	0.11	0.00	0.00	0.11	0.11	0.00
C32	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 6.1 Experimental Change Propagation Result for Sharp Tool (Continued)

	C17	C18	C19	C20	C21	C22	C23	C24	C25	C26	C27	C28	C29	C30	C31	C32
C1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.69	0.00	0.00	0.00	0.00	0.00
C3	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.10	0.00	0.00	0.10	0.00	0.00	0.30	0.00	0.20
C5	0.10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.20	0.70	0.00
C6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C7	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.50	0.00
C9	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.75	0.00
C10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.23	0.00
C11	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C12	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.77	0.00	0.00	0.00	0.00	0.00
C13	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.30	0.00
C15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.86	0.00	0.00	0.00
C16	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.13	0.00	0.00	0.60	0.20	0.00
C17	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C18	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.00	0.00	0.06	0.00	0.00
C19	0.30	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.20	0.00	0.00	0.00	0.00	0.00	0.00
C20	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C21	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C22	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00
C23	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C24	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.50	0.00	0.00	0.00	0.00	0.00
C25	0.14	0.00	0.00	0.00	0.14	0.00	0.00	0.00	1.00	0.29	0.14	0.00	0.00	0.00	0.00	0.00
C26	0.11	0.00	0.00	0.00	0.22	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.46	0.11	0.00
C27	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.05	0.00	0.00	1.00	0.00	0.00	0.19	0.08	0.00
C28	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.75	0.00	0.00
C29	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.26
C30	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.80	0.00
C31	0.11	0.00	0.00	0.00	0.11	0.00	0.00	0.00	0.11	0.00	0.00	0.11	0.11	0.33	1.00	0.22
C32	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00

6.3 Correlation between Analytical Results and Empirical Results

In this section, the author shows the correlation between the matrices C_A from analytical calculation and the matrices C_E from the empirical observations mathematically. The correlation coefficient between all the cells of the analytical single-step matrix and the experimental single-step matrix is:

$$Cor(C_A,C_E) = 0.93 \qquad \text{(r value)}$$

$$R\text{-}Sq = 0.8649,$$

Where "r" denotes the Pearson product-moment correlation coefficient.

For the nontrivial values the author finds:

$$Cor(C_A,C_E) = 0.85 \qquad \text{(r value)}$$

$$R\text{-}Sq = 0.7225$$

A *significant* relationship between some variables does not necessarily mean that the relationship is very useful in building predictive models. Thus the R-Sq values are also shown above to assess the explanatory power of each model.

6.4 Statistical Significance of the Correlations

Now the author needs to validate the correlation results, i.e., to make sure that the positive correlation values observed are statistically significant (did not occur by chance). To test the relationship between analytical and experimental change propagation, a statistical hypothesis testing was performed using the *t*-test (one-tail) for the nontrivial entries using the level of significance $\alpha = 0.05$.

The hypotheses are

- $H0 : \rho = 0$
- $H1 : \rho > 0$

Where ρ denotes the correlation coefficient.

The author has computed the value of the t statistic for the nontrivial values of single-step matrices as t_{ob} = 18.98632 (with n=140 samples), and the corresponding P value is less than $\alpha = 0.05$. Thus, the author rejects the null hypothesis of no correlation, and thus infer that the correlation of 0.85 is statistically significant.

The t-test results showed that the fairly high correlation values between analytical and experimental change propagation values the author obtains do not occur by chance (i.e., are statistically significant).

6.5 Multi-Step Change Propagation Matrix

Form the single-step change propagation results, one can get an estimate of the multi-step change propagation according to Chapter 7 Section 7.1.The author can then estimate the outgoing change propagation of a component as the total change propagation that is exported by this component to other components. He can track the outgoing change propagation as change propagates in multi-steps, and observe the behavior of this component. Thus, he can categorize the components according to their outgoing change propagation as ripple, wave or avalanche.

Ripple components are those having outgoing change propagation that dies out rapidly with very few steps of change propagation. Wave components are those that

sustain a large value of outgoing change propagation for a number of steps, but this value dies out eventually. Avalanche components are those that have an increasing value of outgoing change propagation as the number of steps of change propagation increase, and this value does not die out eventually. From the experimental results, the author can recognize the following patterns of outgoing change propagation for the three kinds of components. This is shown in Figure 6.1.

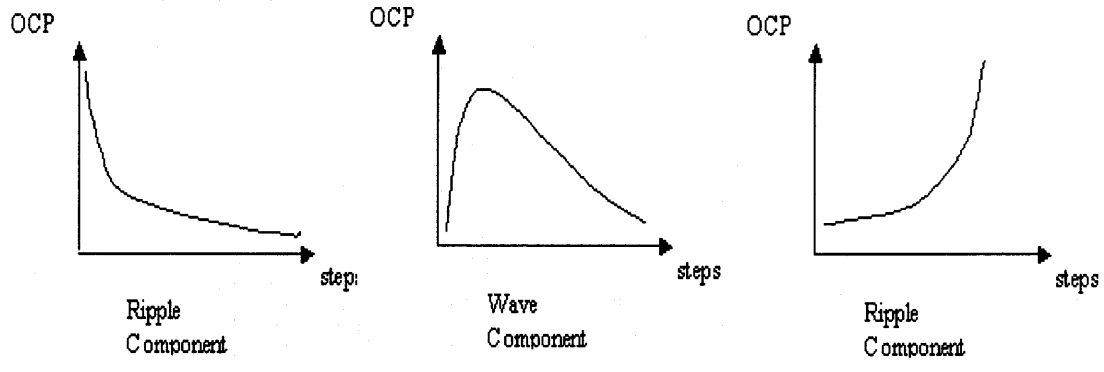


Figure 6.1 Different Components OCP behavior Observed during experiments.

The Mn(Ci),which is directly proportional to outgoing change propagation, for each component in the architecture is shown in Figure 6.2. The author can see that there are only ripple and wave components and no avalanche components. Based on the observation, he can expect that, when making a change in this system, he can recognize ripple change propagation or at most a wave of change propagation. But, it is highly unlikely to have an avalanche change. One can recognize that any change for component C8 should be handled with care, as it is a highly change propagating component. Any maintenance effort that might be needed to deal with this component should be expected

to cause a wave of changes since this is a highly centralized component that might affect the others much by its high mutual dependencies. Then, component C8 is the most critical component in context of any maintenance process involving it.

Three different patterns of change propagation in the analysis can be seen. Figure 6.3 shows a pattern for ripple components where the Mn(Ci) tends to decay in very few steps. When checking Figure 6.4, the author finds a pattern of a potential avalanche component. The Mn(Ci) still have a significant magnitude over a large number of steps. In Figure 6.5, one can recognize a pattern of wave components that are midway between the ripple and the avalanche components.

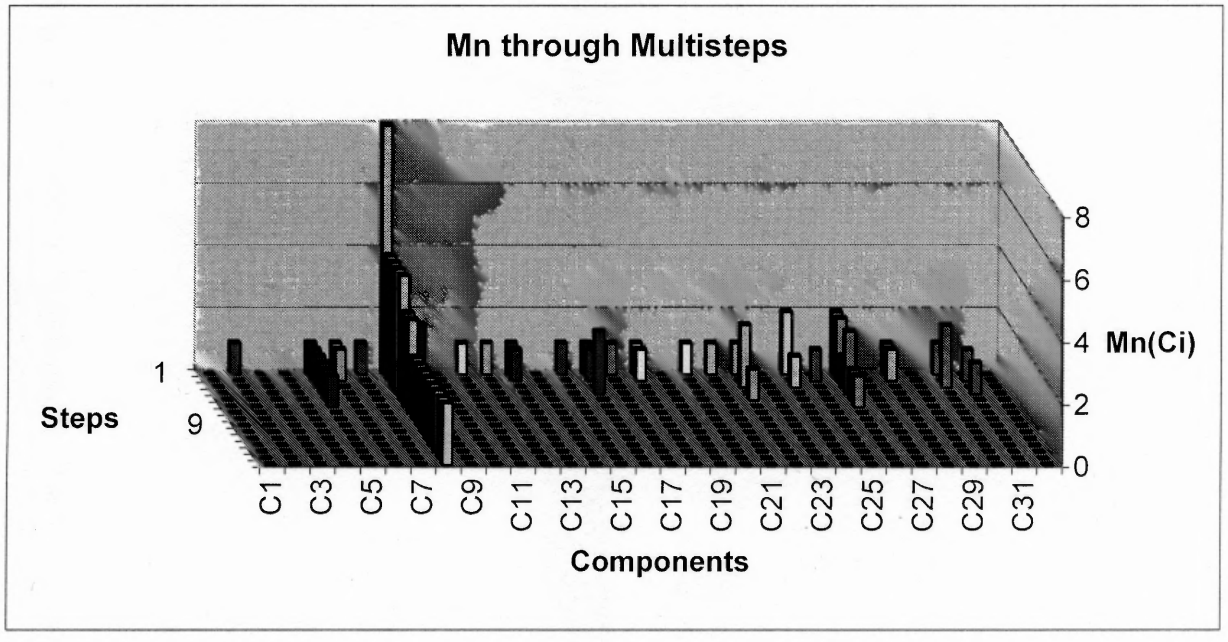


Figure 6.2 Mn(Ci) of the components through multi-step change propagation.

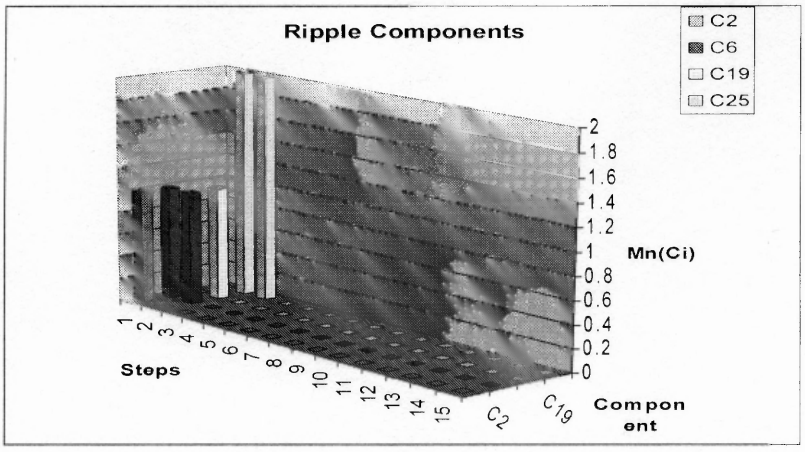


Figure 6.3 Pattern of Ripple components.

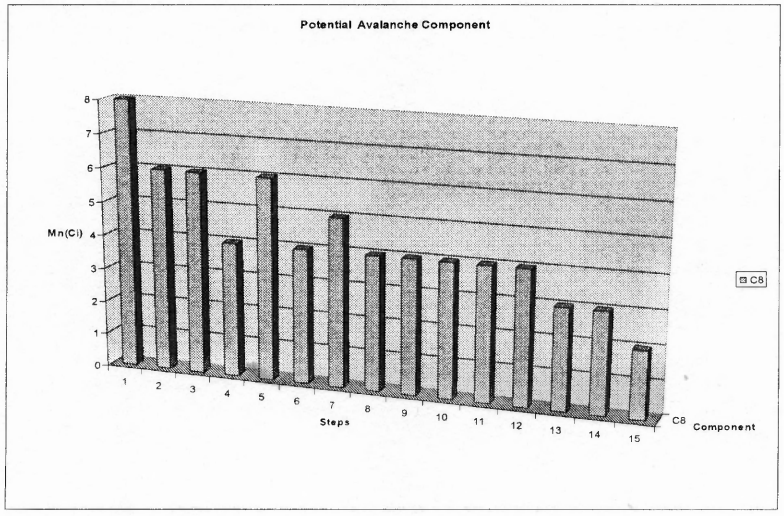


Figure 6.4 Pattern of a potential Avalanche component.

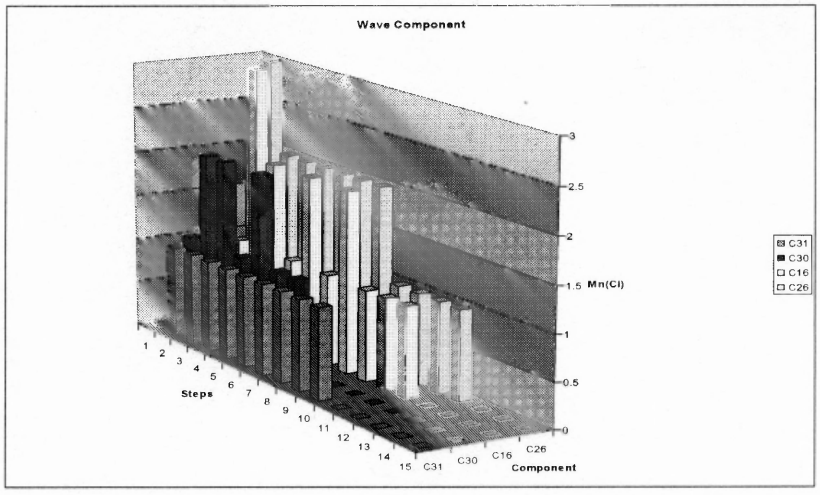


Figure 6.5 Pattern of Wave components.

6.6 Related Work

In the software engineering field, change propagation in evolutionary development is handled by a few models [104, 105]. In these models, a program is divided into parts that are used to construct a propagation graph. These techniques focus upon the potential necessary changes that are required in redesigning the software. They use program variables to locate links that might propagate the change, but they only predict one step of change at a time.

In the computer-aided mechanical design field, engineers use tools, such as C-FAR system [106], to trace and predict change propagation. They divide the system under investigation into parts that are described according to their attributes. Interactions of the attributes are stated in semi C-FAR matrices. Then, the interactions are analyzed to predict the mutual effect between the attributes. C-FAR's computational complexity makes it appropriate for small or relatively simple products.

Also, Design Structure Matrices (DSMs) [107] can estimate how change would

propagate in a system. They are well-established techniques used to identify relationships between the components of the system or the design tasks [108]. High connectivity found between design tasks suggests that high levels of dependency will exist between the resulting system components. No indication such as the probability or scale of any such redesign is given by the DSMs.

The work discussed in [109] is concerned with the prediction and management of changes to an existing product resulting from faults or new requirements. It develops mathematical models to predict the risk of change propagation in terms of likelihood and impact of change.

In [110], L. Briand et al. propose a UML model-based approach to impact analysis that can allow early decision-making and a change planning process. They first make a consistency check for UML diagrams. Then they identify changes between two different versions of a UML model according to change taxonomy, and determine model elements that are impacted by changes using defined impact analysis rules. They prioritize the results of impact analysis according to the likelihood of occurrence using a measure of distance between a changed element and potentially impacted elements. They also present a prototype tool that provides automated support for their impact analysis strategy.

6.7 CASE Tool

To assist the empirical analysis of the proposed metrics, the author has developed a web-based CASE tool that automates the capturing, modeling, and inspection of software

architectures, in order to derive and display the proposed metrics. At this point, the CASE tool only calculates Change Propagation metrics.

6.7.1 Functional Description

The process of deriving the proposed metrics through the CASE tool consists of a 5-step approach centered on the analyst uploading the software architecture's metadata, configuring the resulting model, and finally validating and purifying this model before it is analyzed by the tool's metrics layer.

First, through a user-friendly web-interface, the tool enables analysts to upload architectural artifacts (i.e., source code) for the software they wish to analyze. The CASE tool subsequently parses this input for metadata consisting of components, datatypes, and connectors, and generates a corresponding MOF-based model in a centralized repository, as shown in Figure 6.6.

Once the initial model has been generated, the analyst is further presented with various configuration options for the model. These options allow the analyst to specify grouping levels for the model's various component and datatype artifacts. The tool supports a *standard* grouping, which applies the original grouping semantics extracted from the software being analyzed, as well as a *custom* grouping, where the analyst explicitly assigns datatypes to specific components within the model. In addition, the analyst has the option of altering metadata for orphan or unknown types, that is, those datatypes and components whose definitions are absent from the initial software architecture's input.

With the model configured and refined by the analyst, the tool can now display a set of hyperlinks that represent various metric-specific multi-dimensional views of the

software's architecture. Each view displays the relationship between the model's components and their corresponding metric values. In addition, each view can be further refined and analyzed through a rich set of filtering tools.

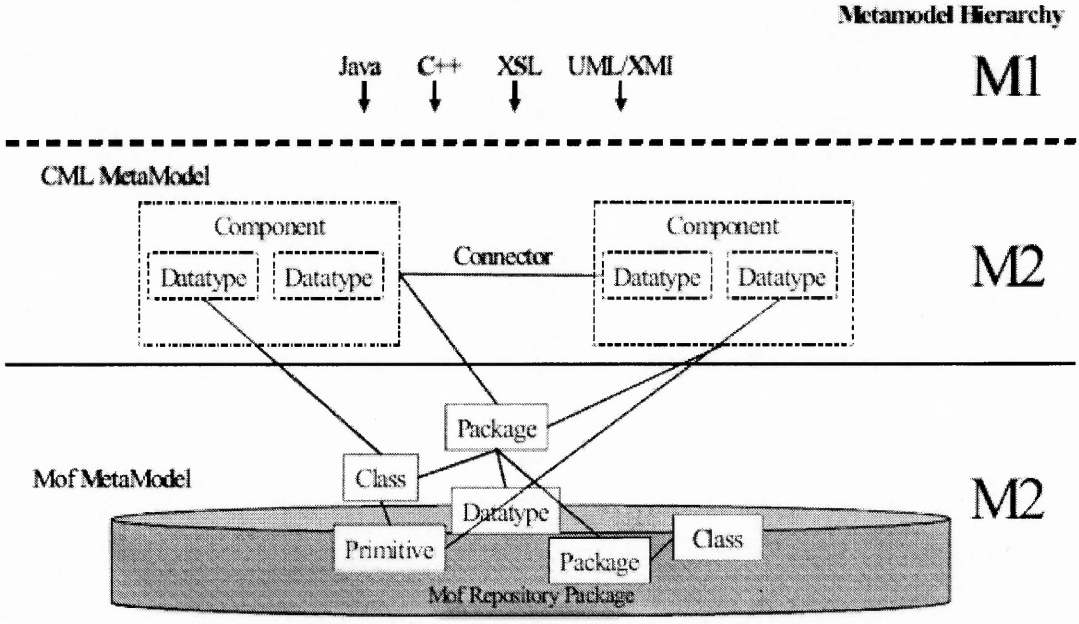


Figure 6.6 SACPT metamodel (CML/MOF metamodel).

6.7.2 Structural Description

The CASE tool's technology can be partitioned into three logical layers, as shown in Figure 6.7. First, a *metamodel* layer captures, models, and stores a software architecture's component, datatype, and connector information. Second, a *metrics* layer inspects the generated model and applies the algorithms defined in Chapter 7 to derive the corresponding metrics. Finally, a web-based *user-interface* layer displays the metric results to the analyst. Each of these layers is implemented as a series of software

component libraries written in C# on the Microsoft .NET™ platform. In addition, the metamodel layer is consistent with the *OMG Meta-Object Facility (MOF™).

Currently the CASE tool supports two different input formats, Java Understand [68] and UML, to produce the change propagation matrix. In the future the author and the group members are planning on adding more input format types that the CASE tool supports. The CASE tool is available online at the following URL: http://www.ccs.njit.edu/swarch/tools.htm

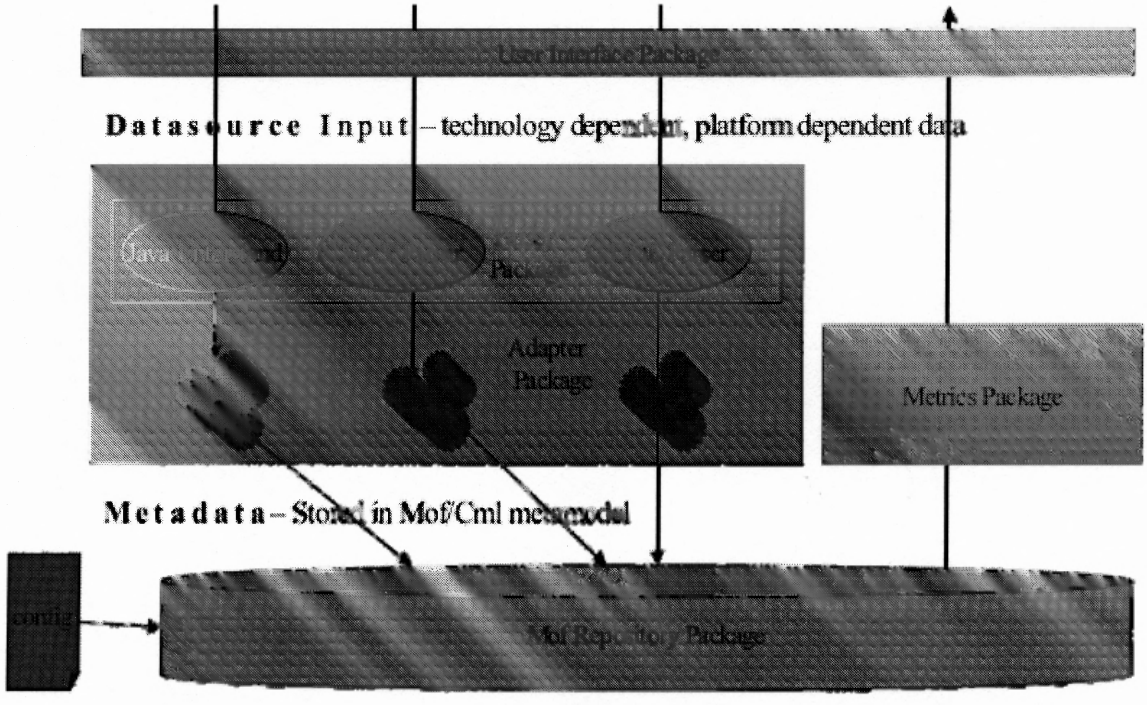


Figure 6.7 SACPT architecture.

* MOF is an extensible model driven integration framework for defining, manipulating and integrating metadata and data in a platform independent manner. MOF-based standards are in use for integrating tools, applications and data.

PART IV REQUIREMENTS PROPAGATION PROBABILITIES

CHAPTER 7

REQUIREMENTS PROPAGATION PROBABILITY

In this chapter, the author studies another type of propagation, requirements propagation. The author believes this attribute is highly related to the modifiability of software systems.

7.1 Background and Definitions

The subject of this chapter is the propagation of requirements changes in a software architecture viewed as an aggregate of components and connectors. Requirements propagation probability represents the expected changes to component *A* stemming from requirements evolution in the context of an adaptive maintenance. Those evolutionary operations of *A* will cause changes to a neighboring component *B* in order to evolve the overall function of the system. In this chapter, the author investigates the analytical means to estimate these probabilities from architectural information, and he explores ways to validate the analytical formulas by means of empirical experiments.

Software systems are expected to evolve and undergo modifications on a continuous basis during their lifecycles. Studies show that 50-70% of the total lifecycle cost of a software system is spent on modifications after initial development [38]. The author defines the *requirements propagation probability* from component *A* to component B as the probability that a change in component *A* mandated by a requirement will cause a change in component *B*, as formally defined in Equation 7.1. This definition can be used to estimate the change impact in an architecture and also can be used to

evaluate the quality of a given architecture.

$$\mathbf{RP}(A,B) = \mathbf{P}\,(B \neq B' \mid ([A] \neq [A']) \wedge (S \neq S')), \tag{7.1}$$

Where *S* is the overall system of which *A* and *B* are components. For an architecture having N components, the author envisions organization of the requirements change propagation probabilities in a square N × N matrix.

7.2 Propagation Types

As discussed in Chapter 6, the changes can be classified into three categories: ripple, wave and avalanche changes. The avalanche type of change propagation is the one that software maintainers worry about most; such changes would make managing the software maintenance very difficult and very costly. It would be a great benefit to be able to predict in advance if a certain change could cause this kind of avalanche of changes. The author can use the estimates of requirement propagation to identify which type of propagation is taking place.

7.3 Related Work

A lot of research has been done on change impact analysis [24, 38, 44, 65-67], which is related to the author's concept of requirements propagation probabilities. Most of this research used manually collected data; the data is either from documentation or from technical personnel. Bengtsson et al. [38] discusses how to select scenarios to get a good estimation of change cost. Lock and Kotonya [44] use traceability and dependency information to construct the complete change propagation path structure. Rajlich [69]

models the change propagation of a system by graph rewriting. Deruelle et al. [70] use similar techniques to model multi-language source code and heterogeneous database system.

7.4 Usage of Requirements Propagation

Given a matrix of requirements change propagation probabilities, one may want to use it in the following manner:

- A summary inspection of the matrix can reveal quickly the difficulty and the cost of requirements change operations on the system. An ideal system is perfectly modular and has an identity matrix. This characteristic is because each change is localized to the component where it is applied and does not propagate to other components. The closer a matrix is to this ideal case, the better.

- If the row corresponding to a component A has high values, the author infers that changes to this component must be avoided because they propagate widely throughout the system. Preventive measures include focusing verification and validation activity on this component (to minimize subsequent corrective maintenance), and optimizing the design of this component (to minimize subsequent perfective maintenance).

- If the column corresponding to a component A has high values, the author infers that this component is likely to undergo frequent changes in the maintenance phase. Preventive measures include special care to design this component for ease of modification.
- This matrix of requirements change propagation probabilities can also be used to compare candidate system architectures when change impact is an important consideration

7.5 Analytical Study

7.5.1 Analytical Approach

The author anticipates that functional dependency is strongly related to requirements change propagation. He proposes using any of the following three metrics to predict requirements change propagation.

Backward functional call Dependency (BD) metric: this metric reflects distribution that the services in component *i* are used by its neighbors. The author defines BD(j, i) as the total times that component *j* calls functions (or services) in component *i* divided by the total times that functions (or services) in component *i* are called by all components, as shown in Equation 7.2.

$$BD(C_i, C_j) = \frac{RC(j,i)}{\sum_j RC(j,i)} \tag{7.2}$$

Where RC (j, i) is the total number of times that component *j* calls functions in component *i*. The denominator is the total number of times that functions (or services) inside component *i* are called by other components.

Forward functional call Dependency (FD) metric: this metric reflects the distribution that component *i* uses the services from other components. The author defines FD(j, i) as the total times that component *i* calls functions in component *j* divided by the total times that component *i* calls functions in other components, as shown in Equation 7.3.

$$FD(C_i, C_j) = \frac{RC(i,j)}{\sum_i RC(i,j)}, \tag{7.3}$$

Total of forward and backward functional call Dependency (TD) metric: This metric combines the previous two. It reflects the total of Fan-in and Fan-out, as shown in Equation 7.4

$$TD(C_i, C_j) = \frac{RC(i,j) + RC(j,i)}{\sum_i RC(i,j) + \sum_j RC(j,i)}, \quad (7.4)$$

7.5.2 Analytical Results

Using the sample system described in next chapter, the author has derived the three matrices using the equations 7.2, 7.3, and 7.4, respectively. It can be seen the rows corresponding to component C2 and C7 have high values. To the author it means if a requirement of C2 or C7 has to be changed, this change has a high probability to require changes to other components. The architects of this system should pay extra attention to these two components. The architects should make effort to minimize the changes to these two components at design time.

It also can be seen that the column corresponding to component C2 and C7 have high values. To the author this information means that these two components have high probability to go through changes during maintenance time. The architects of this system should make effort to design these two components for easy modification.

Table 7.1 Forward Functional Dependency Matrix

	C1	C2	C3	C4	C5	C6	C7
File (C1)	1.000	0.432	0.227	0.000	0.023	0.000	0.341
Table (C2)	0.000	1.000	0.532	0.092	0.009	0.193	0.073
TableOp (C3)	0.000	0.382	1.000	0.000	0.088	0.059	0.206
Edit (C4)	0.000	0.643	0.857	1.000	0.357	0.071	0.357
Chart (C5)	0.000	0.291	0.139	0.114	1.000	0.000	0.152
Function (C6)	0.000	0.214	0.000	0.000	0.000	1.000	0.000
GUI (C7)	0.100	0.250	0.150	0.150	0.250	0.050	1.000

Table 7.2 Backward Functional Dependency Matrix

	C1	C2	C3	C4	C5	C6	C7
File (C1)	1.000	0.000	0.000	0.000	0.000	0.000	0.640
Table (C2)	0.162	1.000	0.105	0.210	0.238	0.257	0.029
TableOp (C3)	0.273	1.000	0.618	0.164	0.327	0.000	0.309
Edit (C4)	0.000	0.417	0.000	1.000	0.417	0.000	0.333
Chart (C5)	0.016	0.016	0.049	0.098	1.000	0.000	0.164
Function (C6)	0.000	0.145	0.016	0.016	0.000	1.000	0.040
GUI (C7)	0.654	0.346	0.462	0.308	0.423	0.000	1.000

Table 7.3 Total Functional Dependency Matrix

	C1	C2	C3	C4	C5	C6	C7
File (C1)	1.000	0.432	0.227	0.000	0.023	0.000	0.705
Table (C2)	0.156	1.000	0.633	0.294	0.239	0.440	0.101
TableOp (C3)	0.221	0.500	1.000	0.132	0.309	0.029	0.353
Edit (C4)	0.000	0.583	0.500	1.000	0.417	0.042	0.792
Chart (C5)	0.013	0.304	0.177	0.190	1.000	0.000	0.278
Function (C6)	0.000	0.218	0.016	0.016	0.000	1.000	0.040
GUI (C7)	0.722	0.538	0.577	0.423	0.615	0.038	1.000

7.6 Empirical Experiment

The author conducts an empirical study by changing the requirements of a system and tracing how these changes propagate throughout the architecture. By making many such observations, he can estimate requirements propagation metrics that correlate with the requirements propagation probabilities.

7.6.1 Sample System

The author uses the same system which is used for Change propagation empirical study to study requirements propagation empirically. As stated in Chapter 8 the system the author has used is a spreadsheet written in Java. It features full formula support (nested functions, auto-updating, and relative/absolute addressing), a file format compatible with other spreadsheets, printing support, undo/redo, a clipboard, sorting, data exchange with Excel, histogram generation, and a built-in help system. The author obtains the requirements and architecture of the system by reverse engineering the source code, which was written in Java.

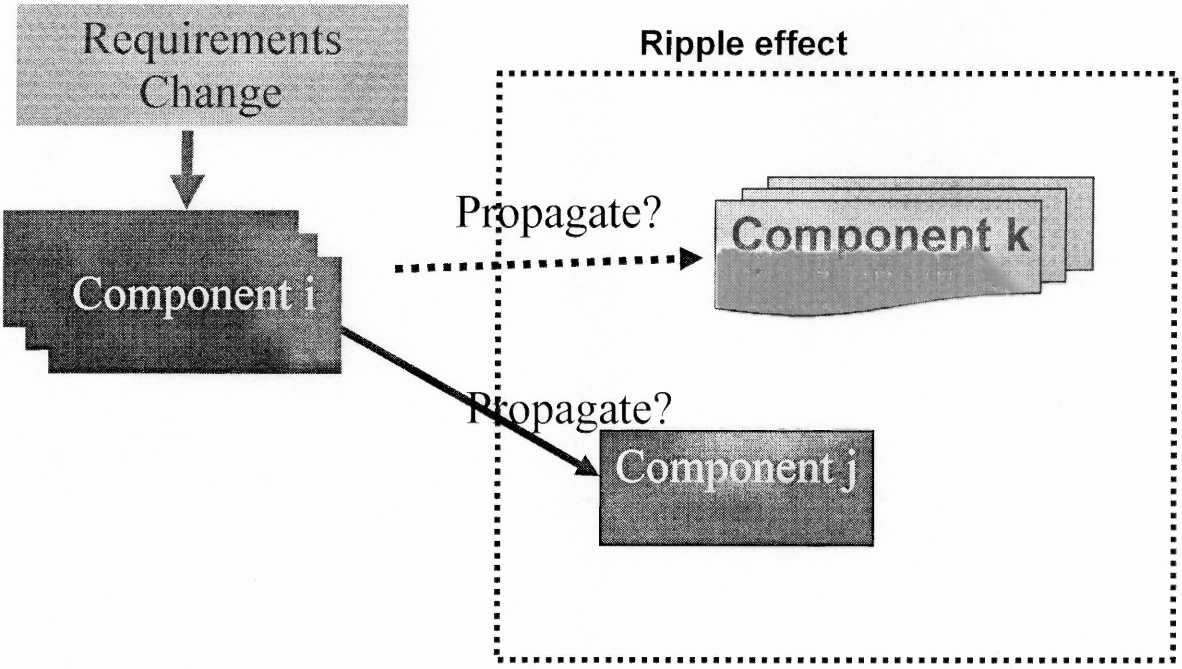


Figure 7.1 Requirements change propagation diagram.

7.6.2 Empirical Experiment Procedure

The basic procedure the author follows is to make a requirements change and map this change to a component/components (called "source" component(s)). He makes the necessary changes in the "source" component. Then check all the neighbors that require necessary changes. The following steps describe the procedure to make requirements changes in the system. One can make as many changes as he wants but the author argues the best results can be achieved if he only selects one change from each type of changes as discussed in [38].

Experiment procedure:

1. Make a requirements change
2. Map the change to a component(s)

3. Find out where the functionality is carried out in the "source component".
4. Check/Guess what part needed changes
5. Check all the neighbors to see any of them need changes to adopt the changes in the "source" component(s).

7.6.3 Empirical Results

The result of the experiment is an N × N matrix (N is the total number of components in the system). If one makes *m* changes to component *i* and there are *k* times the changes propagated to component *j*, in the propagation matrix cell (C_i, C_j) = k/m. The result is shown in Table 7.4.

Table 7.4 Experiment Result of Requirement Change Propagation

	C1	C2	C3	C4	C5	C6	C7
File (C1)	1.000	0.250	0.000	0.000	0.000	0.000	0.250
Table (C2)	0.000	1.000	0.000	0.000	0.000	0.500	0.500
TableOp (C3)	0.000	0.750	1.000	0.000	0.000	0.000	0.750
Edit (C4)	0.000	0.571	0.000	1.000	0.000	0.000	0.286
Chart (C5)	0.000	0.000	0.000	0.000	1.000	0.000	0.250
Function (C6)	0.000	0.333	0.000	0.000	0.000	1.000	0.333
GUI (C7)	0.125	0.000	0.000	0.125	0.000	0.000	1.000

From Table 7.4, the author can see the same pattern as the analytical matrices. The rows corresponding to component C2 and C4 have high values. This information means the changes in these two components have high probabilities to require changes in other components. From Table 7.4, the author also can see high values in the column corresponding to component C2. This information means component C2 has high probability to go through changes during maintenance time. Component C2 should be designed for easy modification.

7.7 Correlation between Analytical and Empirical Results

In this section, the author tries to validate the analytical result using empirical observations. The correlation coefficients between the matrix obtained from the experiment and the analytical matrices from the architecture are listed in Table 7.5.

Table 7.5 Correlation Coefficients

Matrix 1	Matrix 2	Coefficient
BD	Exp	0.812
FD	Exp	0.814
TD	Exp	0.755

The table shows that the correlation coefficients are significant. The author can conclude that the three analytical matrices are highly correlated to the experimental requirements change propagation matrix.

7.8 Conclusion

In this chapter, the author has investigated analytical means to estimate requirements propagation metrics from architectural information. He has conducted a controlled experiment to validate the proposed analytical formulas. Furthermore, he has performed a case study to observe, explore and validate hypotheses pertaining to the requirement propagation phenomena.

The empirical experiments conducted on two case studies show that requirements changes propagate prevalently to components that are directly *coupled* to the source or originating component. Furthermore, these studies show that functional dependency

metrics may not highly correlate with requirements propagation metrics. However further empirical studies are needed to support these hypotheses.

Requirements propagation probability depends on two factors. Probability of a requirement change affecting a component and the probability that this change will propagate to other components in the system. The work here focuses on the probability of a requirement change affecting other components in the system. The author's current and future work involves conducting further empirical studies. Also of interest is determining to what degree a requirement change affects a component by examining how stable the requirements for that component are determined.

PART V EXTENSIONS

CHAPTER 8

QUANTIFYING ATTRIBUTES OF PRODUCT LINE ARCHITECTURE

8.1 Product Line Architecture Introduction

In traditional software reuse, a library of reuse code components is developed. This approach requires the establishment of a library of reusable components and of an approach for indexing, locating, and distinguishing between similar components [35]. The reusable components in this approach are the building blocks used when constructing the new systems. Components are considered to be largely atomic and ideally unchanged when reused. The problems with this approach include managing the library system which contains all the reusable components, documenting the components, and locating the components. The most important disadvantage of this approach is that the reuse is mainly on code level. The overall reuse of this approach is relatively low.

While software reuse has fallen short of the high expectations placed on it [32, 25], and has failed to deliver on its many promises, a specialized form of software reuse, namely *Product Line Engineering*, has been very successful in fulfilling the goals of software reuse in terms of productivity, quality, and time to market [27, 58-63]. Product Line Engineering deploys the methods of software reuse within the confines of a limited application domain, thereby streamlining and focusing the reuse activity. Product Line Engineering is typically divided into two phases: *Domain Engineering,* where the reuse infrastructure and the reusable assets are developed and cataloged; *Application Engineering,* where actual applications are derived from the domain engineering deliverables in a predefined, streamlined development procedure.

As is usually the case in software engineering, managerial and organizational aspects play a dominant role in Product Line Engineering (PLE) [41], overshadowing technical aspects, and dominating the stakes and the costs associated with projects. To support the management of PLE projects, one must focus his attention on characterizing/quantifying relevant PLE artifacts. The author argues that the *Product Line Architecture* is one of the most important artifacts in a PLE project, since it is the focal point of the domain engineering activity, and has a long term impact through its influence on application engineering. In this chapter, the author explores means to characterize important properties of product line architectures, and he attempts to quantify these properties, to the extent that it is possible and meaningful to do so. The approach to this problem can be characterized by the following premises:

- The author distinguishes between two kinds of attributes of product line architectures: Generic attributes that are meaningful for any software architecture; and attributes that are meaningful only to *product line* architectures.

- He does not interpret *quantification* as *assigning numbers.* There are situations where it is not only impossible to assign numbers; it may in fact be counter-productive. Some artifacts are so complex and/or multi-dimensional that no single numeric function will do them justice.

- He occasionally contents himself with quantifying attributes by mapping them onto a partially ordered set, whenever he feels that assigning a number is either not possible or not meaningful. In such cases, the quantification allows partial comparisons between artifacts.

- He distinguishes between two phases in the derivation of metrics: *Definition* and *Quantification.* In the definition phase, he is primarily concerned with defining a concept that captures the relevant attribute. In the quantification phase, he considers how to quantify the concept in practice, and how to compute its numeric value, if applicable. Separating these two phases helps him ensure that the latter does not affect the former.

8.2 PLA Specific Attributes

Whereas the attributes (Error Propagation, Change Propagation, and Requirements Propagation) the author has discussed in previous chapters apply generally to any software architecture, the attributes he discusses in this chapter apply specifically to product line architectures. Specifically, he considers the following attributes

- Scoping
- Variability
- Commonality
- Applicability

For the purposes of the discussions, the author invokes the previous work on measuring distances between specifications done by other members of the author's research group. In [25], Jilani et al. introduce a wide range of different measures between specifications, and assessed their ability to predict adaptation effort. These measures consist of one specification representing the function of a system and the other specification representing the requirements constraining the system modification The distance between the two specifications is used as an indicator of the modification effort. For the purposes of this discussion, two measures of distance are of interest:

- *Functional Consensus.* The measure of functional consensus between two specifications A and B is denoted by $\Gamma(A, B)$ and represents the functional features that are common between A and B. Details on this measure are given in [25].

- *Refinement Distance.* The measure of refinement distance between two specifications *A* and *B* is denoted by $\delta(A,B)$ and represents the functional features that are in *A* and not in *B*, as well as those that are in B and not in A. Details on this measure are given in [25]. The authors in [25] find that this measure of distance satisfies all the axioms of distance, interpreted over the partially ordered set of specifications (δ is symmetric, takes value zero only when A = B, and satisfies the triangular inequality).

Both of these measures take their values, not in the set of real numbers (the author has argued that this is not strictly necessary), but rather in the set of specifications that are partially ordered by the refinement ordering. The author does keep the possibility of mapping these to numeric values, using function points [28] where applicable.

8.3 Scoping

Scoping is traditionally thought of as a measure of the range of applications that are covered by a product line. The author takes a slightly different interpretation in this chapter by equating it with the maximal refinement distance between any two applications of a product line. Formally,

$$\text{Scoping} = \text{Max}_{A,B}\, \delta(A, B).$$

If one pictures the application domain as a set represented in a topological space, then scoping can be interpreted as the diameter of the set, as shown in Figure 8.1. One way to approximate this term in practice is to choose two sample applications that are as distinct as possible (e.g. one has all the optional features that are possible, the other has none), and compute their refinement distance, which is a specification. If one wants to quantify scoping by a numeric value, he can then compute the function point values (or other values) of the specification that he obtains.

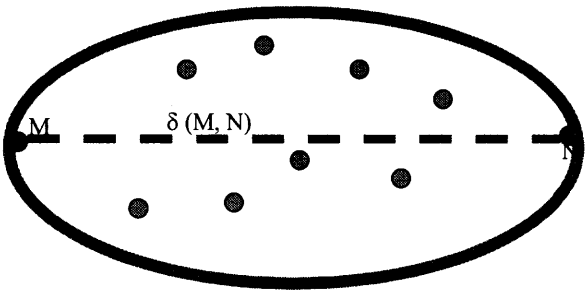


Figure 8.1 Scoping is the maximum distance between any two systems from the PLA.

Application: The author uses a library system architecture [26] as his case study. The architecture of the system is shown in Figure 8.2. In that figure, solid lines represent core assets (included in all the systems derived from the architecture); dashed lines represent optional components (some systems have them and some systems do not); a layered view indicates variance (a component has multiple versions).

Table 8.1 shows the component sizes (they are pseudo numbers) and two systems derived from the architecture. The check mark in a cell defined by a system and a component means that the system includes the component. The two sample systems from the PLA are two extreme systems. One (P2) has all the optional components and if a component has multiple versions the system includes the version with the largest size. Another one (P1) goes opposite direction, i.e. it does not have any optional components and if a component has multiple versions the system includes the version with the smallest size.

The distance between these two systems should represent the scoping of the PLA. Using the data in Table 8.1 the author can calculate the scoping. Scoping = Dist (P1, P2) = (P1 $\cup$ P2) – (P1 $\cap$ P2). The author considers (P1 $\cup$ P2) as one system and (P1 $\cap$ P2) as

another system. He gets the distance as 300 function points. If one decides the PLA should not include *Medium Service* component. Then this time the scoping is 200 function points. This shows with fewer optional components the scoping is smaller. This shows how by reducing the scope of the domain (understood in the intuitive sense), one can reduce the measure of *scoping* (as defined above) from 300 to 200 function points.

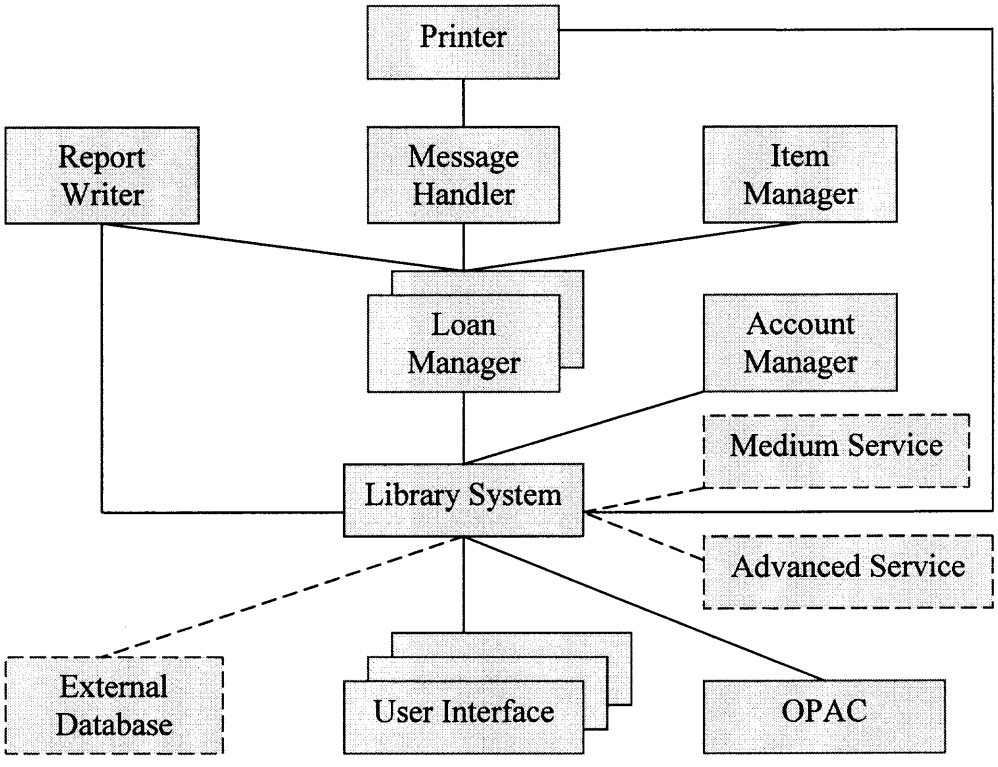


Figure 8.2 Architecture of the library system PLA.

Table 8.1 Components and Their Sizes in Function Points

	CP	FP	P1	P2
Account Manager	C	40	√	√
External Database (ED)	O	10		√
Item Manager	C	10	√	√
Library System	C	90	√	√
Loan Manager (V1) (LM1)	V	60	√	
Loan Manager (V2) (LM2)	V	70		√
Message Handler	C	10	√	√
OPAC	C	10	√	√
Printer	C	10	√	√
Report Writer	C	20	√	√
User Interface (V1) (UI1)	V	230		√
User Interface (V2) (UI2)	V	190		
User Interface (V3) (UI3)	V	90	√	
Medium Services (MS)	O	100		√
Advanced Services (AS)	O	40		√

Table 8.1 Legend: CT — Component Type; C — Core component; V — Variance component; O — Optional component. P1, P2 — two products (or systems) derived from the PLA.. Check mark means the product has the component.

8.4 Variability

The author interprets variability to represent the degree of variation between the functional properties of applications within the domain of interest. This is important because it has an impact on the precision that is required to identify a particular application in the domain, hence could potentially have an impact on many aspects of

domain engineering and application engineering, such as:

- The effort required to specify an application (with sufficient precision to distinguish it from other applications in the domain)--an application engineering impact.

- The effort required to build an application from specifications, using domain engineering assets--an application engineering impact.

- The effort required to support/provide for all the dimensions of variability that the author wishes to support--a domain engineering impact.

To use an analogy with geography, if the author represents the domain by a territory, scoping measures the size of the territory while variability measures its population density; together, they can be used to assess the size of the population (or the domain). The author has not found a totally satisfactory measure for this feature; one possible candidate is the minimal distance between any two distinct applications, as shown in Figure 8.3.

$$\text{Variability} = \text{Min}_{A,\, B:\, A \neq B}\, \delta(A, B)$$

The distance $\delta(A, B)$ is itself a specification, to which the author can apply function point procedures to derive a numeric value.

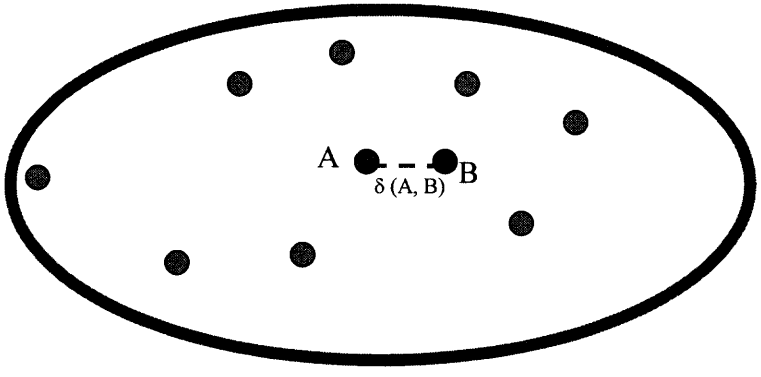


Figure 8.3 Variability is the minimum distance between any two systems from the PLA.

Application: The author uses the same Library System PLA to do this case study. In order to estimate variability, he needs to find the minimal distance between any two products of this domain. As a substitute for computing the distance between any two products to find the smallest, he merely computes the distance between two neighboring products, and uses that as an upper bound of the distance. To this effect, the author lets product P3 have every feature of P1, and have an extra feature, *External Database* (ED). Then, $\delta(P1, P3) = \text{Size (ED)} = 10$ Function Points. Then, he can say the variability of this PLA is less than 10 Function Points.

$$\text{Variability} \leq 10$$

8.5 Commonality

Commonality is usually understood to be a measure of how much [what?] applications in the domain have in common. In [25], the authors have defined the *functional consensus* of two specifications A and B as the composition of A and B by a lattice operator, which is commutative and associative. Hence, this operator can be applied on more than two arguments, even on an infinity of arguments (due to properties of the lattice, whose discussion is beyond the scope of this paper). Using the measure of *functional consensus*, the author defines commonality as the functional consensus of all the applications in the domain.

$$\text{Commonality} = \Gamma_{A \in \text{DOMAIN}}\ A$$

Intuitively, this definition provides that the commonality of a domain is the specification that captures all the functional details that all applications of the domain have in common. Even though, formally, this measure involves all the applications in the

domain, in practice it does not require that one enumerates all the applications to compute their product. The information that all applications of the domain share is usually readily available as a prominent feature of the domain description. If one wants a numeric value for commonality, he can apply the function point process to the specification derived from the definition above.

Application: The author still uses the Library System as the sample PLA. If one requires all the applications derived from the PLA has the following features:

- Account Management which has the following functions: create an account, identify an account, remove an account from the system, and print the information of an account.
- Item Manager which has the following functions: get an item.
- Library System which has the following functions: identify an account, create a new account, remove an account, identify an item, loan an item, print out information of an account, return an item, re-loan an item, and search the system.
- Loan Manager which has the following functions: loan an item, print the information of an account, provide loan information, re-loan an item, return an item, set up an account, and close an account.
- Message Handler which shall display messages to users.
- OPAC which shall provide online access catalog.
- Printer which has print function.
- Report writer which shall have print account information and print function.
- User interface which shall provide users at lease with the access to the following functions: identify an account, create a new account, remove an account, identify an item, loan an item, print out information of an account, return an item, re-loan an item, and search the system.

The above description represents the common features required for all the

applications derived from the PLA. So the description is the commonality of the PLA. If one wants to represent it in a numeric value the commonality should be the size (could be in function points or in other measurements) of all the common features. In this case, one just adds all the function points of the core components resulting 340 function points.

8.6 Applicability

The author understands applicability to refer to the ease with which one can produce a domain application from domain assets. The domain engineer has to find a tradeoff between two conflicting requirements

- *Usefulness.* To enhance the usefulness of the application domain, the domain engineer must widen the scope; a wider domain caters to more needs and encompasses more applications.

- *Usability.* To enhance the usability of domain assets, the domain engineer must minimize the effort required to derive an application within the domain.

These requirements are clearly in mutual conflict: To enhance usefulness one needs to make domain assets more general. In other words, what one needs to make them cater to a wider range of possible functions. Whereas to enhance usability, one needs to make domain assets more specific or specialize them to a narrower range of functions. Genericity helps maintain usefulness (generality) without adversely affecting usability.

The author wants *applicability* to reflect/measure where the domain engineer has struck the tradeoff between usefulness and usability. The author uses two possible definitions of applicability:

- The *refinement distance* between domain requirements and application requirements; a numeric value can be obtained by computing the corresponding function point value.

- More simply, the average application engineering cost, computed over all possible domain applications.

Empirical studies conducted by Jilani et al. [25] show that these two measures are correlated. There approach gives a sense that refinement distance can be used to predict application engineering costs. Hence for the purposes of this definition, the author defers to the first interpretation of applicability:

$$\text{Applicability} = \text{Min } A \in \text{DOMAIN } \delta(R,A)$$

Where R is the domain requirements specification and A is the requirements specification of a variable element of the domain. The interpretation is shown in Figure 8.4.

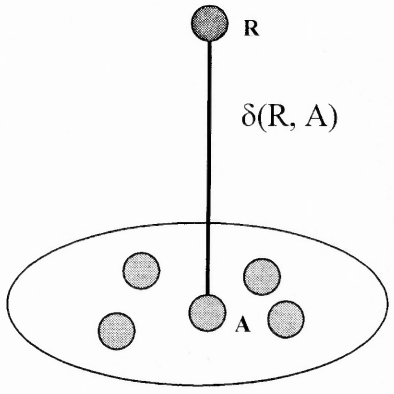


Figure 8.4 Applicability is the minimum distance from the requirements specification of a PLA to the requirements of a system R.

8.7 Conclusion

In this chapter, the author has attempted to propose some attributes that pertain to product

lines, and have discussed means to characterize them and quantify them. He has shown how some of the proposed characterizations/quantifications reflect the desired properties. He feels that the design of a product line involves a lot of tradeoffs between the imperatives of usefulness and usability, and that the set of measures (scoping, variability, commonality, applicability) can potentially provide a sound objective basis for quantifying these tradeoffs, and possibly optimizing them. To define these measures, the author has used measures of distance between specifications, and have taken some liberties with converting these into numeric functions; this matter needs to be revised carefully, for possible revision. This work is clearly in its infancy, and at a very tentative stage. However, it does show some potential for further exploration.

CHAPTER 9

SUMMARY

In this dissertation, the author introduces what is software architecture; what are the benefits of using software architecture. The necessity of measuring software architecture is discussed. Three metrics for general architecture are proposed, mathematically modeled, and validated by empirical studies; four metrics for product line architecture are proposed, discussed.

9.1 Error Propagation

In Chapter 3 and 4, error propagation metrics is defined. The error propagation can be used in the following ways.

- Good indicator for the reliability of a system.
- Help architects to design better software architecture; minimize the EP.
- If a component has high EP value to all other components, take special action to minimize errors at run-time.
- If a column has high values, take special steps to immunize this component against errors, e.g. by providing it with fault tolerance capabilities (error detection, damage assessment, error recovery)

A mathematical model is proposed. A case study is carried out on a lift-critical sample system. The author uses a fault injection experiment to validate the analytical result. The correlation coefficient is 0.628, which indicates analytical result is related to the empirical study result.

9.2 Change Propagation

In Chapter 5 and 6, change propagation metrics are defined. Change propagation can be used in many ways to help architects to design better system. The following is several ways CP can be used.

- Good indicator for the Modifiability of a system.
- Lower CP indicates lower changing cost.
- A good software architecture should have minimum CP

If a component has a high CP value, take special caution to design it. Intense validation and verification should be carried out when changing such a component.

A case study on a spreadsheet application is performed. The analytical results are validated by the results from an empirical study. The correlation coefficient between analytical results and empirical results is 0.85. It is concluded that the results are highly related.

9.3 Requirements Propagation

In Chapter 7, the author proposes a metrics called requirement propagation. This metric has at least the following usages.

- This matrix can be used to find out components that are likely to undergo frequent changes in the maintenance phase.
- RP matrix can also be used to compare candidate system architectures when change impact is an important consideration.
- If the row corresponding to a component A has high values, the author infers that changes to this component must be avoided because they propagate widely throughout the system.
- If the column corresponding to a component A has high values, the author infers that this component is likely to undergo frequent changes in the maintenance phase.

Three mathematic formulas are proposed. A same system as used for CP is used to perform a case study. The correlation coefficients between analytical results and the empirical results are high.

9.4 Metrics for Product Line Architecture

Product line architecture is a common structure among a group of related products. In Chapter 8, the author quantifies some properties of product line architecture. Four metrics are proposed, scoping, variability, commonality, and applicability.

- Scoping is used to measure the scope of the whole architecture.
- Variability is used to explore the variation of a given PLA.
- Commonality is the measurement of how much in common of the products derived from the architecture.
- Applicability is the cost to produce a system from a PLA.

A library system is used to perform a case study. Although most of the measurements are done using requirement specifications, the author tries to convert some of the specifications into numbers.

9.5 Conclusion

In this dissertation, several metrics for software architecture are defined. Mathematic models are proposed. The results from analytical results are validated by empirical studies. Also an automatic CASE tool is developed for calculating change propagation. The author argues that EP metrics is related to reliability of a software system. CP and RP metrics' are reflections of modifiability and maintainability of software systems.

APPENDIX A

ANALYTICAL FORMULA OF ERROR PROPAGATION

Analytically, the error propagation probability (as defined in Chapter 4), can be shown to have the following expression in terms of the probabilities of the individual A-to-B messages and states of B:

$$EP(A \rightarrow B) = \frac{1 - \sum_{x \in S_B} P_B(x) \sum_{y \in S_B} P_{A \rightarrow B}[F_x^{-1}(y)]^2}{1 - \sum_{v \in V_{A \rightarrow B}} P_{A \rightarrow B}[v]^2},$$

where $F_x^{-1}(y) = \{ v \in V_{A \rightarrow B} \mid F_x(v) = y \}$, and the author assumes a probability distribution P_B on the set of states S_B and a probability distribution $P_{A \rightarrow B}$ on the data vocabulary $V_{A \rightarrow B}$. As one can see from the above formula, the value of $EP(A \rightarrow B)$ depends on two expressions. The expression

$$\xi(A,B) := \sum_{v \in V_{A \rightarrow B}} P_{A \rightarrow B}[v]^2$$

is the so called *collision probability* of the ensemble $(V_{A \rightarrow B}, P_{A \rightarrow B})$. It is easy to see that $1/|V_{A \rightarrow B}| \leq \xi(A,B) \leq 1$.

Interestingly, the expression

$$-\log\left(\sum_{v \in V_{A \rightarrow B}} P_{A \rightarrow B}[v]^2 \right)$$

behaves very much like the Shannon entropy of the probabilistic ensemble $(V_{A \rightarrow B}, P_{A \rightarrow B})$: the more homogeneous is the distribution– the larger is the value of (4), which reaches its maximum of $\log|V_{A \rightarrow B}|$ at the ensemble of highest uncertainty (all elements have equal

probabilities), and its minimum of 0 at the ensemble of lowest uncertainty (all but one elements have probability zero). In fact, expression (4) turns out to be the so-called *2nd order Renyi entropy* $H_2^R(V_{A\to B}, P_{A\to B})$ (see e.g. [9]) of the *A-to-B data flow ensemble* ($V_{A\to B}$, $P_{A\to B}$). Thus,

$$\xi(A,B) = \exp(-H_2^R(V_{A\to B}, P_{A\to B})).$$

Let us now look at the expression that appears in the numerator of the error propagation formula,

$$\eta(A,B) := \sum_{x\in S_B} P_B[x] \sum_{y\in S_B} P_{A\to B}[F_x^{-1}(y)]^2$$

Assume, for the sake of simplicity, that all states $x \in S_B$ of component B are equi-probable (with probabilities $P_B(x) = \frac{1}{|S_B|}$), and all messages $v \in V_{A\to B}$ sent by A to B are also equi-probable probable (with probabilities $P_{A\to B}(v) = \frac{1}{|V_{A\to B}|}$). In this case the expression for $\eta(A,B)$ is reduced to:

$$\frac{1}{|S_B \| V_{A\to B}|^2} \sum_{x\in S_B} \sum_{y\in S_B} |F_x^{-1}(y)|^2,$$

where $|F_x^{-1}(y)|$ is calculated simply by counting the number of messages B receives from A that trigger the state transition from x to y.

Also, when all the messages are equally probable, the expression $\xi(A,B)$ reaches its minimum value $\frac{1}{|V_{A\to B}|}$ (respectively, the Renyi entropy reaches its maximum value

of log| $V_{A\to B}$ |). Thus, under the equi-probability assumption (stated above), the formula for the error propagation from A to B gets simplified as follows:

$$EP(A\to B) = \frac{1-\frac{1}{|S_B||V_{A\to B}|^2}\sum_{x\in S_B}\sum_{y\in S_B}|F_x^{-1}(y)|^2}{1-\frac{1}{|V_{A\to B}|}}.$$

Since one cannot always extract from the available representation the detailed information on the transition table F for the architectural components, it would helpful to

$$\sum_{x\in S_B}\sum_{y\in S_B}|F_x^{-1}(y)|^2$$

be able to estimate the term in the absence of any knowledge of function F. For any fixed $x\in S_B$, $\{F_x^{-1}(y) \mid y\in S_B\}$ is a partition of the set of messages $V_{A\to B}$ into S_B subsets, some of which may be empty. Thus,

$$\sum_{y\in S_B}|F_x^{-1}(y)| = |V_{A\to B}|,$$

And it is easy to see that, for such a partition, the sum

$$\sum_{y\in S_B}|F_x^{-1}(y)|^2$$

reaches its maximum value of $|V_{A\to B}|^2$ for the trivial partition consisting of a single set, and it reaches its minimum value of $|V_{A\to B}|^2/|S_B|$ for a 'uniform' partition that consists of $|S_B|$ sets of equal cardinality. It should be noted that, clearly, the minimum is reachable only when $|S_B|$ divides $|V_{A\to B}|$; otherwise it is an unattainable (but tightest possible) lower bound. Thus

$$|V_{A\to B}|^2 \le \sum_{x\in S_B}\sum_{y\in S_B}|F_x^{-1}(y)|^2 \le |S_B||V_{A\to B}|^2 .$$

This yields the following bounds for the error propagation value

$$0 \le EP(A,B) \le \frac{1-\frac{1}{|S_B|}}{1-\frac{1}{|V_{A\to B}|}}.$$

Thus, for instance, if the receiving component *B* has 2 states, and is capable of receiving 5 different messages from component A, the A-to-B error propagation cannot exceed (1-0.5)/(1-0.2) = 0.625.

APPENDIX B

SHARPTOOL APPLICATION SUMMARY MATRIX

Appendix B lists all the files in SharpTool application.

File Name	Alias	No. of Lines	No. of Methods	Avg. Complexity
AddressField	C1	117	5	2
Cell	C2	317	19	2
CellPoint	C3	130	10	1
CellRange	C4	147	12	1
Config	C5	230	10	2
ConnectDialog	C6	244	6	3
Database	C7	230	4	5
Debug	C8	23	3	1
EditOp	C9	434	12	3
FileOp	C10	876	29	3
FindDialog	C11	84	6	1
Formula	C12	1166	32	4
Function	C13	943	76	1
Help	C14	243	8	2
HistoDialog	C15	472	35	1
Histogram	C16	189	9	1
History	C17	291	17	1
NewFileDialog	C18	141	8	1
Node	C19	300	34	1
NumberField	C20	233	10	2
ParserException	C21	61	5	1
PasswordDialog	C22	129	6	2
SharpCellEditor	C23	73	4	1
SharpCellRenderer	C24	144	4	3
SharpClipboard	C25	122	7	1
SharpDialog	C26	356	27	1
SharpOptionPane	C27	236	8	2
SharpTableModel	C28	1676	56	3
SharpTools	C29	1189	81	1
SortDialog	C30	199	11	1
TableOp	C31	311	9	3
TabPanel	C32	502	22	2

APPENDIX C

SHARPTOOL FEATURES AND THEIR CORRESPONDING CLASSES

Appendix C lists all the features of SharpTools system.

Features		Related Components
1. File Operation	1.1 New File	C10: FileOp & C5: Config
	1.2 Open File	C10: FileOp
	1.3 Open Database	C10: FileOp, C7: Database, C6: ConnetDialog
	1.4 Save	C10: FileOp
	1.5 Save as	C10: FileOp
	1.6 Print	C10: FileOp
	1.7 Set Password	C10: FileOp & C22: PasswordDialog
	1.8 Exit	C10: FileOp
2. Record the history of recently used files		C10: FileOp & C5: Config
3. Table		C28: SharpTableModel, C2: Cell, C3: CellPoint
4. Edit Operations	4.1 Undo	C17: History
	4.2 Redo	C17: History
	4.3 Cut	C9: EditOp
	4.4 Copy	C9: EditOp
	4.5 Paste	C9: EditOp
5. Search Ability	5.1 Find	C9: EditOp
	5.2 Find Next	C9: EditOp
6. Table Operations	6.1 Insert Row	C32: TableOp, C4: CellRange
	6.2 Insert Column	C32: TableOp, C4: CellRange
	6.3 Delete Row	C32: TableOp, C4: CellRange
	6.4 Delete Column	C32: TableOp, C4: CellRange
	6.5 Sort Row	C32: TableOp, C30 SortDialog
	6.6 Sort Column	C32: TableOp, C30 SortDialog
	6.7 Set Column Width	C32: TableOp, C5: Config C27: SharpOptionPane
7. Chart Operation	7.1 Display Histogram	C16: Histogram, C31: TabPanel
	7.2 Hide Histogram	C16: Histogram, C31: TabPanel
8. Functions		Function
9. Help□	9.1 Help File	C14: Help
	9.2 Function Info	C14: Help, C13: Function, C12: Formula
	9.3 About	C14: Help
10. Graphical User Interface		C29: SharpTools

REFERENCES

1. P. Clements, "Evaluating Software Architectures: methods and Case Studies", Addison-Wesley, 2002.

2. J.J. Marchiniak, "Encyclopedia of Software Engineering", John-Wiley & Sons (Vol. I), 1994.

3. SEI software architecture website: http://www.sei.cmu.edu/architecture/definitions.html. (Accessed December 5, 2005)

4. Len Bass, Paul Clements, and Rick Kazman, "Software Architecture in Practice", Addison Wesley, 1998.

5. M. Shereshevsky, H. Ammari, N. Gradetsky, A. Mili, and H. Ammar, "Information theoretic metrics for software architectures", In Proceedings, COMPSAC 2001: Computer Software and Applications, Chicago, IL, 2001.

6. T.J. McCabe, "A complexity measure", IEEE Transactions on Software Engineering, Vol. 2(4), 1976, pp. 308-320.

7. J.M. Bieman and B.K. Kang, "Measuring design level cohesion", IEEE Transactions on Software Engineering, Vol. 24(2), February 1998, pp. 111-124.

8. L.C. Briand, S. Morasca, and V.R. Basili, "Property based software engineering measurement", IEEE Transactions on Software Engineering, Vol. 22(1), January 1996.

9. A. Renyi. "On Measures of Entropy and Information", in N. Neyman (editor), Proceedings of the Fourth Symposium on Mathematics, Statistics, and Probability. San Francisco, CA 1961, pp. 547-561.

10. C. Shannon. "A Mathematical Theory of Communication". Bell Syst. Tech. Journal, Vol. 27, 1948, pp. 379-423 and 623-656.

11. D. M. Nassar, W. A. Rabie, M. Shereshevsky, N. Gradetsky, H.H. Ammar, Bo Yu, S. Bogazzi, and A. Mili, Estimating Error Propagation Probabilities in Software Architectures", Technical Report, College of Computer Science, New Jersey Institute of Technology 2002. http://www.ccs.njit.edu/swarch/ep.pdf. (Accessed December 5, 2005)

12. Karol Zyczkowski, "Renyi Extrapolation of Shannon Entropy", 2003 http://arxiv.org/abs/quant-ph/0305062. (Accessed December 5, 2005)

13. Rational Rose Realtime, IBM Rational Software, http://www.rational.com. (Accessed December 5, 2005)

14. M. Hiller, A. Jhumka, and N. Suri, "An Approach for Analyzing the Propagation of Data Errors in Software," Dependable Systems and Networks, 2001, pp. 161-170.

15. SEI Product Line Web site: http://www.sei.cmu.edu/productlines/index.html. (Accessed December 5, 2005)

16. J.M. Bieman and L.M. Ott. "Measuring Functional Cohesion". IEEE Transactions on Software Engineering, Vol. 20(8), August 1994, pp. 644-657.

17. M. Shaw. "Architectural Issues In Software Reuse: It's Not Just The Functionality, It's The Packaging", Proceedings, Symposium on Software Reusability, Seattle, WA, April 1995. Association for Computing Machinery.

18. M. Shereshevsky, H. Ammari, N. Gradetsky, A. Mili, and H. Ammar. "Information Theoretic Metrics For Software Architectures". In Proceedings, COMPSAC 2001: Computer Software and Applications, Chicago, IL, 2001.

19. V.R. Basili and H.D. Rombach, "The Tame Project: Towards Improvement Oriented Software Environments", IEEE Transactions on Software Engineering, Vol. 14(6), June 1988, pp. 758-773.

20. M. H. Halstead, "Elements of Software Science", North Holland, Amsterdam, 1977.

21. C. Kobryn, "UML 2001: A Standardization Odyssey", Communications of the ACM Vol. 42(10), pp. 29-37, New York: ACM Press.

22. F. Bachman, Bass, S. Buhman, S. Comella-Dorda, F. Long, R. C. Seacord, and K. C. Wallnau, "Volume II: Technical Concepts of Component-Based Software Engineering", Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, 2000.

23. C. Szyperski, Component Software - Beyond Object-Oriented Programming, ISBN 0-201-17888-5, Addison-Wesley, 1998.

24. P. Bengtsson, et al., "Architecture-level modifiability analysis (ALMA)", The J. of System. And Software, Vol. 69, (2004 pp. 129-147.

25. L. Jilani, J. Desharnais, A. Mili, "Defining And Applying Measures of Distance Between Specifications", IEEE Transactions on Software Engineering, Vol. 27 (8), 2001, pp. 673-703.

26. J. Bayer, D. Muthig, , B. Göpfert, "The Library System Product Line- A KobrA Case Study", http://www.iese.fraunhofer.de/pdf_files/iese-024_01.pdf. (Accessed December 5, 2005)

27. P. Clements, and L. Northrop, "Software Product Lines: Practices and Patterns", Addison-Wesley Pub Co, 2001.

28. D. Garmus, and D. Herron, "Function Point Analysis: Measurement Practices for Successful Software Projects", Addison-Wesley, 2001.

29. O. Gotel, and A. Finkelstein, "An Analysis of the Requirements Traceability Problem", Proc. First Int'l Conf. Requirements Eng., 1994, pp. 94-101.

30. B. Ramesh, and M. Jarke, "Toward Reference Models for Requirements Traceability", IEEE Transactions on Software Engineering, Vol. 27(1), Jan 2001, pp. 58-93.

31. L. Bass, P. Clements, and R. Kazman, "Software Architecture in Practice", Addison Wesley, 1998.

32. C. Atkinson, J. Bayer, C. Bunse, et al., "Component-based Product Line Engineering with UML", Addison Wesley, 2002.

33. D. Yakimovich, J.M. Bieman, V.R. Basili, "Software Architecture Classification For Estimating The Cost Of COTS Integration", Proceedings of the 1999.

34. B. Boehm, E. Horowitz, et al., "Software Cost Estimation with COCOMO II", Prentice Hall PTR, 2000.

35. R. Prieto-Diaz, P. Freeman, "Classifying Software for Reusability", IEEE Software, Vol. 4(1), 1987, pp. 6-16.

36. H. Ammar, S. M. Yacoub, A. Ibrahim, "A Fault Model for Fault Injection Analysis of Dynamic UML Specifications," International Symposium on Software Reliability Engineering, IEEE Computer Society, November 2001.

37. D.S. Moore and G.P.McCabe, "Introduction to The Practice of Statistics", W.H. Freeman and Company, 4th edition, 2003.

38. P. Bengtsson, et al., "Architecture-Level Modifiability Analysis (ALMA)", The Journal of Systems and Software, Vol. 69, 2004, pp. 129-147.

39. P.J. Clarkson, C.Simons, and C.M. Eckert, "Change Propagation in the Design of Complex Products", in Engineering Design Conference (EDC2000), Brunel University, Uxbridge, 2000, pp. 563-570.

40. P.J. Clarkson, , C.Simons, and C.M. Eckert, "Predicting Change Propagation in Complex Design", Proceedings 13th International Conference on Design Theory and Methodology (DETC'01), ASME Design Engineering Technical Conferences and Computers and Information in Engineering Conference, Pittsburgh, Pennsylvania, USA, 2001.

41. H. Mili, A. Mili, S. Yacoub E. Addy, "Reuse-Based Software Engineering: Techniques, Organization, and Controls", Wiley-Interscience; 1st edition (December 15, 2001).

42. K. Moller, "Software Metrics: A Practitioner's Guide to Improved Product Development", Chapman & Hall; 1st edition, 1993.

43. V. Cote, P. Bourque, S. Oligny, and N. Rivard, "Software Metrics: an Overview of Recent Results", The Journal of Systems and Software, Vol. 8, 1988, pp. 121-131.

44. S. Lock, G. Kotonya, "An Integrated Framework for Requirement Change Impact Analysis", proceedings of the 4th Australian Conference on Requirements Engineering, September 1999.

45. M. Halstead, "Elements of Software Science". Elsevier, New York, 1977.

46. B. Boehm, "Software Engineering Economics", Prentice Hall PTR (1981).

47. L. Putnam, "Tutorial on Software Cost Estimation and Life Cycle Control: Getting the Software Numbers", Computer Society Press, Los Alamitos, CA, 1980.

48. H. Gomaa, "Designing Software Product Lines with UML : From Use Cases to Pattern-Based Software Architectures", Addison-Wesley Professional (July 7, 2004).

49. H. Gomaa, "Reusable Software Requirements and Architectures for Families of Systems", Journal of Systems and Software, Vol. 28, 1995, pp. 189-202.

50. G. Booch, "Object-Oriented Analysis and Design with Applications", 2nd Ed. Reading, MA: Addison-Wesley, 1994.

51. I. Jacobson, "Object-Oriented Software Engineering: A Use Case Driven Approach, Reading", MA: Addison-Wesley, 1992.

52. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson, "Object – Oriented Modeling and Design", Upper Saddle River, NJ: Prentice Hall, 1991.

53. OMG website: http://www.omg.com. (Accessed December 5, 2005)

54. R. Soley and the OMG Staff Strategy Group, "Model-Driven Architecture", ftp://ftp.omg.org/pub/docs/omg/00-11-05.pdf. (Accessed December 5, 2005)

55. IBM Rational: http://www-306.ibm.com/software/rational/.

56. W. Boggs, M. Boggs, "Mastering UML with Rational Rose 2002", Sybex Books, 2002.

57. OMG Meta-Object Facility: http://www.omg.org/technology/documents/formal/mof.htm. (Accessed December 5, 2005)

58. J.D.McGregor, , L.M. Northrop, S. Jarrad, K. Pohl, "Initiating Software Product Lines", IEEE Software, Vol. 19(4), Jul/Aug, 2002, pp. 24-27.

59. F. Van der Linden, "Software Product Families in Europe: the Esaps & Cafe projects", IEEE Software, Vol: 19(4), Jul/Aug, 2002, pp. 41-49.

60. K. Schmid, M. Verlage, "The Economic Impact of Product Line Adoption and Evolution", IEEE Software, Vol 19(4), Jul/Aug, 2002, pp. 50-57.

61. J.Bosch, "Product-Line Architectures in Industry: a Case Study", Proceedings of the 1999 International Conference on Software Engineering, 16-22 May 1999, pp. 544-554.

62. D. Batory, "Product-Line Architectures, Aspects, and Reuse", Proceedings of the 2000 International Conference on Software Engineering, 4-11 June 2000, pp. 832-832.

63. A. Garg, M. Critchlow, P. Chen; C. Van der Westhuizen A. van der Hoek, "An Environment for Managing Evolving Product Line Architectures", 2003. ICSM 2003. Proceedings. International Conference on Software Maintenance, 22-26 Sept. 2003, pp. 358-367.

64. J.S. O'Neal, D.L. Carver, "Analyzing the Impact of Changing Requirements", 2001. Proceedings. IEEE International Conference on Software Maintenance, 7-9 Nov. 2001, pp. 190-195.

65. A.J.C. Blyth, J. Chudge, J.E. Dobson, M.R. Strens, "A Framework for Modeling Evolving Requirements", COMPSAC 93. Proceedings., Seventeenth Annual International Computer Software and Applications Conference, 1-5 Nov. 1993, pp. 83-89.

66. J. Han, "Supporting Impact Analysis and Change Propagation in Software Engineering Environments", 8th International Workshop on Software Technology and Engineering Practice [incorporating Computer Aided Software Engineering], July 14-18, 1997, pp. 172 –182.

67. S. A. Bohner, "Software Change Impacts-an Evolving Perspective", Proc. 2002 IEEE Int. Conference on Software Maintenance, 3-6 Oct. 2002, pp. 263-272.

68. Java Understand: http://www.scitools.com/uj.html. (Accessed December 5, 2005)

69. V. Rajlich, "A Model for Change Propagation Based on Graph Rewriting", Proc. 1997 IEEE Int. Conference on Software Maintenance, 1997, pp. 84-91.

70. L. Deruelle, M. Bouneffa, N. Melab, H. Basson, "A Change Propagation Model and Platform for Multi-Database Applications", Proc. 1997 IEEE Int. Conference on Software Maintenance, 2001, pp. 42-51.

71. N. Fenton, and S. Pfleeeger, "Software Metrics: A Rigorous & Practical Approach", 2nd Ed, PWS Publishing Company, Boston, MA, 1997.

72. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, "Pattern-Oriented Software Architecture", Volume 1: A System of Patterns (Hardcover), 1st Ed, John Wiley & Sons, August 8, 1996.

73. A.S. Tanenbaum, "Modern Operation System", Prentice Hall, 1992.

74. A. Aho, R. Sethi, J. Ullman, "Compilers- Principles, Techniques, and Tools", Addison Weseley, 1986.

75. R. Engelmore, T. Morgan, "Blackboard Systems", Addison Weseley, 1988.

76. Object Management Group, "The Common Object Request Broker: Architecture and Specification", OMG Document Number 91.12.1, Revision 1.1 1992.

77. A. Goldberg, D. Bobson, "Smalltalk-80: the Language and its Implementation", Addison Wesley, 1983.

78. C. Traving, H, Stadtherr, "Building a Traffic Management System with C++", Proceedings of the C++ User Group Technical Conference, Munich, 1993.

79. S. Kee, "Object-Oriented Programming in Common Lisp – A Programmer's Guide to CLOS", Addison Wesley, 1989.

80. Chorus systems, "Chorus Kernel v3.2, Implementation Guide", CS/TR-90-5.

81. M. Kogure, Y. Akao, "Quality Function Deployment and CWQC in Japan", Quality Progress, October 1983, pp. 25-29.

82. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative Evaluation of Software Quality", Proceedings of the Second International Conference on Software Engineering, 1976, pp. 592-605.

83. J. A. McCall, P.K. Richards, G.F. Walters, "Factors in Software Quality", Rome Air Development Center, RADC TR-77-369, 1977.

84. Forrest Research, http://www.forrester.com/Research/Document/Excerpt/0,7211,37381,00.html. (Accessed December 5, 2005)

85. Scientific Toolworks, Inc., Understand for Java, http://www.scitools.com/. (Accessed December 5, 2005)

86. Carnegie Mellon University, "The Acme Architectural Description Language", On-line at: http://www.cs.cmu.edu/~acme/. (Accessed December 5, 2005)

87. D. Garlan, R. T. Monroe, and D. Wile, "Acme: An Architecture Description Interchange Language", *Proceedings of CASCON '97*, Toronto Canada, November 11, 1997, pp. 169-183.

88. S. Roh, K, Kim, and T. Jeon, "Architecture modeling language based on UML2.0", Rapid System Prototyping, 2005. (RSP 2005). The 16th IEEE International Workshop on 8-10 June 2005, pp. 202-208.

89. N. Wirth, "Program Development by Stepwise Refinement", Communications of the ACM, Vol. 14(4), 1971, pp. 221-227.

90. D. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules", Communications of the ACM, Vol. 15(12), 1972, pp. 1052-1058.

91. E. Yourdon and L. Constantine, "Structured Design: Fundamentals of Discipline of Computer Program and Systems Design", Prentice Hall, 1985.

92. D. Perry and A. Wolf, "Foundations for the Study of Software Architecture", Software Engineering Notes, Vol. 17(4), October, 1992, pp. 40-52.

93. T. M. Khoshgoftaar, K. Ganesan, E. B. Allen, F. D. Ross, R. Munikoti, N. Goel, and A. Nandi, "Predicting fault prone modules with case-based reasoning", 8th International Symposium on Software Reliability Engineering, Albuquerque, NM, November 1997.

94. W. Harrison, "An entropy-based measure of software complexity", IEEE Transactions on Software Engineering, Vol. 18(11), Nov. 1992, pp. 1025-1029.

95. N. Chapin, "Entropy metric for systems with COTS software", Proceedings, Metrics 2002, Ottawa, Ont, Canada, 2002.

96. N. Fenton, "Software measurement: A necessary scientific basis", IEEE Transactions on Software Engineering, Vol. 20(3), March 1994, pp. 199-206.

97. J. Voas, "Error propagation analysis for COTS system," Journal of Computing & Control Engineering, Vol. 8(6), Dec. 1997, pp. 269 –272.

98. C. Michael, and R. C. Jones, "On the Uniformity of Error Propagation in Software," Proc. of the 12th Annual Conference on Computer Assurance (COMPASS'97), 1997, pp. 68-76.

99. S. J. Geoghegan, D. Aversky, "Method for Designing and Placing Check Sets based on Control Flow Analysis of Programs", Proc. Int. Symposium on Software Reliability Engineering (ISSRE' 96), 1996, pp. 256-265.

100. A. Jhumka, M. Hiller, and N. Suri, "Assessing Inter-Modular Error Propagation in Distributed Software", in Proc of the 20th Symposium on Reliable Distributed System, 2001, pp. 152-161.

101. K. G. Shin, T. Lin., "Modeling and Measurement of Error Propagation in a Multi-module Computing System", IEEE Transactions on Computers, Vol. 37(9), 1988, pp. 1053-1066.

102. Iyer R. K., Tang D., "Experimental Analysis of Computer System Dependability", Chapter 5 in Fault-Tolerant Computer System Design (ed. D.K, Pradhan), Prentice Hall, 1996.

103. Powell D., et al., "Estimators for Fault Tolerance Coverage Evaluation", IEEE Transactions on Computers, Vol. 44(2), 1995, pp. 261-274.

104. V. Rajlich, "Modeling Software Evolution by Evolving Interoperation Graphs," Annals of Software Engineering Vol. 9, 2000, pp. 235-248.

105. S. R. Schach, and A. Tomer, "A Maintenance-oriented Approach to Software Construction," Journal of Software Maintenance-Research and Practice, Vol. 12(1), 2000, pp. 25-45.

106. T. Cohen, S. Navthe, and R. Fulton, "C-FAR, Change Favorable Representation." Computer-Aided Design, Vol. 32, 2000, pp. 321-38.

107. D. V. Steward, "The Design Structure System: A Method for Managing the Design of Complex Systems," IEEE Transactions on Engineering Management, Vol. 28 (3), 1981, pp. 71-74.

108. S. D. Eppinger, D. E. Whitney, R. P. Smith, and D. A. Gebala, "A Model-based Method for Organizing Tasks in Product Development," Research in Engineering Design, Vol. 6(1), 1994, pp. 1-13.

109. P. J. Clarkson, C. Simons, and C. M. Eckert, "Predicting Change Propagation in Complex Design", Proceedings, 13th International Conference on Design Theory and Methodology (DETC'01), ASME Design Engineering Technical Conferences and Computers and Information in Engineering Conference, Pittsburgh, Pennsylvania, USA, 2001

110. L. Briand, Y. Labiche, "Impact Analysis and Change Management of UML Models", Technical Report SCE-03-01, Carleton University, Feb. 2003.