2007

An Entropy-Based Measurement Framework For Component-Based Hierarchical Systems

Ozgur Aktunc
University of Alabama at Birmingham

AN ENTROPY-BASED MEASUREMENT FRAMEWORK FOR COMPONENT-BASED HIERARCHICAL SYSTEMS

by

OZGUR AKTUNC

MURAT M. TANIK, COMMITTEE CHAIR
GARY J. GRIMES
CHITTOOR V. RAMAMOORTHY
MURAT N. TANJU
GREGG L. VAUGHN
B. EARL WELLS

A DISSERTATION

Submitted to the graduate faculty of The University of Alabama at Birmingham
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

BIRMINGHAM, ALABAMA

2007

AN ENTROPY-BASED MEASUREMENT FRAMEWORK FOR COMPONENT-BASED HIERARCHICAL SYSTEMS

OZGUR AKTUNC

COMPUTER ENGINEERING

ABSTRACT

Component-based software systems are becoming larger and more complex. The quantification of these systems is possible through representation in a hierarchical structure with the use of software metrics. Metrics provide information about such features as complexity, design, quality, size, development time, effort, and cost. Entropy-based software metrics promise important improvements in measuring complexity, design quality, and information flow. The entropy-based measurement framework, the core of this dissertation, offers quantitative representation and decomposition processes to apply the entropy-based metrics that we developed: Design Information Content, Interaction Complexity, Interaction Coupling, and Interaction Cohesion.

The objective is to achieve a hierarchical representation of the system in which we can see the logic flow and quantify the interaction among the components. The framework is useful to both the designer and the user in the design and maintenance phases to detect structural complexity issues and comprehend the interaction of the software modules. Another motivation of this work is to lead the way to the development of measurement software which uses entropy-based methods. The representation-decomposition methods are based on graph theory, mainly cubic control flowgraphs, and the computation methods are based on information theory, mainly entropy; thus, the entropy-based measurement framework has a solid foundation to become an analytical measurement framework for component-based systems.

DEDICATION

I would like to dedicate this dissertation to my parents, Bilgen and Cevdet Aktunc, and Teri for their support, patience, and encouragement throughout my Ph.D. program. I could not have completed without them.

ACKNOWLEDGEMENTS

I would like to acknowledge many people for helping me during my doctoral work. I would especially like to thank my advisor, Dr. Murat M. Tanik, for his generous time, commitment, and friendship. Throughout my doctoral work he encouraged me to develop independent thinking and research skills. He stimulated my analytical thinking and greatly assisted me with scientific writing. I would like to express my gratitude to Dr. Murat N. Tanju for the countless hours of discussions and reviews during my research and writing. I thank my other committee members, Dr. Grimes, Dr. Ramamoorthy, Dr. Vaughn, and Dr. Wells for their wonderful comments during the preparation of this dissertation. I would like to thank Dr. Jannett for his invaluable help as the graduate program coordinator.

I would like to extend my acknowledgements to people and friends that helped me through this study. Many thanks to Abidin Yildirim, who provided me the opportunity to work at Vision Science Research Center (VSRC). I thank the director of the VSRC, Dr. Keyser, and also V. Brooks for their support to me. I acknowledge Dr. Jololian and Professor Jones for their support during my graduate assistantship. I thank my friends and colleagues Dr. Seker, B. Ozaydin, Z. Demirezen, C. Togay, R. Sadasivam, D. Gurler, and Dr. U. Tanik for their enthusiastic discussions and help throughout this study. I express my gratitude to Mrs. O. Tanik for her encouraging comments. I thank my brother, Erdem, and sister-in-law, Umut. Finally, I would like to thank all my family, teachers, students, and friends, who supported me and may not be listed here.

TABLE OF CONTENTS

ABSTRACT iii

DEDICATION iv

ACKNOWLEDGEMENTS v

LIST OF TABLES x

LIST OF FIGURES xii

1 INTRODUCTION 1

1.1 Objectives 2

1.2 Methodology 3

1.3 Outline of the Dissertation 5

2 THE FOUNDATION 8

2.1 Software Measurement Theory 9

2.1.1 Defining Measurement 11

2.1.2 Measurement Concepts and Definitions 12

2.1.3 Representational Measurement Theory 13

2.1.3.1 Empirical Relational System 14

2.1.3.2 Numerical Relational System 15

2.1.3.3 Measure 16

2.1.3.4 Scale 17

2.1.3.5 Scale Types and Meaningfulness 18

2.1.4 Objective versus Subjective Measures....20
2.2 Information Theory....20
2.2.1 Formal Definition of Entropy....22
2.2.2 Shannon Entropy....23
2.2.3 Entropy of a Graph....24
2.2.4 Entropy Measurement....26
2.2.5 Conditional Entropy and Mutual Information....27
2.3 Decomposition Theory....28
2.3.1 Decomposition with Cubic Control Flowgraphs....29
2.3.1.1 Cubic Control Flowgraphs....30
2.3.1.2 Structured Programming Constructs by CCFGs....30
2.3.1.3 Composition and Decomposition Principles of CCFGs....32
2.3.2 Decomposition of Structured Programs into CIUs....33
2.3.2.1 A Complete Decomposition of a Program into its Constructs....33
3 HIERARCHY, COMPLEXITY, AND COMPONENT-BASED SOFTWARE....39
3.1 Hierarchy of Systems....40
3.2 Complexity of Software....42
3.3 Types of Software Complexities....44
3.4 Component-Based Software Development....46
4 SOFTWARE METRICS....51
4.1 The Need for Software Metrics....51
4.2 Definition of Software Metrics....52
4.3 Benefits of Software Metrics....53

4.4 Attributes of Software Metrics....54
4.5 How and What to Measure....55
4.6 Software Metrics Classification....59
4.6.1 Product Metrics....62
4.6.1.1 Metrics for the Analysis Model....63
4.6.1.2 Metrics for the Design Model....66
4.6.1.3 Metrics for Source Code....68
4.6.1.4 Quality Metrics....76
4.6.2 Process Metrics....78
4.6.2.1 Empirical Models....79
4.6.2.2 Statistical Models....79
4.6.2.3 Composite Models....80
4.6.2.4 Reliability Models....80
4.6.3 Resource Metrics....81
4.7 Metrics for Component-Based Systems....82
5 ENTROPIC MEASUREMENT FRAMEWORK....84
5.1 Entropy Measurement in Software....85
5.2 Representation of Flowgraphs using Connectivity Matrix....86
5.2.1 Introduction....86
5.2.2 Adjacency Matrix....87
5.2.3 Obtaining a Connectivity Matrix Using an Adjacency Matrix....90
5.2.4 Warshall's Algorithm to Compute a Connectivity Matrix from an Adjacency Matrix....94
5.2.5 Flowgraph Notation....95

5.3 Entropy and Information Content of Software 99
5.3.1 Entropy of Software 99
5.3.2 Information Content of Software Systems 101
5.3.3 Measurement of Information Content in Software 102
5.3.4 How to Measure a Software Component's Entropy 103
5.4 Representation of Programs with Control Flowgraphs 106
5.5 Measurement of the Structural Complexity of Software 109
5.6 Using Entropy to Examine Structural Complexity 112
6 CASE STUDY, CONCLUSION, AND FUTURE WORK 115
6.1 Calculations of Edge Probabilities and Stationary Probability Distribution for the Nodes Using Perron-Frobenius Theorem 119
6.2 Calculation of the Metrics 126
6.2.1 Design Information Content 126
6.2.2 Interaction Complexity 127
6.2.3 Interaction Cohesion 131
6.2.4 Interaction Coupling 134
6.3 Comparison of Information Theory-Based Metrics to Counting Metrics for Graphs 138
6.4 The Program Used in the Case Study 141
6.5 Validation 144
6.6 Conclusion 146
6.7 Future Work 148
REFERENCES 150
APPENDIX 159

LIST OF TABLES

Table *Page*

2.1 Scale Types 18

4.1 Entities and Attributes 61

4.2 Metrics for Component-Based Software 82

5.1 Lexical Analysis for Calculating the Information Content 105

6.1 Edge Probabilities for the CCFG 122

6.2 Node Probabilities for the CCFG 122

6.3 Node Entropies for the CCFG 125

6.4 Edge Entropies for the CCFG 125

6.5 Interaction Complexities of the Modules 130

6.6 Comparison of the Interaction Complexity Metrics to the Counting Metrics 139

6.7 Comparison of the Interaction Cohesion Metrics to the Counting Metrics 140

6.8 Comparison of the Interaction Coupling Metrics to the Counting Metrics 140

6.9 The Results of Metric Analysis of the Modules by Analyst4j 143

A.1 Edge Weights 161

A.2 Node Probabilities for the CCFG 163

A.3 Edge Probabilities for the CCFG 164

A.4 Node Entropies for the CCFG 164

A.5 Edge Entropies for the CCFG 164

A.6 Comparison of the Interaction Complexity Metrics
to the Counting Metrics 167

A.7 Comparison of the Interaction Cohesion Metrics
to the Counting Metrics 167

A.8 Comparison of the Interaction Coupling Metrics
to the Counting Metrics 167

LIST OF FIGURES

Figure *Page*

1.1 Basic Representation of the Measurement Framework 4

2.1 The Three Constructs of Structured Programming and their Equivalent CCFGs 31

2.2 A Program, its Control Flowgraph and Cubic Control Flowgraph....34

2.3 The Two Components Obtained from the CCFG shown in Figure 2.2 36

2.4 First Level Decomposition of the CCFG shown in Figure 2.2 37

2.5 Second Level Decomposition of the CCFG shown in Figure 2.2....37

2.6 All of the Prime Components of the CCFG shown in Figure 2.2....38

3.1 Classification of Software Complexity....46

3.2 A Process Model that Supports CBSE....49

4.1 Metrics Architecture 57

4.2 Example Graph for Cyclomatic Complexity 74

5.1 Graph G....88

5.2 The Adjacency Matrix of Graph G 88

5.3 The Connectivity Matrix of Graph G....90

5.4 A Mesh Network....92

5.5 A Flowgraph and its Corresponding Weighted Adjacency Matrix 96

5.6 Adjacency Matrix of the Flowgraph shown in Figure 5.5 97

5.7 Adjacency Matrix and RS-1....99

6.1 The Sample Program and its Control Flowgraph 116
6.2 CCFG of the Program 117
6.3 CCFG of the Program with Edge Weights 118
6.4 Modules of the CCFG 124
6.5 Module 1 128
6.6 Module 2 129
6.7 Module 3 129
6.8 Module 4 130
6.9 Intramodule Edges of Module 1 132
6.10 Intramodule Edges of Module 2 133
6.11 Intramodule Edges of Module 3 133
6.12 Intramodule Edges of Module 4 134
6.13 Intermodule Edges of Module 1 136
6.14 Intermodule Edges of Module 2 136
6.15 Intermodule Edges of Module 3 137
6.16 Intermodule Edges of Module 4 138
A.1 A CCFG with 6 Nodes 161
A.2 The CIUs of the CCFG 162

CHAPTER 1

INTRODUCTION

Component-Based Software Engineering (CBSE) emerged in the late 1990s with the objective of increasing the reuse of existing components. The main objectives of Component-Based Development (CBD) are coping with the inherent complexity of large software systems and enabling the reuse of components. The CBD process includes defining, implementing, and integrating loosely coupled components into systems (Sommerville 2007).

As component-based systems are becoming more complex, representation/decomposition methods to represent these systems in a hierarchical structure and metrics to give insight into their design qualities are becoming more crucial. Knowing the complexity of the software enables us to predict the cost of testing and maintenance, the number of errors left in the software, and the size of the final program; to assess the development time, effort and cost; and to and identify critical modules or parts of the software (Podgorelec and Hericko 2007). There has been reasonable progress in the software metrics area to quantify various aspects of software, such as the complexity of the system. However, much work is needed for the assessment of CBS systems. Entropy metrics promise important improvements in predicting errors and providing insight into design quality (Bianchi and others 2001). They also provide a quantitative means of measuring information flow in the software system (Masri and Podgurski 2006).

Entropy has been defined in terms of the information content of software and has been used to measure software code complexity by Harrison (1992), Torres and Samadzadeh (1991), and Davis and LeBlanc (1998). However, much improvement has been needed in the application of entropic metrics to component-based systems. The typical decomposition techniques used with entropy metrics are not analytical. As we observe in Davis and LeBlanc (1998) and Chapin's (2002) works, in which they applied entropy metrics to a system decomposed by chunk-equivalent classes, that approach is not an analytic way to decompose a system. We also recognize other problems that contradict the nature of components, such as applying entropy metrics to components' source code. Software engineering practice dictates that components have a black-box nature that hides their source code from users. One clear advantage of entropy metrics is their applicability to component-based software systems without requiring the source code. The framework presented in this dissertation can be applied during the design and evolution phases of such systems to assess attributes of the systems such as information content, complexity, coupling, and cohesion.

1.1 Objectives

The current approaches to apply entropy metrics to component-based systems lack an analytical measurement framework. The proposed measurement framework offers quantitative representation and decomposition methods to apply entropy-based metrics. By using our entropy-based measurement framework, we aim to achieve a hierarchical representation of the system in which we can see the logic flow and quantify the interaction between the components. Our framework uses Cubic Control Flowgraphs (CCFG) to

represent the system. These CCFGs have a set of rules for decomposition. This framework would be helpful to both designers and users to represent the system in a comprehensible form and to apply the necessary measures. The framework targets the design and evolution phases in component-based systems to detect structural complexity issues and comprehend the interaction of the software modules. Another motivation of this dissertation is to lead the way in the development of measurement software using entropy-based methods.

1.2 Methodology

Our measurement framework consists mainly of three phases: representation, decomposition, and computation. In short, we represent the system with CCFGs and apply their decomposition methods to an irreducible decomposed system. The edge and node probabilities are then computed using the Perron-Frobenius theorem. The entropy values are calculated using Shannon's entropy formula. The final step is the application of the entropy metrics that we developed: Design Information Content, Interaction Complexity, Interaction Cohesion, and Interaction Coupling. The entropy-based measurement framework is summarized in Figure 1.1.

By using cubic flowgraph representation and decomposition, we represent a component-based system with Component Integration Units (CIU) that are composed of components. Therefore, a CIU can be a composite component, a subsystem, or a system. The CIUs of a CBS can be represented with control flowgraphs (CFG), specifically with CCFGs. The composition/decomposition principles that are derived from the cubic graph formalism may be used to investigate the integration of the system.

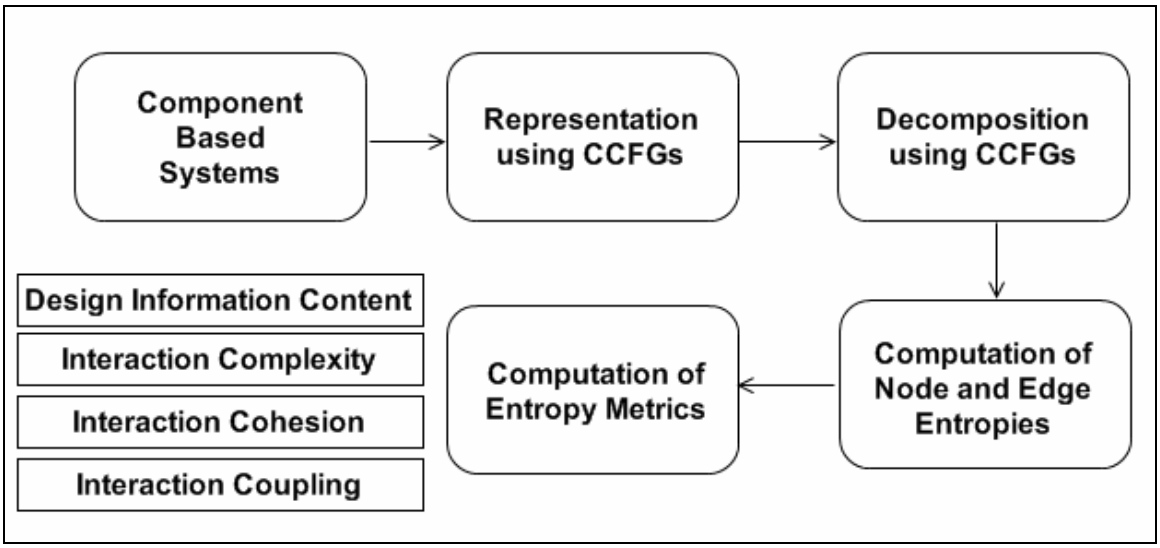


Figure 1.1 Basic representation of the measurement framework

The computation phase consists of two sub-phases; the first one to compute the node and edge entropies of the CCFGs that represent the system, and the second one to compute the entropy metrics. During the first computation phase, the weighted adjacency matrix is created from the CCFGs and the weights of the parameters of interest. The Perron-Frobenius theorem provides the rules to find the stationary state (node) and branch (edge) probabilities. It is necessary to find the left and right eigenvectors of the weighted adjacency matrix to determine the probabilities. Entropy is a function of probability and, by using Shannon's formula for entropy, the edge and node entropies can be computed. The second phase of the computation includes applying the four entropy metrics that we developed: Design Interaction Content, Interaction Complexity, Interaction Cohesion, and Interaction Coupling. These metrics assess certain attributes of the systems, such as performance, complexity, coupling, and cohesion. By applying entropy metrics to a system decomposed by CCFG methods, it is possible to achieve a good representation of the

overall structure of the system. This structural view not only includes static structural information but information about the logic flow and interaction among the components.

The framework presented by Kitchenham and others (1995) was used for the validation of metrics. The validation framework can help to understand how to validate a measure, how to assess the validation work of others, and when to apply a measure. The theoretical validation of the framework is based on the fact that we use two established theories in our framework: the entropy-based metrics are based directly on information theory, and the representation/decomposition methods are based on graph theory. The empirical validation has been done in the case study; however, there may be more work to do in this area to reach a complete validation framework.

1.3 Outline of the Dissertation

This dissertation is composed of six chapters including this introductory chapter. In this section, the outline of the work done in each of the remaining five chapters is summarized.

Chapter 2 summarizes the theoretical foundation of this dissertation, which consists of software measurement theory, information theory, and decomposition theory. The measurement theory section includes some fundamental definitions of measures and metrics that are used throughout the dissertation. It explains the representational measurement theory that is followed as a guideline in this dissertation. The next section, regarding information theory, explains the notion and measurement of entropy. The final section in Chapter 2 begins with a brief overview of the decomposition of systems. CFGs

and CCFGs are then explained with an example of their representation and decomposition methods.

Chapter 3 discusses the hierarchy of systems, complexity issues in software, how to deal with these issues, and component-based development. This chapter proves that representing software as a hierarchical system is necessary in order to apply measurement methods. The decomposition and abstraction of a software system is considered an essential part of the measurement framework described in the later chapters.

Chapter 4 addresses the need for software metrics, then defines software metrics. The benefits of metrics are explained through the perspectives of the organizations and individual developers. The next section attempts to uncover the answers to the questions of how and what to measure with software metrics. These answers are followed by a software metrics taxonomy with three main classes: product, process, and resource metrics. The last section of Chapter 4 focuses on metrics that could be used for CBD.

Chapter 5 is an introduction to the entropy-based measurement framework. The chapter begins with a review of efforts to use entropy in software measurement. It continues with a section about flowgraph representation and how to represent flowgraphs with matrices. This section builds the transition from programs to flowgraphs, and from flowgraphs to matrices that are used to compute the metrics. The chapter continues with a summary of the measurement of information content in software and related issues, such as entropy measurement, lexical analysis, and Halstead metrics. Representing software using CCFGs and the measurement of coupling and cohesion is viewed from a structural complexity perspective. The last section summarizes the idea of how the information-

theoretical approach based on entropy concepts can help quantify some of the fundamental structural complexity measures in software engineering.

Chapter 6 consists mainly of the case study the measurement framework was built on, the validation of the metrics, the summary of the work, the contribution to the body of knowledge, and future research directions.

The Appendix includes another example where the entropy-based metrics were applied to a program. This example contributes to the validation of the framework.

CHAPTER 2

THE FOUNDATION

This chapter is an overview of the three areas this research was built on: measurement, information, and representation/decomposition theories. It is necessary to review some of the important concepts of each theory that are relevant to the dissertation topic in order to build an abstract understanding of software metrics using information theory.

The chapter begins with an introduction to software measurement theory. It proceeds with definitions of software measurement abstractions, metrics, measures, and scale types. This section summarizes the representational measurement theory that is followed as a guideline for software measurement. The section concludes with a comparison of objective and subjective measures.

The next section summarizes the necessary information related to information theory, specifically entropy, to understand the software metrics used in the framework. It begins with the definition of entropy and of Shannon Entropy and its properties. The section continues with the measurement of the entropy of graphs and concludes with an example of entropy measurement and two other related concepts, conditional entropy and mutual information.

The final section of this chapter discusses representation/decomposition theory. This section begins with a brief overview of representation/decomposition of systems. It then explains the method of decomposition using flowgraphs that are used to represent

the skeleton of component-based software. The method is explained with a detailed example in the remainder of the section.

2.1 Software Measurement Theory

Software measurement theory provides and validates a body of knowledge about software engineering that can be used to understand, monitor, control, and improve software processes and products. Measurement theory is a convenient theoretical framework used to explicitly define the underlying theories upon which software engineering measures are based.

In the last three decades, several methods have been proposed to identify, characterize, and measure the characteristics of software processes and products. Surveys about software measurement have been written by Zuse (1997), Fenton and Pfleeger (1998), and Munson (2003). The methods of measurement have been utilized to evaluate the software engineering measures proposed in the literature and to establish criteria for the statistical techniques. These statistical techniques are used in data analysis and in the search for patterns.

Software engineering differs from other engineering disciplines in a number of aspects that have important consequences for software measurement. Software engineering is a relatively young discipline compared to mature engineering disciplines with well-established theories, methods, and models. In addition, in empirical software engineering, as in other empirical sciences (e.g., experimental psychology), measurement is noisy, uncertain, and difficult. For instance, in a number of software measurement applications, the repeatability of results may not be achieved. As a consequence, one cannot expect soft-

ware engineering measurement models and theories to be of the same nature, precision, and accuracy as those developed for more traditional branches of engineering. Another difference is the fact that there are several types of software development for various application areas and purposes; thus, software measurement models may not be valid on a general basis, like those used in other engineering branches (Briand and Morasca 1997).

Measurement is necessary in software engineering. The main reason for measurement is to obtain data that helps us to better control the schedule, cost, and quality of software products. It is important to be able to consistently count and measure basic entities that are directly measurable, such as size, defects, effort, and time.

Consistent measurements provide data for performing the following activities:

- Quantitatively expressing requirements, goals, and acceptance criteria.
- Monitoring progress and anticipating problems.
- Quantifying trade-offs incurred in allocating resources.
- Predicting the software attributes for schedule, cost, and quality (Florac 1992).

A software measurement method typically has two phases: creation of an abstraction of the software and measurement of that abstraction. The abstraction is created to characterize the attributes of interest. The measure is designed to quantify the attributes of interest in a way that facilitates further analysis (Allen and others 2007). This dissertation follows the mentioned approach by providing methods to decompose and represent software and to apply formal methods of measurement to quantify the parameters of interest using entropy metrics.

2.1.1 Defining Measurement

Measurement has a long history in the natural sciences. At the end of the last century, the physicist, Lord Kelvin, stated, “When you can measure what you are speaking about, and express it into numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: It may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the stage of science (Kelvin 1891).” The history of software measurement goes back to the late 1960s and early 1970s. One of the first papers about software complexity was published by Rubey and Hartwick (1968). There is no reference to an earlier publication about software complexity or measurement.

Fenton and Pfleeger provide the following definition of measurement: “Formally, we define measurement as a mapping from the empirical world to the formal, relational world. Consequently, a measure is the number or symbol assigned to an entity by this mapping in order to characterize an attribute (Fenton and Pfleeger 1998).”

There are two broad types of measurement: direct and indirect. Direct measurement of an attribute is measurement that does not depend on the measurement of any other attribute. Indirect measurement of an attribute involves the measurement of one or more other attributes (Fenton 1994). While some attributes can be measured directly, we usually obtain more in-depth results when we measure indirectly. The following section states some of the important measurement concepts and definitions.

2.1.2 Measurement Concepts and Definitions

As stated in the previous section, measurement is based on two concepts:

- Entity. An entity may be a physical object (e.g., a program), an event that occurs at a specified time (e.g., a milestone), or an action that spans a time interval (e.g., the testing phase of a software project).
- Attribute. An attribute is a characteristic or property of an entity (e.g., the size of a program, the time required during testing).

Entities consist of products and processes. Requirements, specifications, designs, code, and test sets are all product entities. Domain analysis and all activities related to software creation, such as coding and testing, are processes. All products and processes have specific attributes. For instance, product attributes include size, complexity, cohesion, coupling, and reliability, while process attributes include time, effort, and cost.

The following definitions of measure and metrics are important since they are mentioned frequently throughout the dissertation. The following definition of measure is based on the definition in the previous section. The definition of metric is based on IEEE standard 1061 (IEEE 1992):

- Measure. A measure is the result of the measurement process, so it is the assignment of a value to an entity with the goal of characterizing a specified attribute.
- Metric. A metric is a measurement function, and a software quality metric is a function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which the software possesses a given attribute that affects its quality.

The following formal definitions of measure and metrics are offered by the Software Engineering Institute (Ford 1993):

Definition 2.1 (Measure): Let *A* be a set of physical or empirical objects. Let *B* be a set of formal objects, such as numbers. A measure *m* is defined to be a one-to-one mapping $m : A \rightarrow B$. The requirement that the measure be a one-to-one mapping guarantees that every object has a measure, and every object has only one measure. It does not require that every number (in set *B*) be the measure of some object (in set *A*).

Definition 2.2 (Metric): Let *A* be a set of objects, let *R* be the set of real numbers, and let $m : A \rightarrow R$ be a measure. Then *m* is a metric if and only if it satisfies these three properties

$$m(x, y) = 0 \text{ for } x = y \tag{2.1}$$

$$m(x, y) = m(y, x) \text{ for all } x, y \tag{2.2}$$

$$m(x, z) \equiv m(x, y) + m(y, z) \text{ for all } x, y, z. \tag{2.3}$$

Other important concepts, including empirical and numerical representation systems, scale, and scale types, will be explained in the following section.

2.1.3 Representational Measurement Theory

Representational measurement theory formalizes the intuitive, empirical knowledge about an attribute of a set of entities and the quantitative, numerical knowledge about the attribute. The intuitive knowledge is captured via an empirical relational system and the quantitative knowledge via a numerical relational system. Both the empirical and numerical relational systems are built using set algebra. A measure links the empirical relational system with the numerical relational system in such a way that inconsistencies

are not possible, as formalized by the Representation Condition. In general, many measures may exist that equally quantify one's intuition about an attribute of an entity (e.g., weight can be measured in kilograms, grams, pounds, ounces, etc.). The relationships among the admissible measures for an attribute of an entity are illustrated in Section 2.1.3.4. Classification of measures into different categories is described in Section 2.1.3.5. Objectivity and subjectivity of measures are discussed in Section 2.1.4.

A relational system *A* is an ordered tuple $(A, R_1,..., R_n, o_1,..., o_m)$, where *A* is a nonempty set of objects, the R_i, $i = 1,..., n$ are ki-ary relations on *A*, and the o_j, $j = 1,..., m$ are closed binary operations. For measurement there are two types of relational systems: the empirical and formal relational systems.

2.1.3.1 Empirical Relational System

An empirical relational system for an attribute of a set of entities is defined through the concepts of sets of entities, relations among entities, and operations among entities. Therefore, an empirical relational system is defined as an ordered tuple $ERS = (E, R_1,..., R_n, o_1,..., o_m)$, as explained below:

- *E* is the set of entities, i.e., the set of objects of interest subject to measurement with respect to the attributes of interest.
- $R_1,..., R_n$ are empirical relations. Each empirical relation R_i has an *arity* n_i, so $R_i \subseteq E^{n_i}$, i.e., R_i is a subset of the cartesian product of the set of entities, $E \times E \times ... \times E$ n_i times.

- $o_1,...,o_m$ are binary operations among entities. Each binary operation o_j is a function $o_j : E \times E \to E$. Therefore, its result is an entity, so we have $e_3 = e_1 o_j e_2$ (infix notation is usually used for these operations). As an additional assumption, all binary operations o_j's are closed, i.e., they are defined for any pair of entities (e_1, e_2).

The empirical relations do not involve any numerical values. Empirical relations do not focus on the comparison of values obtained by measuring e_1's and e_2's sizes, but they state the intuitive understanding and knowledge relating to e_1's and e_2's sizes and that e_3 is the concatenation of e_1 and e_2. Since this knowledge is intuitive, empirical relational systems are built in a somewhat subjective way, as intuition varies among individuals.

2.1.3.2 Numerical Relational System

The intuitive knowledge of the empirical relational system is translated into the numerical relational system, which involves numerical values. A numerical relational system is defined as an ordered tuple $NRS = (V, S_1,..., S_n, p_1,..., p_m)$:

- *V is* the set of values that can be obtained as a result of measures.
- $S_1,..., S_n$ are numerical relations. Each numerical relation S_i has the same *arity* n_i of the empirical relation R_i, so $S_i \subseteq V^{ni}$, i.e., S_i is a subset of the n_i times cartesian product of the set of values.

- $p_1, ..., p_m$ are binary operations among values. Each binary operation p_j is a function $p_j : V \times V \to V$. Therefore, its result is a value, so we have $v_3 = v_1 p_j v_2$ (infix notation is usually used for these operations). As an additional assumption, all binary operations p_j's are closed, i.e., they are defined for any pair of values (v_1, v_2).

For instance, the set *V* may be the set of nonnegative integer numbers. A binary relation may be greater than, i.e., $>$, so $> \subseteq V \times V$. A binary operation may be the sum between two integer values, i.e., $v_3 = v_1 + v_2$. Therefore, the numerical relational system does not describe the entities or attributes but solely assigns values to them.

2.1.3.3 Measure

The link between the empirical relational system and the numerical relational system can be established using the definition of measure, which associates entities and values. It also includes scale, which associates the elements of the tuple of the empirical relational system with elements of the numerical relational system. A measure is a function $m : E \to V$ that associates a value with each entity.

As defined previously, a measure establishes a link between the set of entities and the set of values, regardless of the relations and operations in the empirical and numerical relational systems. Therefore, a measure for the size of program segments may be inconsistent with our intuitive knowledge about size, as described by the relation *longer-than*. For example, we may have three program segments: e_1, e_2, and e_3, such that $(e_1, e_3) \in$ *longer-than*, $(e_2, e_3) \in$ *longer-than*, $m(e_1) < m(e_2)$, and $m(e_2) < m(e_3)$. This ex-

ample shows that not all measures are appropriate for quantifying intuitive knowledge. The Representation Condition places constraints on measures to avoid counterintuitive results.

Definition 2.3 (Representation Condition): A measure $m(e_1) > m(e_2)$ must satisfy these conditions. The first condition requires that a tuple of entities be in the relation R_i if and only if the tuple of measures computed on those entities is in the relation S_i that corresponds to R_i. Therefore, if the relation > is the one that corresponds to *longer-than*, we have $(e_1, e_2) \in$ *longer-than* if and only if $m(e_1) > m(e_2)$, as one would intuitively expect. The second condition requires that the measure of an entity, obtained with the binary operation o_j from two entities, be obtained by computing the corresponding binary operation p_j on the measures computed on those two entities.

2.1.3.4 Scale

Definition 2.4 (Scale): A scale is a tuple (ERS, NRS, m), where *ERS* is an empirical relational system, *NRS* is a numerical relational system, and $m : E \to V$ is a measure that satisfies the Representation Condition.

Given two relational systems, *A* and *B*, we can ask whether a measure exists, such that (ERS, NRS, m) is a scale. This condition is called the representation problem. If such a measure exists, we can ask how uniquely the measure is defined. This condition is called the uniqueness problem. The uniqueness problem leads to the definition of scale types, such as ordinal or ratio scales.

2.1.3.5 Scale Types and Meaningfulness

Definition 2.5 (Admissible Transformation): Given a scale (ERS, NRS, m), a transformation of a scale *f* is admissible if (ERS, NRS, m') is a scale, where $m' = f \circ m$, m' is the composition of *f* and *m*.

Scale types are also defined by admissible transformations. For real scales, there is a classification of scales according to their admissible transformations as given in Table 2.1 (Roberts 1979).

Table 2.1 Scale Types

Name of the Scale	Transformation g
Nominal Scale	Any one to one g
Ordinal Scale	g : Strictly increasing function
Interval Scale	$g(x) = ax + b$
Ratio Scale	$g(x) = ax$
Absolute Scale	$g(x) = x$

Descriptions of the types of scales are given as:

- A nominal scale measure is a classification of the objects. The only possible transformations are those that preserve the fact that objects are different, i.e., one-to-one transformations.
- An ordinal scale measure is a ranking of the objects based on some ordering criteria.

- An interval scale is a measure that considers the differences between values meaningful, but not the values of the measure itself (e.g., temperature variations measured on a Celsius or Fahrenheit scale).
- A ratio scale is a measure that has a meaningful zero value, and the ratios between values are meaningful.
- An absolute scale measure is such that the entity has a meaningful numerical value that is not affected by transformations (e.g., count of the objects).

These types of measurement scales (e.g., levels of measurement) are ranked from less powerful to more powerful. In particular, the more powerful scales (interval, ratio, and absolute) provide added information and are more useful for measurement purposes.

Some concerns for the user include knowing what scale type is assumed and what the conditions are for the use of a measure on a certain scale level, and determining how a measure that creates numbers can be transformed by certain admissible transformations of scales. Admissible transformations also help to define meaningful statements (Roberts 1979). A statement with measurement values is meaningful if and only if its truth or falsity value is invariant to admissible transformations.

Meaningfulness guarantees that the truth value of statements with measurement values is invariant to admissible scale transformations. For example, if we say that the distance *D1* is twice as long as distance *D2*, then this statement is true or false no matter whether length is measured in meters or yards. Similar issues exist in the area of software measures. This is the case if we want to make statements with measurement values, for example after the application of statistical methods.

2.1.4 Objective versus Subjective Measures

A distinction should be made between objective and subjective measures. The distinction is based on the way measures are defined and collected. Objective measures should be precisely defined and may be collected using automated tools, while subjective measures leave room for interpretation and require human involvement. For instance, ranking a failure as catastrophic, severe, non-critical, or cosmetic, may be done based on human judgment. As a consequence, subjective measures are believed to be of lower quality than objective ones. However, there are a number of cases in which objective measures cannot be collected, so subjective measures are the only way to collect information that may be important and useful. Efforts must be made to minimize discrepancies among the values provided by different people. Guidelines can be utilized in order to limit the amount of variability in the values that different people assign to a measure. As an example, the definition of values for an ordinal measure should include an explanation of what those values mean. If we have a size measure with the levels of small, medium, and large, we need to explain how to determine each of the three levels. Otherwise, people with different intuitions may provide contrary values when ranking a program according to that measure.

2.2 Information Theory

This dissertation is based on the application of the principles of information theory to the field of software measurement. In this section, relevant information related to information theory, specifically entropy, is summarized.

Most software measurement methods rely on counting. However, there have been studies of new measures based on information theory (Khoshgoftaar and Allen 1994). Repetitive patterns have low information content since they can be described more easily compared to random patterns with higher entropies. Entropy-based measures can be applied to several software entities, such as control flowgraphs, object-oriented class interfaces, or code itself.

Entropy is a concept in thermodynamics, statistical mechanics, and information theory. In information theory, entropy is related to the randomness of a signal or an event. The randomness of a signal is associated with the information carried by the signal. If a portion of English text is considered as an example, it can be seen as an encoded string of letters, spaces, and punctuation (e.g., our signal is a string of characters). Although there are some frequently occurring characters, it is hard to predict consecutive characters. Hence, there is randomness in the signal. This measure of randomness is defined as entropy by Shannon in his paper, "A Mathematical Theory of Communication (1949)."

The following assumptions were used by Shannon to derive his definition of entropy:

- The entropy has to be continuous. Changing one of the probabilities by a small amount should make the entropy change only by a small amount.
- If the outcomes of an event are equally likely, then an increase in the number of letters in the mentioned example should always increase the entropy.

2.2.1 Formal Definition of Entropy

In an experiment *S*, the probability of an event $A(P(A))$ is defined as the measure of uncertainty of the occurrence or non-occurrence of *A* (Papoulis and Pillai 2002). We are certain about the occurrence of the event *A* when either $P(A) \cong 0.999$ or 0.1; our uncertainty is maximum when $P(A) = 0.5$. Here we discuss the problem of assigning a measure for the uncertainty of the occurrence or non-occurrence of any event A_i of a partition *U* of the sample space *S*, where a partition is a collection of mutually exclusive events whose union equals *S*. The measure of uncertainty about *U* will be denoted by $H(U)$ and will be called the entropy of the partition *U*.

The function $H(U)$ has been derived from a number of postulates based on the heuristic understanding of uncertainty. The following is a typical set of such postulates:

- The term $H(U)$ is a continuous function of $p_i = P(A_i)$.
- If $p_1 = ... = p_N = 1/N$, then *H(U)* is an increasing function of *N*.
- If a new partition *B* is formed by subdividing one of the sets of *U*, then $H(B) \geq H(U)$.

It can be shown that the sum

$$H_{nm}(P*Q) \leq H_n(P) + H_m(Q) \tag{2.4}$$

satisfies these postulates. Shannon referred to this function as the actual definition for entropy and said it could determine the minimum channel capacity needed to transmit an information source of encoded binary digits reliably. This introduction of entropy, in terms of postulates, establishes a link between entropy and our heuristic understanding of uncertainty.

2.2.2 Shannon Entropy

The Shannon Entropy is a well-known, widely-used measure of information. This section presents a brief description of the Shannon Entropy and some of its properties that are essential to its use as a measure of software information. The Shannon Entropy, H_n, is defined as

$$H_n(P) = -\sum_{k=1}^{n} p_k \log_2 p_k \tag{2.5}$$

where

$$p_k \geq 0 \ \ (k = 1, 2, ..., n) \text{ and } \sum_{k=1}^{n} p_k = 1 \ \ (n \geq 1).$$

Because a logarithm to the base 2 is used, the resulting unit of information is called a bit (a contraction of binary unit). The Shannon Entropy satisfies many desirable properties. The following properties are of special interest to our topic (Aczel and Daroczy 1975):

1. *Nonnegativity*: Receiving information about an experiment does not make an individual more ignorant than before

$$H_n(P) \geq 0 . \tag{2.6}$$

2. *Symmetry*: The amount of information is invariant to a change in the order of events

$$H_n(P) = H_n(p_k(1), p_k(2), ..., p_k(n)) \tag{2.7}$$

where *k* is an arbitrary permutation of the indices on the probabilities.

3. *Normality*: A simple alternative, which in this case is an experiment with two outcomes of equal probability, 0.5, results in one unit of information

$$H_2(0.5, 0.5) = 1. \tag{2.8}$$

4. *Expansibility*: Additional outcomes with zero probability do not change the uncertainty of the outcome of an experiment

$$H_n(P) = H_{n+1}(p_1, p_2, ..., p_n, 0). \tag{2.9}$$

5. *Decisivity*: There is no uncertainty in an experiment with two outcomes: one of them of probability 1, the other of probability 0

$$H_2(1,0) = 0. \tag{2.10}$$

6. *Additivity*: The information, expected from two independent experiments, is the sum of the information expected from the individual experiments

$$H_{nm}(P*Q) = H_n(P) + H_m(Q). \tag{2.11}$$

7. *Subadditivity*: The information, expected from two experiments, is not greater than the sum of the information expected from the individual experiments

$$H_{nm}(P*Q) \le H_n(P) + H_m(Q). \tag{2.12}$$

8. *Maximality*: The entropy is greatest when all outcomes have equal probabilities

$$H_n(P) = H_n(\frac{1}{n}, \frac{1}{n}, ..., \frac{1}{n}). \tag{2.13}$$

There are several other entropies that are different from and less popular than the Shannon Entropy. The definitions of these entropies are not used in the dissertation.

2.2.3 Entropy of a Graph

Many software measurements are based on graphs, in particular cubic flowgraphs (Zuse 1990). An attribute that characterizes the complexity of a graph can partition the set of vertices (edges) into equivalence classes (Davis and LeBlanc 1988). Information theory-based measures model the vertices of a graph as samples from a discrete probabil-

ity distribution estimated from the partition. An entropy measure of this distribution can be interpreted as the degree of surprise once expects from an arbitrary vertex in the graph, i.e., the average information of the graph per vertex.

Graph entropy, $H(G,P)$, is an information-theoretic function on a graph *G* with a probability distribution *P* on its vertex set. The definition of a vertex packing polytope is necessary (vertex packing refers to independent sets of nodes, also called stable sets). The following definitions are given by Korner (1971):

Definition 2.6 (The vertex packing polytope): $VP(G)$ of a graph *G* is the convex hull of the characteristic vectors of stable sets of *G*.

Definition 2.7: Let *G* be a graph on vertex set $V(G) = \{1,...,n\}$ and let

$$P = \{p_1,..., p_n\} \tag{2.14}$$

be a probability distribution on *V(G)* (e.g., $p_1 + ... + p_n = 1$ and $p_i \geq 0$ for all *i*). The entropy of *G* with respect to *P* is then defined as

$$H(G,P) = \min_{a \in VP(G), a>0} \left| \sum_{i=1}^{n} p_i \log \frac{1}{a_i} \right. . \tag{2.15}$$

Additional entropy measures of graphs, such as automorphism entropy and chromatic entropy are useful to determine the information content of graphs (Mowshowitz 1968). Automorphism entropy, H_g, informally, is a measure of the symmetry of a graph. An automorphism, α, is a one-to-one mapping of vertices of a graph, *G*, to itself that preserves adjacency. In other words, if $[x, y]$ is an edge of *G*, then $[x\alpha, y\alpha]$ is also an edge of *G*. For example, if a graph is a reflection of itself, then the mapping of each vertex of that graph to its reflection is an automorphism. The set of all automorphisms on a graph form a group. The orbits of that automorphism group partition the vertices into equivalence

classes. Assuming that vertices are independent and identically distributed, the entropy of this partition can be calculated using

$$H = -\sum_{i=1}^{k} \frac{n_i}{n} \log \frac{n_i}{n}. \tag{2.16}$$

Chromatic entropy, H_c, informally, is the information content of a graph connection. A coloring scheme for a graph with a chromatic number, *k*, partitions the *n* vertices of the graph, *G*, into *k* equivalence classes. Let n_i be the number of vertices in the i^{th} equivalence class. The entropy of the partition can be calculated using (2.16). Since a coloring scheme is not unique, chromatic entropy is defined as the minimum entropy of all possible decompositions

$$H_c = \min\left(-\sum_{i=1}^{k} \frac{n_i}{n} \log \frac{n_i}{n} \right). \tag{2.17}$$

2.2.4 Entropy Measurement

In this section, an entropy measurement example is given, followed by two important definitions related to entropy: mutual information and conditional entropy.

Example: Average information content in the English language.

a. Calculate the average information in bits/character for the English language, assuming that each of the 26 characters in the alphabet occurs with equal likelihood. Neglect spaces and punctuation. Using (2.5), we could calculate the average information content

$$H = -\sum_{k=0}^{25} \frac{1}{26} \log_2 \left(\frac{1}{26} \right) = 4.7 \text{ bits/character.}$$

b. Since the alphabetic characters do not appear with equal frequency, the answer in part (a) will represent the upper bound on average information content per character. Repeat part (a) under the assumption that the alphabetic characters occur with the following probabilities:

- $p = 0.10$ for the letters a, e, o, t.
- $p = 0.07$ for the letters h, i, n, r, s.
- $p = 0.02$ for the letters c, d, f, l, m, p, u, y.
- $p = 0.01$ for the letters b, g, j, k, q, v, w, x, z.

The entropy of the example used above is

$$H = -\left(4 \times 0.1 \log_2 0.1 + 5 \times 0.07 \log_2 0.07 + 8 \times 0.02 \log_2 0.02 + 9 \times 0.01 \log_2 0.01\right)$$
$$= 4.17 \text{ bits/character.}$$

The source rate of information is

$$R = \frac{H}{T} \text{ bps} \tag{2.18}$$

where

H is entropy, and T is the time required to send a message.

To deal with the noisy environment, another notation of information theory, Mutual Information, will be defined.

2.2.5 Conditional Entropy and Mutual Information

Assume that the output y represents a noisy version of the input x. How can the uncertainty about x be measured after observing y? The conditional entropy of x given y is defined as

$$H(y|x) = -\sum_{x,y} p(x,y)\log_2 p(y|x) = H(x,y) - H(x) \tag{2.19}$$

where

$$H(x,y) = -\sum_{x,y} p(x,y)\log_2 p(x,y) \tag{2.20}$$

is the entropy of the joint event, and $p(x,y)$ is the joint probability of x and y.

The conditional entropy in (2.19), also called equivocation, represents the amount of uncertainty about the transmitted message x, having received message y. Shannon showed that the average effective information content, $I(x,y)$, at the receiver is obtained by subtracting the equivocation from the entropy of the source

$$I(x,y) = H(x) - H(x|y). \tag{2.21}$$

This quantity is called the average mutual information. The entropy is a special case of this mutual information, since $I(x,x) = H(x)$. Some properties of the mutual information are

$$I(x,y) = I(y,x) \tag{2.22}$$

$$I(x,y) \geq 0 \tag{2.23}$$

$$I(x,y) = H(y) - H(y|x). \tag{2.24}$$

2.3 Decomposition Theory

In this section, the methods used to decompose the complex systems are explained. Complex systems share certain features, such as having a large number of elements, possessing high dimensionality, and representing an extended range of possibilities. Such systems are hierarchies consisting of different levels, each having its own prin-

ciples, laws, and structures. The four aspects of complex systems according to Herb Simon are (Simon 1969):

- Complex systems are frequently hierarchical.
- The structure of complex systems emerges through evolutionary processes, and hierarchic systems will evolve much more rapidly than non–hierarchic systems.
- Hierarchically organized complex systems may be decomposed into sub-systems for analysis of their behavior.
- Because of their hierarchical nature, complex systems can frequently be described, or represented, in terms of a relatively simple set of symbols (Maxwell and others 2002).

As component-based systems evolve, the project size and complexities also increase. Decomposition of such systems has become crucial to understanding and developing them. Software modeling and decomposition methods deal with complexity by creating representations in hierarchical structures. Metrics provide a set of rules for assessment of a system with respect to desired parameters. Therefore, metrics based on models become critical to the assessment of systems (Seker and Tanik 2004). In the following sections, we will explain the decomposition method we use in this dissertation, Prather's decomposition using CCFGs.

2.3.1 Decomposition with Cubic Control Flowgraphs

Control flowgraphs have been very useful in analyzing programs and their algorithms. They not only represent the logic flow throughout the program, but the overall control structure. Representing a program by graph enables one to use analysis tech-

niques from graph theory. There have been various measures based on the control flowgraphs of programs. Control flowgraphs have not only been used for deriving measures, but also for analyzing certain metrics commonly used in the assessment of certain attributes of program quality. When used to represent the skeleton of a component-based software, directed CFGs with cubic graph properties, namely CCFGs, enable the composition and decomposition of the structural system skeleton (Seker and others 2004).This section starts by introducing CCFGs and then provides some graph theoretical notions and formalisms. An example is provided to show the transition of a program's CFG to a CCFG, and then its decomposition into the CIUs.

2.3.1.1 Cubic Control Flowgraphs

A specific class of CFGs, CCFGs are of special interest. A structured program can be represented by a CCFG, then the CCFG can be reduced to its components, namely the CCFGs of the three constructs for structured programming. Prather used CCFG properties in designing and analyzing software metrics (Prather 1995). Moreover, by using the sequencing and nesting properties of CCFGs, Prather identified a collection of independent metrics and constructed a combination metric (Prather 1996).

2.3.1.2 Structured Programming Constructs by CCFGs

Following Prather (1995), Figure 2.1 shows the three basic constructs in flowchart form and their equivalent CCFGs and cubic graphs. The main focus of this chapter, related to flowgraphs, is the CCFGs. In CCFGs, the decision nodes are colored black and have indegree one and outdegree two, while the junction nodes are white and they have

indegree two and outdegree one. Tang used CCFGs to investigate McCabe's Cyclomatic Number (Tang and others 2001) and showed that the Cyclomatic Complexity is the number of decision nodes plus one. Tang also developed a framework for the composition and decomposition of CBS systems using the CCFG representation of a system skeleton (Tang 1999).

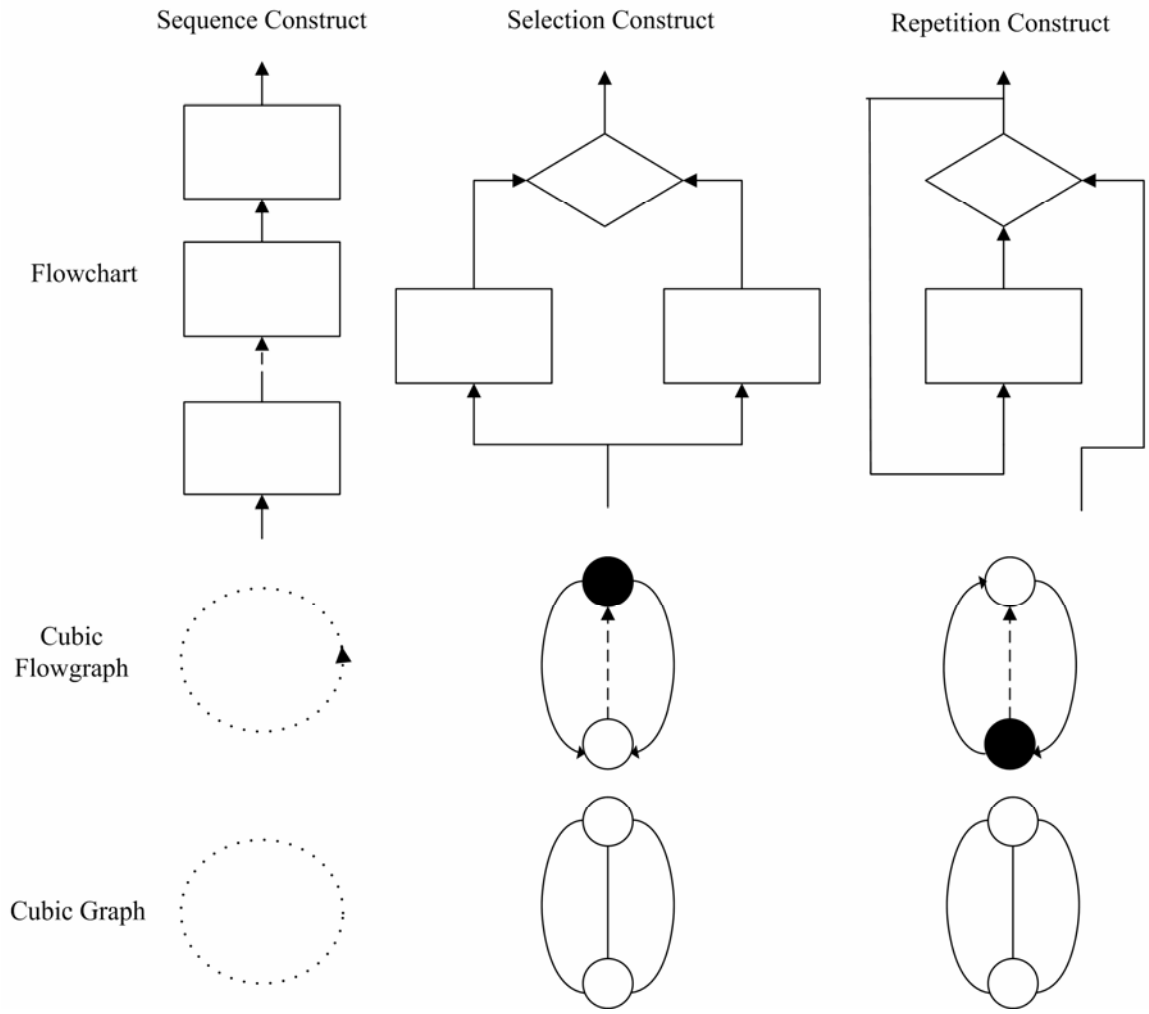


Figure 2.1 The three constructs of structured programming and their equivalent CCFGs

2.3.1.3 Composition and Decomposition Principles of CCFGs

Following Prather (1996), the steps for the decomposition and composition of CCFGs are stated as:

Decomposition of CCFGs

- Identify two edges to be removed to get a disconnected graph.
- Add one edge to each of the resulting graphs to retain their cubic graph property.

Composition of CCFGs

- Identify one edge from each of the cubic graphs to be composed and remove the edges. The removed edges must be from opposite directions, and at least one of them must be the dashed edge. When one edge is dashed, the resulting flowgraph is sequencing of the two. When both edges are dashed, the result is nesting of the two CCFGs. The flow direction to and from the corresponding nodes must be kept consistent with the originals.
- Bridge the graphs with two edges such that the edges extend from one graph to the other, retaining the cubic graph property of the resulting graph.

The decomposition process can be continued until no further decomposition is possible. In this case, the undecomposable components are called irreducible, or prime CCFGs. A cubic graph must be doubly connected (e.g., two edges must be removed to disconnect the graph) so that it can be decomposed into two components. Singly connected cubic graphs have no corresponding program CFG due to the violation of strong connectedness, and therefore they are not of interest for this study. Triply connected cubic flowgraphs are irreducible (Seker 2002).

2.3.2 Decomposition of Structured Programs into CIUs

In order to represent the construction of the CCFG of a program and decompose it, an example is provided. Although the focus of this study is not to replicate existing knowledge, the detailed decomposition process is used later on in the decomposition/representation phase of the entropy framework. Therefore, the decomposition steps are demonstrated. Two definitions are needed before the example is explained (Seker 2002).

Definition 2.8: Every CCFG is a Component Integration Unit (CIU).

Definition 2.9: Every irreducible CCFG is a CIU.

2.3.2.1 A Complete Decomposition of a Program into its Constructs

The following example is used to present the decomposition of a CBS based on its CCFG. The decomposition process is continued until the irreducible (prime) CIUs are obtained. The system used in this example is a structured program and does not include any statement that violates any of the structured programming constructs.

A program and its CFG are seen side by side in Figure 2.2. In this program, the capital letters $(A, B, ...)$ may be used to represent either program segments or components. Moreover, they can be CIUs provided as components from a component provider. *P1*, *P2*,..., *P5* represent the conditional statements on which decisions are made. The next step is to convert the CFG of Figure 2.2 to a CCFG. This conversion is done by collapsing the process nodes that correspond to components and introducing junction nodes where necessary to render the resulting control flowgraph cubic. Figure 2.2 shows the CCFG of the program and its control flowgraph.

The resulting CCFG shown in Figure 2.3 has its decision nodes colored black. The junction nodes are white, while all of the process nodes have been collapsed. The process nodes that correspond to components on this system skeleton are hidden along the arcs of this CCFG.

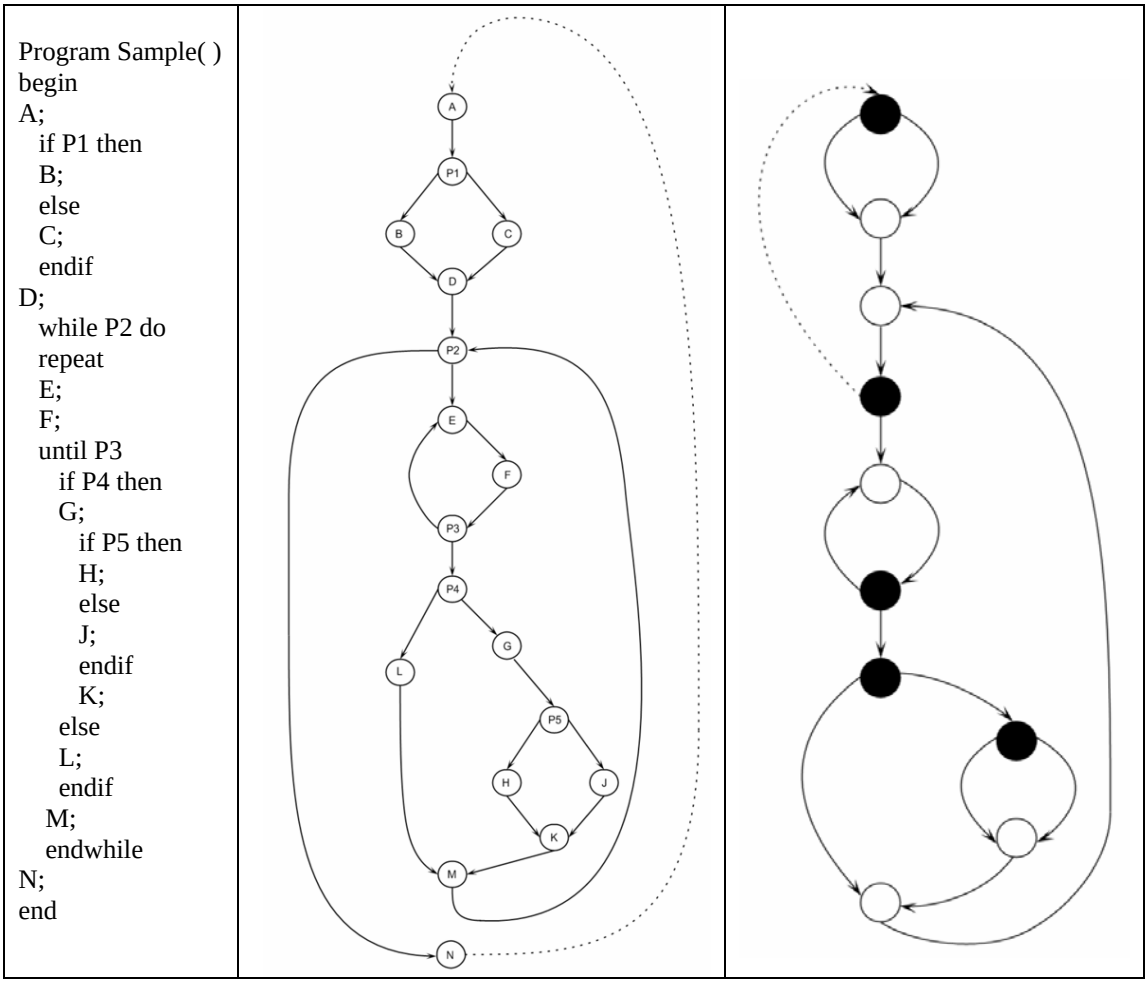


Figure 2.2 A program, its control flowgraph and cubic control flowgraph

Figure 2.3 shows some of the components of the CCFG shown in Figure 2.2. The first step of decomposition produces CIU-1 and CIU-2, enclosed in dotted rectangles. CIU-2 can be decomposed further. Some of the components of CIU-2 are marked and

labeled in the figure, but one of them is not seen as easily as the others. After the decomposition process is finalized, all of the resulting CIUs are prime components, hence no further decomposition is possible. Then the components that are hard to see at first sight become visible.

First level decomposition

In the first level of decomposition, the system level CCFG is decomposed into two main components, CIU-1 and CIU-2, which are seen in Figure 2.3. To explain further steps in the decomposition process, the CIUs that make up CIU-2 are enclosed in dotted rectangles and numbered accordingly, as shown in Figure 2.4, in which the first step of the decomposition process is depicted.

Second level decomposition

In the second level of decomposition, both of the CIUs that resulted from the first level decomposition need to be processed. For this specific example, CIU-1 is prime (not decomposable). Moreover, CIU-1 corresponds to the selective construct, which is one of the basic constructs of structured programming languages. Therefore, the decomposition process will focus on decomposing CIU-2 into its two main components that are seen in Figure 2.5. One can see the decomposition is not yet finalized, since there are still doubly connected components in CIU-2-1 and the remainder of CIU-2.

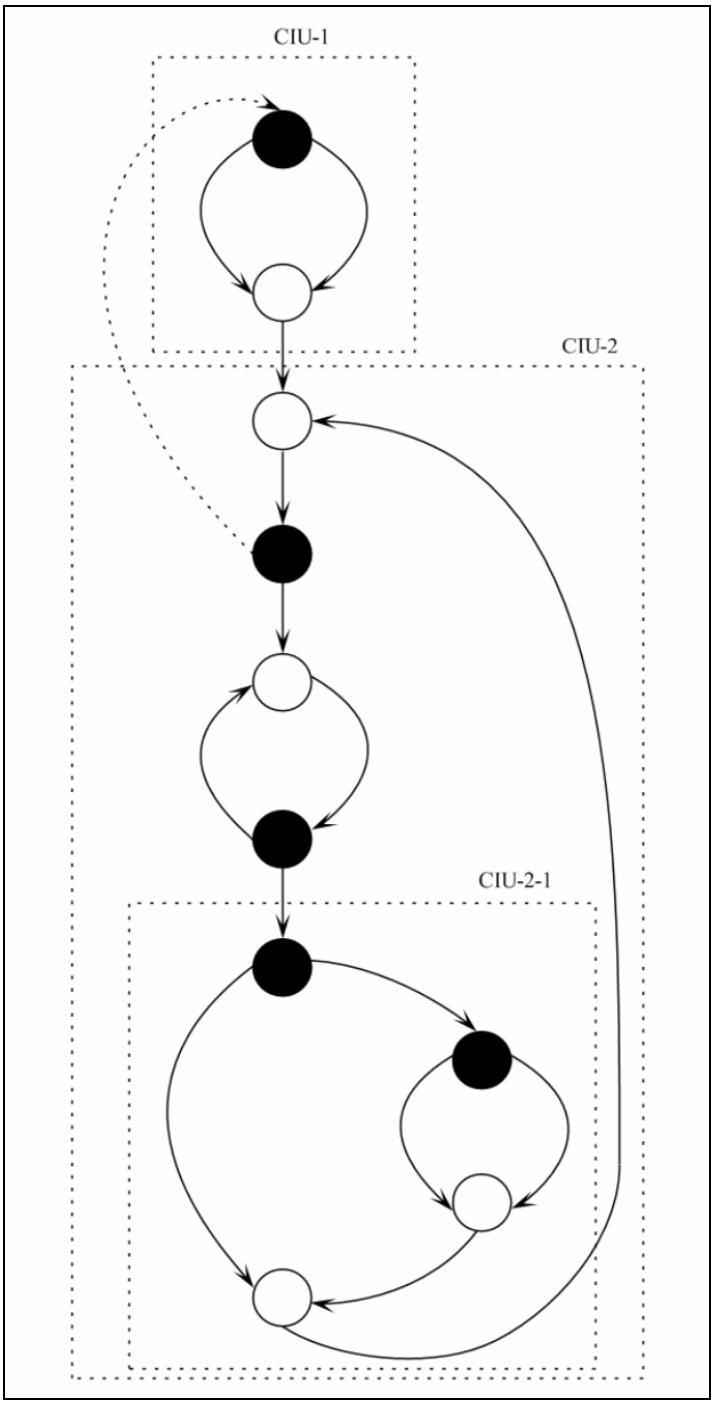


Figure 2.3 The two components obtained from the CCFG shown in Figure 2.2

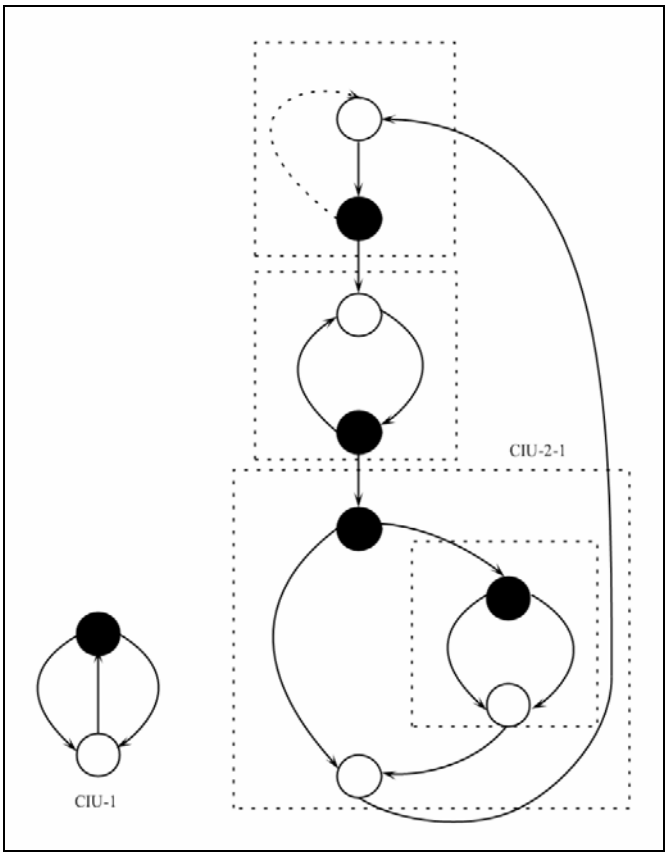


Figure 2.4 First level decomposition of the CCFG shown in Figure 2.2

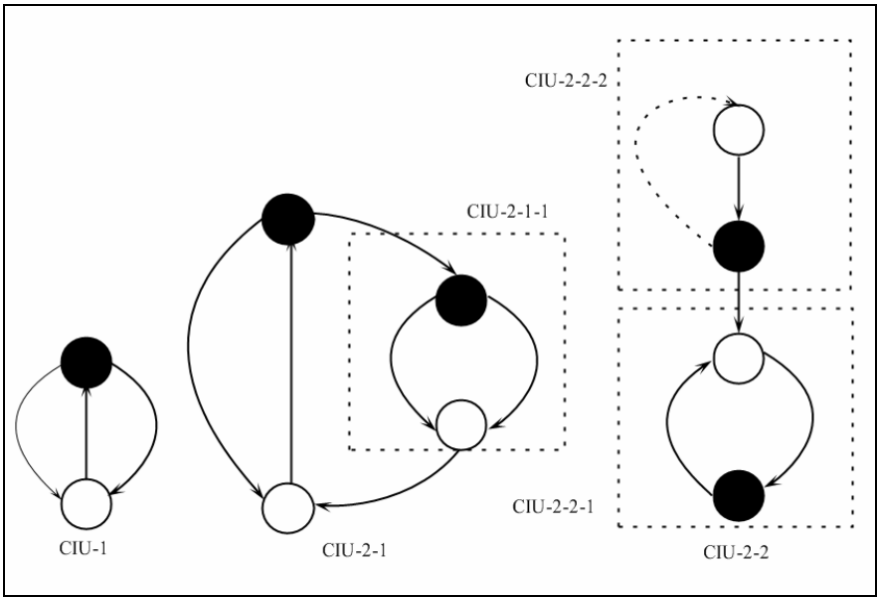


Figure 2.5 Second level decomposition of the CCFG shown in Figure 2.2

The prime CIUs

The final products of the decomposition process are shown in Figure 2.6. All of the CCFGs are triply connected and hence are irreducible. These results can be verified by examining the prime CIUs to determine that they do not contain any CIU that can be extracted. As expected, all of the prime CIUs obtained at the end of the decomposition process are of the three basic structured programming constructs.

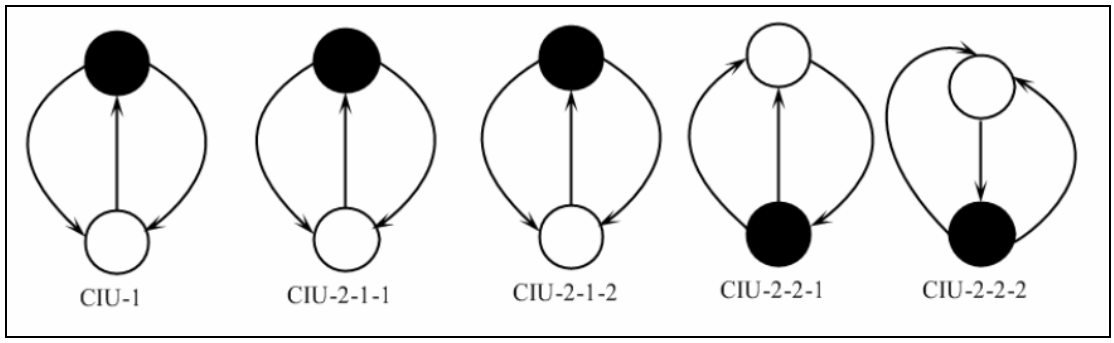


Figure 2.6 All of the prime components of the CCFG shown in Figure 2.2

Summary

This chapter summarized the theoretical foundation of this dissertation, which consists of software measurement theory, information theory, and decomposition theory. The measurement theory section presented the fundamental definitions used throughout the dissertation, including measure and metric. The information theory section explained entropy and the measurement of entropy. The decomposition theory section explained the method of decomposition of flowgraphs. These sections are referenced throughout the dissertation to explain the decomposition of hierarchical software systems and the application of entropy-based metrics.

CHAPTER 3

HIERARCHY, COMPLEXITY, AND COMPONENT-BASED SOFTWARE

It is a commonplace observation that nature loves hierarchies. Most of the complex systems that occur in nature find their place in one or more of four intertwined hierarchic sequences (Simon 1969).

Herbert Simon

In the last chapter, complex systems were defined as having hierarchies and it was shown that they could be decomposed into sub-systems for an analysis of their behavior. According to Morowitz (1995), the complex systems share certain features, such as having a large number of elements, possessing high dimensionality, and representing an extended range of possibilities. Such systems are hierarchies consisting of different levels with each having its own principles, laws, and structures. Twenty years ago, Brooks noted the essential complexities that make computer software the most complex entity among human-made artifacts (Brooks 1987). Decomposition, abstraction, and separation of concerns are general methods for tackling the complexity problem. Identifying the hierarchies within a complex software system is often not easy, because it requires the discovery of patterns among many objects, each of which may embody tremendously complicated behavior. Once these hierarchies are exposed, the structure of a complex system and an understanding of it become simplified (Booch and others 2007).

In this section, the hierarchy of systems, complexity issues in software, types of software complexities, and an effective way of coping with complexity, component-based development, are discussed. This section proves that representing software as a hierarchi-

cal system is necessary to apply measurement methods. Hence, the decomposition and abstraction of a software system is considered an essential part of the measurement framework that is described in the related chapter of this dissertation.

3.1 Hierarchy of Systems

Hierarchy theory is a subset of general systems theory that emerged as part of an effort to deal with the complexity problem. Rooted in the work of economist Herbert Simon, chemist Ilya Prigogine, and psychologist Jean Piaget, hierarchy theory focuses upon levels of organization and issues of scale. There is significant emphasis placed upon the observer in the system (Allen and Starr 1982).

In mathematical terms, a hierarchy is a collection of parts with ordered asymmetric relationships inside a whole; meaning that upper levels are above lower levels, and that the relationships upward are asymmetric with the relationships downward. Hierarchical levels are populated by entities whose properties characterize the level in question.

It is possible to describe two levels in a hierarchical structure, level of organization and level of observation. Level of organization fits into its hierarchy by virtue of a set of definitions that lock the level in question to those above and below. For example, a biological population level is an aggregate of entities from the organism level of organization, but it is only so by definition. There is no particular scale involved in the population level of organization, in that some organisms are larger than some populations, as in the case of skin parasites. Level of observation fits into its hierarchy by virtue of relative scaling considerations (Allen and Starr 1982).

It is also possible to distinguish complexity from complicatedness with the hierarchy theory. A hierarchical structure with a large number of lowest level entities, but with simple organization, offers a low, flat hierarchy that is complicated rather than complex. The behavior of structurally intricate systems is elaborate and complicated, whereas the behavior of deeply hierarchical complex systems is simple.

We follow the approach to hierarchy that Herb Simon described in his book, the Sciences of the Artificial (Simon 1969). Simon argues that complexity takes the form of hierarchy and that hierarchical systems evolve faster than non-hierarchical ones. Generally, a hierarchy is a recursive partition of a system into subsystems. Examples of hierarchies are common in social, biological, physical, and symbolic (e.g., books) systems. In biological systems, it is argued that hierarchical systems evolve faster because the many sub-systems form as intermediate stable stages in the process. Similarly, in problem solving, mainly a selective trial-and-error process, intermediate results constitute stable subassemblies that indicate progress. Simon also states that hierarchies have the property of near decomposability; namely that the short-term (high-frequency) behavior of each subsystem is approximately independent of the other components, and in the long run, the long-term (low-frequency) behavior of a sub-system depends on that of other components only in an aggregate way. In conclusion, a general theory of complex systems must refer to a theory of hierarchy, and the near decomposability property simplifies both the behavior of a complex system and its description.

3.2 Complexity of Software

Complexity has been discussed in the software domain for the last thirty years. One early work is the doctoral thesis of Van Emden, “An Analysis of Complexity (1971).” The work of Van Emden was based on the concept of conditional probabilities on the formalism of information theory, and appeared suitable to model the complexity of interconnected systems, such as programs built of modules. The reasons for creating or inventing software measures are based on the knowledge that program structure and modularity are important considerations for the development of reliable software. Most software specialists agree that higher reliability is achieved when software systems are highly modularized and module structure is kept simple (Schneidewind 1977). Modularity was discussed earlier, by Parnas (1975), who suggested that modules should be structured so that a module has no information pertaining to the internal structure of other modules.

Software complexity measurement is an area of software engineering that has been studied for over 30 years. Its objective is measuring factors that affect the cost of developing and maintaining software. Over the years there have been more than a hundred different metrics to measure software complexity (Henderson-Sellers 1996; Fenton and Pfleeger 1998). However, there is no established common definition of software complexity. Zuse (1990) states that the term "software complexity" is poorly defined and that software complexity measurement is a misnomer. He gives this definition of software complexity: “The true meaning of software complexity is the difficulty to maintain, change, and understand software. It deals with the psychological complexity of programs.” According to Basili (1980), software complexity is a measure of the resources

expended by a system (human or other) while interacting with a piece of software. If the interacting system is a programmer, then complexity is defined by the difficulty of performing tasks, such as coding, debugging, testing, or modifying the software. Even though this definition does not clarify the complexity of the problem or the difficulty of finding the solution, it provides a good understanding of problem complexity.

The software design process can be defined as an operational model obtained from a set of transformations applied to the initial model of that problem (Lehman and Belady 1985). Therefore, behind each solution there is a problem. However, the complexity of the problem differs from the complexity of the solution. For example, complex solutions may exist for simple problems. The complexity of a problem is the amount of resources required for an optimal solution. The complexity of a solution is the amount of resources required for that solution (Cardoso and others 2001). Henderson-Sellers (1996) proposes another definition, in which software complexity is a concept which refers to the characteristics that cause the person who performs a task on the software to experience difficulty. For him, the cognitive complexity of software refers to those characteristics of software that affect the level of resources used by a person performing a given task.

Despite the fact that there are numerous works on software complexity, systematic approaches in its classification and use are lacking. The situation in this field is confusing, and unsatisfying for the user (Zuse 1990). Researchers urgently need theoretical guidance on the measurement of software in general, and software complexity in particular. Baker and others (1990) suggested, “For research results to be meaningful, software measurement must be well grounded in theory.” Kearney and others (1986) had stated

previously, “Successful software complexity measure development must be motivated by a theory of programming behavior.”

The importance of software complexity lies in the fact that knowing the complexity of a specific software product or its module enables one to do the following:

- Predict the cost of testing and maintenance, the number of errors left in the software, and the size of the final program.
- Assess the time, effort, and cost of development.
- Identify critical modules or parts of the software.
- Compare programs, modules, and programmers according to software complexity (Podgorelec and Hericko 2007).

Complexity measures do not necessarily represent the software development or maintenance effort expended or the software cost. For example, software complexity in terms of the Cyclomatic Number (McCabe 1976) is related to the number of defects rather than to software cost or effort. Albrecht’s Function Point Analysis (1979) is widely used to measure the software size of Management Information Systems and is derived from software functionalities.

3.3 Types of Software Complexities

There have been different classifications of software complexities. According to Zuse, there are two main categories of software complexity: computational and psychological (Zuse 1990). Computational complexity refers to algorithm efficiency in terms of the time and memory needed to execute a program. Psychological complexity refers to the human effort needed to perform a software task, or, in other words, the difficulty ex-

perienced in understanding or performing such a task. In the literature, software complexity measures refer to psychological complexity. Three specific types of psychological complexity that affect a programmer's ability to comprehend software have been identified (Cardoso and others 2001; Conte and others 1986): problem complexity, system design complexity, and procedural complexity.

Problem complexity is a function of the problem domain. Simply stated, it is assumed that complex problems are more difficult for a programmer to comprehend than simple problems. Since this type of complexity is impossible to control, it is generally ignored in software engineering.

System design complexity addresses the mapping of a problem space into a given representation. Structural complexity and data complexity are the two types of system design complexity defined for structured systems (Cardoso and others 2001). Structural complexity addresses the concept of coupling, which measures the interdependence of modules of source code, i.e., C functions calling other C functions. It is assumed that the more coupling between modules, the more difficult it is for a programmer to comprehend a given module. Data complexity addresses the concept of cohesion. Cohesion measures the intradependence of a module. In this case, it is assumed that the more cohesive a module, the easier it is for a programmer to comprehend the module (Stevens and others 1976).

The complexity of a system is based on the sum of the structural and data complexity for all modules in the system (Cardoso and others 2001). These measures address information system complexity at the system and module levels. This multiple level approach with an emphasis on cohesion and coupling provides a basis in traditional soft-

ware measurement for the proposed software complexity model of OO systems. Procedural complexity is associated with the logical structure of a program. This approach to complexity measurement assumes that the length of the program (number of tokens or lines of code) (Halstead 1977) or the number of logical constructs (sequences, decisions, or loops) (McCabe 1976) that a program contains determines the complexity of the program (Cardoso and others 2001). A classification of software complexity is given in Figure 3.1.

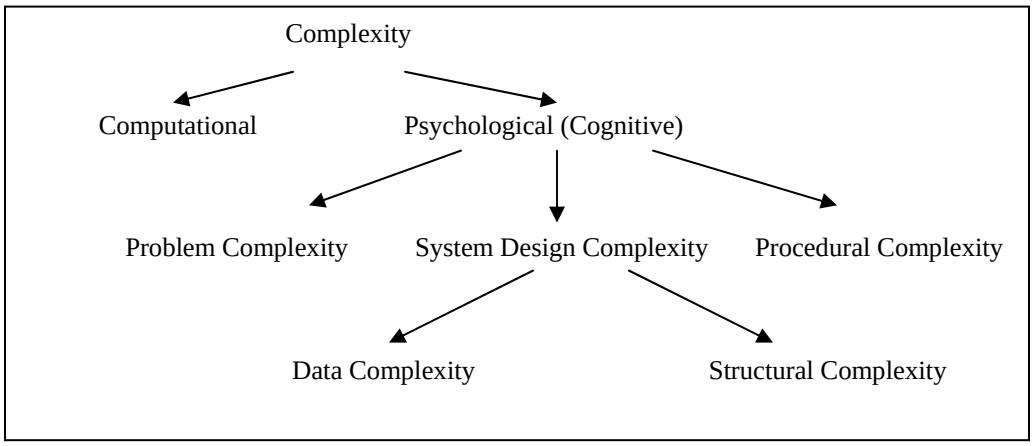


Figure 3.1 Classification of software complexity

In the next section, one of the recent software paradigms that tackles software complexity by building software from loosely coupled components is explained.

3.4 Component-Based Software Development

CBSE emerged in the late 1990s with the objective of increasing the reuse of existing components. The main objectives of CBD are to cope with the inherent complexity of large software systems and to enable the reuse of components. The CBD process includes defining, implementing, and integrating loosely coupled components into systems

(Sommerville 2007). The history of the decomposition of software into modules goes back to the early 1970s. Parnas (1972) explained the need for separating the software into modules. The benefits of the modularity he summarized are still valid current software practice. These benefits are:

- Shorter time-to-market (development time) because modules can be developed by separate groups.
- Increased product flexibility.
- Ease of change.
- Increased comprehensibility, as modules can be studied separately.

CBSE is a software development process that is based on modularity principles with a special emphasis on component reuse. CBD allows the developers to perform their activities in a repeatable way to build systems that have predictable properties. In addition to decomposition and modularity, integrating components into larger applications is a primary objective of CBD. As software systems grow larger and more complex, the only way to cope with complexity is to reuse existing components. CBSE addresses the complexity of software by decomposing software into components that conform to a specific component model. Using components allows application developers to package and organize code in a more formal, high level manner (Heinemann and others 2004).

There have been many definitions offered for software components (Szyperski 1998). The general consensus is that a component is an independent software unit that can be composed with other components to create a software system (Sommerville 2007). Another definition that includes two important characteristics of a component is the fol-

lowing: A component is an object with a well-defined run-time interface, and an independently developed body for composition with other components.

The implementation details of a component are hidden from the casual users and are known by those, who need to modify or test the component. The components communicate through well-defined interfaces, and one implementation of a component can be replaced by another without changing the system (Aktunc 2002). For integration issues, the interface must provide information about the component's methods and attributes. A good documentation of a component should include the syntax, and ideally, the semantics of all component interfaces (Sommerville 2007). Components must also conform to a standardized component model that may define the component interfaces, composition rules, and deployment. The component models are the basis for system middleware that provides support for executing components. The most important component models are CORBA, Enterprise Java Beans model, and .NET.

The CBSE process is composed of two sub-processes, domain engineering and component-based development. The activities of domain engineering include identifying, constructing, cataloging, and disseminating a set of components in a particular application domain. CBD activities include adapting and integrating components for software creation. Figure 3.2 identifies these sub-processes and their interactions. In this process model, the domain experts analyze the domain and create a domain model that acts as a basis for analyzing user requirements. In the meantime, software architects begin the design phase according to requirements analysis. After reusable components are located in the component libraries or created, they are used by software engineers during compo-

nent based development. The process continues with application testing and continuous updates.

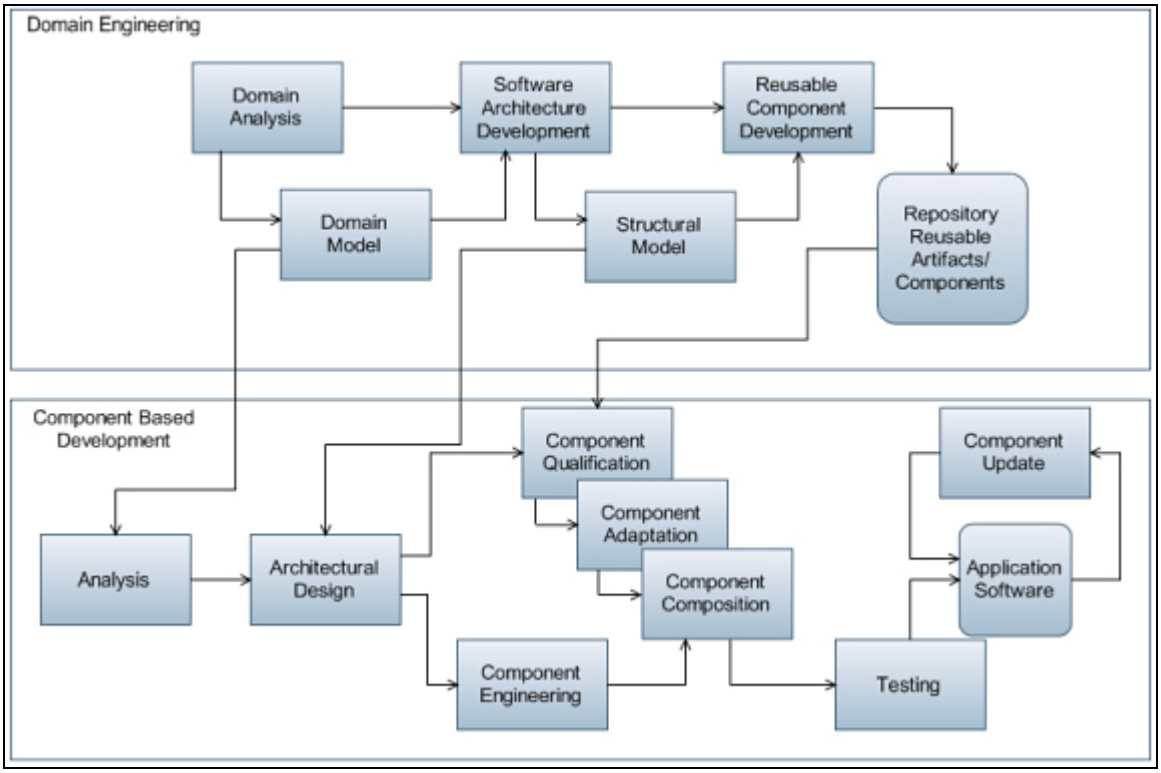


Figure 3.2 A process model that supports CBSE (adapted from Pressman 2007)

In the following chapter, CBSE is discussed in terms of measurement, following a summary of the current use of metrics in software engineering. CBSE is compared to traditional engineering disciplines. This comparison leads to the argument that measurement and use of metrics are necessary for CBSE to become an established engineering discipline.

Summary

In this chapter, Simon's outlook on the hierarchy of systems was discussed. Then, the complexity issues in software that have emerged since the early 1970s were summarized. Methods to tackle software complexity, such as modularity, decomposition, abstraction, and separation of concerns were explained briefly. In the following section, the taxonomy of software complexity was presented; mainly, computational and cognitive complexities. The last section summarized CBD, a relatively recent method to develop software using such methods as modularity and reuse. The process of CBSE was explained using its sub-processes: domain-based engineering and component-based development. The next chapter deals with software metrics and offers a discussion as to why software metrics are needed in order to achieve success in applying CBD principles.

CHAPTER 4

SOFTWARE METRICS

Software management has witnessed significant growth in the last twenty years. Effective management of any process requires quantification, measurement, and modeling (Mills 1988). Software metrics provide a quantitative basis for the development and validation of software development process models. As software development matures, effective metrics will enable organizations to monitor and improve their productivity. The transition from measurements to metrics is similar to the transition from observation to understanding (Pandian 2004). Metrics are created by the developer and designed to reveal a chosen characteristic in a reliable and meaningful manner. The metrics are then applied to software, depending on the property that needs to be addressed.

This chapter begins by addressing the need for software metrics, followed by a definition and discussion of the benefits of software metrics. Then, the attributes of software metrics are listed. The next section answers the questions of how and what to measure with software metrics. It is followed by a software metrics taxonomy that branches under three main classes: product, process, and resource metrics. The last section focuses on metrics that could be used for CBD.

4.1 The Need for Software Metrics

Software projects are notorious for running over schedule and budget, yet still having quality problems. IT projects run over budget by about 43 percent on average,

representing more than $17 billion in IT spending, according to the Standish Group, a research firm. In 2005, only 29 percent of IT projects finished on time and within budget. Also, 53 percent missed deadlines or budgets, while 18 percent failed outright (Wilson 2005). There are many reasons for the failure of these projects, such as weak project management, poor planning, communication shortfalls, excessive focus on technology at the expense of business processes, and organizational problems. A key solution to software management problems is the application of a metrics program, primarily a process which involves collecting, analyzing and using measurement data in order to have control over projects. Software measurement allows the developers to quantify schedule, work effort, product size, project status, and quality performance. Accurate schedule and cost estimation can be achieved through effective software management, which can be facilitated by the use of software metrics. Improvement of the management process depends upon an enhanced ability to identify, measure, and control essential parameters of the development process. The goal of software metrics is the identification and measurement of the essential parameters that affect software development (Mills 1988).

4.2 Definition of Software Metrics

Software metrics are an integral part of software engineering. Goodman (1993) defines software metrics as, “The continuous application of measurement based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products.”

Metrics strongly support software project management activities. They relate to the four functions of management (USC 2001):

- Planning. Metrics serve as a basis for cost estimating, planning training, resource planning, scheduling, and budgeting.
- Organizing. Size and schedule metrics influence a project's organization.
- Controlling. Metrics are used to track software development activities for compliance to plans.
- Improving. Metrics are used as a tool for process improvement, to identify where improvement efforts should be concentrated and to measure the effects of these efforts.

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 items, such as the source lines of code (LOC), documentation pages, staff hours, tests, and requirements. Examples of derived metrics include the source LOC per staff hour, defects per thousand LOC, and a cost performance index.

4.3 Benefits of Software Metrics

Software metrics provide quantitative and qualitative values to management. Metrics allow the success of the project to be assessed and provide a basis for calculating a return on investment (ROI). Metrics are also a valuable way to track the health of the software and to help identify issues quickly and systematically for effective solutions. Metrics initiate observation and analysis, which leads to a discovery of goals, capabilities,

and constraints. Quantitative expressions of the observations give additional clarity and simplicity. Metrics are catalysts for improvement initiatives and quality movements. By integrating knowledge and providing communication, resources are better utilized.

Using metrics data improves the personal thinking process. Aside from rational models, metrics lead to cognitive models that improve the vision of the developers. Problem solving cycles benefit from metrics. Metrics are used for recognition and diagnosis of problems, and they provide necessary data and methods for the managers and developers to find solutions (Pandian 2004). Metrics provide specific success criteria for projects, allowing the outcomes to be assessed at the end of implementation. Metrics provide a concrete way of assessing the success of various approaches which leads to greater understanding. This assessment can be considered when establishing new initiatives. Metrics can be of tangible benefit both at the early stages of a project and throughout its life.

4.4 Attributes of Software Metrics

There has been a major increase in the quantity of software metrics in the last twenty years. During the evaluation of software using metrics, it is critical that we evaluate the metrics themselves according to some criteria. Ejiogu (1991) defined a set of attributes that metrics should encompass:

- Simplicity. Metrics should be simple, easy to understand, and meaningful. They should be understood by many members of a project team, and should not be too complicated to apply.
- Empirically and intuitively persuasive. The metrics should satisfy the engineer's intuitive notions about the product attribute under consideration.

- Consistent and objective. Metrics should yield repeatable, consistent data independent of the people applying them. The data should not come from unrealistic assumptions or easily manipulated resources.
- Consistent in the use of units. The computation of the metrics should use consistent units that are mathematically sound.
- Language independent. Metrics should not be applied according to the programming language but should be based on the design model, or the structure of the program.
- High-quality feedback. Metrics should present valuable results that can be used by the software team to improve the product.

These attributes could be used as a general guideline to determine the usability of metrics; however, not all useful metrics satisfy all of these attributes. A common example is the popular “Function Point (FP)” metric. The subjective value used in this metric does not meet the “consistent and objective” standard, because each person may assign different values causing different results. However, FP still provides useful information and distinct value (Pressman 2007).

4.5 How and What to Measure

Choosing metrics for measurement raises fundamental questions: How to measure, which includes other questions about how detailed is the collected data, how often the measurement should be done, or how the software should be sized? The question of what to measure also raises additional questions regarding what is necessary and possible to measure. The main objective is to measure what is necessary; however, in most software

engineering areas, this aspect is unknown, especially for modern software development paradigms or methodologies (Dumke 1998).

Several architectures and frameworks have been proposed to apply software metrics in the last twenty years, such as SEI's software metrics framework, the Goal-Question-Metric framework, and even quality models (CMMI) emphasizing measurement (Berander and Jonsson 2006). Designing a metrics architecture is an initial step in starting a metrics program. Architecture needs to connect metrics to business goals and decision making in order to make metrics meaningful. Metric models provide the methods that allow metrics to be applied to the artifacts. There is more than one way to build metrics architecture; however, Figure 4.1 presents an architecture that represents the common method. The layers of the architecture include business goals, decision center, metric models, metrics, and measurement instruments.

Metrics cannot exist without business goals, and businesses must determine those goals before selecting metrics. It is not possible to track the success of any method or metrics without having a clear vision of the business goals. Answering questions, such as how to increase usability, customer satisfaction, or sales, assists in developing this vision. After defining the goals, the decision center that might consist of many organizational layers, such as a decision tree, would decide on the metrics model and specific metrics to be applied. Many of the metrics models are goal-centered models that have a strong, dynamic interaction between goals and metrics. Basili's GQM (1994) is the best known of these models. Metrics models are usually hybrid structures that are composed of cognitive, semantic, and quantitative methods.

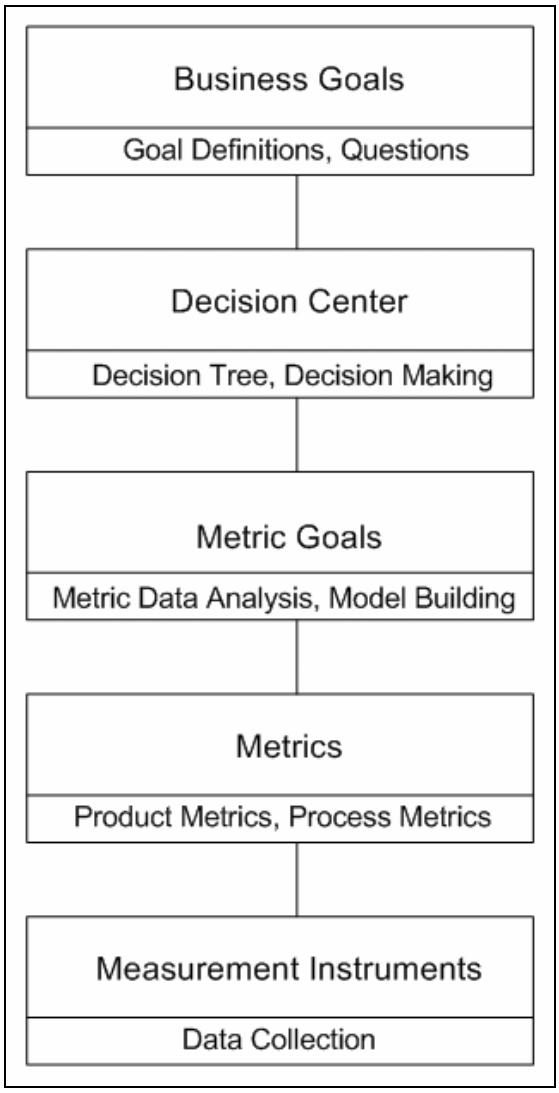


Figure 4.1 Metrics architecture

This section thoroughly explains the features of metrics and the methods for using them; however, the following points should be taken into consideration in a metric framework (Robertson 2003):

- Metrics should be specific and avoid generalized targets. They should specify target value, timeframe, who or what will be measured, and dependencies on other projects or systems.

- Metrics should determine a baseline. The application of metrics at the beginning of a project makes it possible to quantify the success of the project when the first metrics are assessed. This quantification enables rapid, tangible feedback while the issues are still relevant.
- Metrics should be automated. The design of the system should include the methods of data collection so that the metrics may become an automatically generated product of normal usage. This incorporation of metrics greatly decreases the burden of implementing and managing metrics.
- Metrics should focus on target entities. Metrics should be implemented to only what needs to be measured. If the wrong metrics are put in place, they will distract from the real issues and goals. At worst, they can entrench undesirable behavior or reduce productivity.
- The implemented metrics should also be dynamically evaluated to determine their continuing validity. It is often necessary to discontinue or modify some metrics, or to establish new ones.

These are general guidelines about using metrics; however, it is necessary to explain what should be measured as well. There are numerous entities that may change from project to project, though the following are the most common (Poulin 2001):

1. Schedule. Schedule progress needs to be tracked continuously. Estimated progress can be updated according to the results of the measurement. Earned value reporting, such as the value of work completed compared to the budgeted cost of work scheduled, is one of the methods used by project managers.

2. Source instructions per component. Code counting tools can list the total SLOC (Source Lines of Code) per component.
3. Labor hours. Recording the number of hours people work on projects or components.
4. Classification of the component. Classifying components into categories, such as new code, changed code, built for reuse, and reused code. This data is helpful in calculating the amount of reuse in a component as well as the costs to write reusable component, write new components, and to reuse components.
5. Cost. Linking time to cost which generates data, such as cost per component and the cost to develop a line of code.
6. Change requests. Recording the number of change requests indicates the volatility of the component design, as well as the level of work in the pipeline.
7. Defects. Maintaining the count of defects discovered throughout the life cycle. This process indicates the quality of components, and allows one to analyze where in the life cycle there are more defects and how much it will cost to fix them.

4.6 Software Metrics Classification

It is possible to make two different classifications in software measurement: One is to classify what we are measuring, and the other is to classify what we are measuring with. The entities we measure are processes, products, and resources. The entities considered in software measurement are (Fenton 1991):

- Process. Any activity related to software development.
- Product. Any artifact produced during software development.

- Resource. People, hardware, or software needed for the processes.

To measure these entities, we need to identify their attributes, which can be internal and external. Grady and Caswell (1987) define these attributes as primitive (internal) or computed (external). Internal attributes of an entity can be measured based only on the entity, therefore the measures are direct. Program size (LOC), number of defects observed in unit testing, or total development time for the project are all internal attributes. External attributes of an entity can be measured only with respect to how the entity relates with the environment and can only be measured indirectly. For example, reliability, an external attribute of a program, does not depend only on the program itself but also on the compiler, machine, and user. Productivity, an external attribute of a person, clearly depends on many factors, such as the kind of process and the quality of the software produced (Riguzzi 1996). External metrics are a combination of other metric values and are often more useful in understanding or evaluating the software process than the internal metrics.

Table 4.1, adapted from Riguzzi (1996), shows examples of entities and attributes for the different phases of software development. Measurement can take place in all phases of software development: requirements analysis, specification, design, coding, and verification. Undoubtedly, measurement is most useful when carried out in the early phases. Although these are the phases in which it is more difficult to apply measurement. In fact, most measures have been designed for the coding phase.

Table 4.1 Entities and Attributes (Based on Riguzzi 1996)

Entity	Internal Attributes	External Attributes
Product		
requirements, specification, design, code, test set	size, reuse, modularity, redundancy, functionality, coupling, cohesion, control flow complexity, coverage level	understandability, stability, maintainability, comprehensibility, quality, reliability, usability, reusability
Process		
requirements analysis, specification, design, coding, testing	time, effort, number of requirements changes, number of specifications changes, number of design changes, number of code changes	cost effectiveness
Resources		
personnel, team, software, hardware	age, cost, size, communication level, structure, price, speed, memory	productivity, experience, usability, reliability

The basic components of software development, i.e., process, product, and resource characteristics, define individual application domains for software measurement and allow different measurement strategies (Dumke 1998). At the same time, these characteristics serve as a starting point for detailed classification of software metrics. Several references present different classifications of software metrics, but most of them agree on the following classification (Fenton and Pfleeger 1998):

- Product metrics. These metrics measure the software product at any stage of its development. They are often classified according to size, complexity, quality, and data dependency.
- Process metrics. These metrics measure the process in regard to the time the project will take, cost, methodology followed, and how the experience of the team

members can affect these values. They can be classified as empirical, statistical, theory-based, and composite models.

- Resource metrics. These metrics measure available resource characteristics. They are sometimes referred to as project metrics.

4.6.1 Product Metrics

Early works on product metrics were mostly about characteristics of source code, such as size. An important number of product metrics also deal with software complexity. They measure several software entities, such as specifications documents, design diagrams, and source code listings. The following classes of metrics are under the product metrics classification tree (Pressman 2007):

- Metrics for the analysis model address various aspects of the analysis model.
  - Function metrics measure the functionality of the system.
  - Specification quality metrics measure the specificity and completeness of the requirements specification.
- Metrics for the design model quantify the design quality.
  - Architectural metrics measure the quality of the architecture.
  - Component-level metrics measure the complexity and quality of software components.
  - Interface design metrics measure the usability of the system interface.
  - Specialized object-oriented design metrics measure characteristics of classes, and their communication and collaboration characteristics.
- Metrics for source code measure various aspects of the source code.

- Halstead metrics predict the size of programs based on the number of operators and operands.
- Complexity metrics measure the logical complexity of source code.
- Length metrics measure the size of the source code.

- Metrics for testing assist in the design of test cases and evaluate testing methods.
  - Statement and branch coverage metrics lead to the design of test cases that provide program coverage.
  - Defect-related metrics measure the defects in the program.
  - Testing effectiveness metrics measure the effectiveness of tests in real time.
  - In-process metrics are process-related metrics that are determined as testing is conducted.

The following section briefly explains the metrics listed above.

4.6.1.1 Metrics for the Analysis Model

4.6.1.1.1 Function Points. The Function Point is a metric that was developed by Albrecht (1979). It focuses on measuring the functionality of the software product according to the following parameters: user inputs, user outputs, user inquiries, number of files, and the number of external interfaces.

Once the parameters are counted, a complexity (simple, average, or complex) value is associated to each parameter. A complexity adjustment value is added to the previous count. This value is obtained from the response to fourteen questions related to reliability of the software product. The following equation is used to calculate Function Points

$$FP = count\ total \times (0.65 + 0.01 \times \sum F_i) \quad (4.1)$$

where

FP = total number of adjusted function points, *count total* = the sum of all user inputs, outputs, inquiries, files, and external interfaces to which the weighting factor has been applied, and F_i = a complexity adjustment value from the response to the fourteen reliability questions.

FP is a widely used metric type due to its usability at the early stages of software development, its technology independence, the understandability of it from the perspectives of both developers and users, and its different types, such as Feature Points. The FP metric is difficult to use because one must identify all parameters of the software product. This identification is subjective, and various organizations could interpret the parameters differently. Moreover, interpretations could be different from one project to another in the same organization, and could also be different from one software release to another.

4.6.1.1.2 Bang metrics. The Bang metric was described by DeMarco (1982) as a measure of size, based on the functionality of the system. It can be calculated from certain algorithms and data primitives available from a set of formal specifications for the software. The diagrams generated during the design show the functional entities. The design diagrams to consider for the Bang metric are data dictionary, entity relationship, data flow, and state transition. Other design diagrams including the business model, use cases, and sequence and collaboration diagrams of the Unified Modeling Language (UML), can also be used to find the functional entities. The Bang metric can also be combined with other measures to estimate the project cost.

4.6.1.1.3 Specification Quality metrics; Davis's metrics. Davis (1993) proposed a set of characteristics to evaluate the quality of requirements specification. It includes specificity, completeness, correctness, understandability, verifiability, internal and external consistency, achievability, concision, traceability, modifiability, precision, and reusability. Each characteristic can be represented using one or more metrics. For example: There are n_r requirements in a specification, such that

$$n_r = n_f + n_{nf} \tag{4.2}$$

where

n_f is the number of functional requirements and n_{nf} is the number of nonfunctional requirements.

Davis suggests the Q_1 metric that determines the specificity of functional requirements based on the consistency of the reviewers' interpretation of each requirement

$$Q_1 = \frac{n_{ui}}{n_r} \tag{4.3}$$

where

n_{ui} is the number of requirements for which all reviewers had identical interpretations. The closer the value of Q_1 to 1, the lower the ambiguity of the specification (Pressman 2007).

Coulter's metrics. Coulter and others (1987) proposed the following metrics to determine specification complexity. The complexity of specification assumes there is a finite set of memory locations *M*, a finite set of location values *Q*, and the problem space

PS, the set of state descriptions defined by the program specification. The specification complexity is then determined to be

$$SC = H(PS) \tag{4.4}$$

$$SC = \log_2 \|PS\| \tag{4.5}$$

where

H is entropy. Due to the assumption that a priori probabilities of state descriptions are equal, the entropy, and hence the *SC*, reduces to the logarithm of cardinality of the problem space.

4.6.1.2 Metrics for the Design Model

4.6.1.2.1 Architectural metrics. These metrics focus on characteristics of the program architecture with an emphasis on the effectiveness of components within an architecture. These operate as black-box metrics in that they do not require any knowledge of a component's inner structure.

Card and Glass (1990) define three design complexity metrics: structural complexity, data complexity, and system complexity. System complexity is defined as the sum of structural and data complexity

$$C(i) = S(i) + D(i). \tag{4.6}$$

For hierarchical architectures, structural complexity of a module, $S(i)$, is defined as

$$S(i) = f_{out}(i)^2 \tag{4.7}$$

where

$f_{out}(i)$ = fan-out of module *i*.

Fan-out is defined as the number of modules immediately subordinate to the module *i*, that is, the number of modules that are directly invoked by module *i.* Fan-in is defined as the number of modules that directly invoke module *i*.

Data complexity is an indication of the complexity in the internal interface of a module and is defined as

$$D(i) = \frac{v(i)}{[f_{out}(i) + 1]}. \tag{4.8}$$

The relative system complexity measurement is defined as

$$RSC = \frac{C(i)}{n} \tag{4.9}$$

where

n = number of modules.

There are other architectural design metrics including one proposed by Fenton (1991), and the Design Structure Quality Index (DSQI), developed by U.S. Air Force Systems Command (USAF 1987).

4.6.1.2.2 Object-oriented design metrics. Object-Oriented Metrics (OOM) are used to determine the quality of the design and bring insight to other issues, such as size and complexity. Major studies of OOM have been done by Chidamber and Kemerer (1991), Whitmire (1997), Harrison and others (1998), and Lorenz and Kidd (1994). The most referenced research is Chidamber and Kemerer's Metrics Suite. The authors made the most thorough research regarding OOM investigation. They proposed six class-based design metrics for OO systems:

- The Weighted Methods per Class (WMC) Metric is defined as the sum of the complexities of all methods of a class.
- The Depth of Inheritance Tree (DIT) Metric is defined as the maximum length from the node to the root of the tree.
- The Number of Children (NOC) Metric is defined as the number of immediate subclasses.
- The Coupling Between Object (CBO) Classes Metric is defined as the count of the classes to which this class is coupled. Two classes are coupled when methods declared in one class use methods or instance variables of the other class.
- The Response For a Class (RFC) Metric is defined as the number of methods in the set of all methods that can be invoked in response to a message sent to an object of a class.
- The Lack of Cohesion in Methods (LCOM) Metric is defined as the number of different methods within a class that reference a given instance variable.

Component-level design metrics are covered in section 4.7, metrics for component-based systems.

4.6.1.3 Metrics for Source Code

Source code metrics measure the properties of the source code, such as size, complexity, and maintainability.

4.6.1.3.1 Size metrics. SLOC and Halstead metrics are the most known size metrics. The oldest software metrics, LOC, have the drawback of being difficult to measure and estimate prior to completion of coding.

LOC. LOC is the oldest and most widely used software metric. It is a quick method that can be performed using automated tools. Ease of use is one reason why counting LOC has been widespread among software development companies, despite the limitations of the measure.

Currently, the most accepted definition of LOC is given by Conte and others (1986), "A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements."

The LOC metric specifies the number of lines in the code. The comments and blank lines are ignored during this measurement. The LOC metric is often represented by thousands of LOC (KLOC) or source LOC (SLOC). LOC is often used during the testing and maintenance phases, not only to specify the size of the software product but also in conjunction with other metrics to analyze other aspects of its quality and cost.

One of the main disadvantages of LOC is that it does not take into account the quality of the code; if we use LOC to measure productivity, a short, well-designed program is punished by such a metric. Another disadvantage is that it does not allow comparisons of programs written in different languages. One cannot compare software products using LOC unless they are coded in the same language. When developing software

in a homogenous environment that is not a problem, but today almost every software project uses a variety of languages (Nysted and Sandros 1999).

Other serious deficiencies associated with LOC (Nysted and Sandros 1999):

- The lack of a national or international standard for a line of code metric that encompasses all procedural languages.
- Software can be produced using methods in which such entities as LOC are irrelevant, i.e., program generators, spreadsheets, graphic icons, reusable modules of unknown size, and inheritance.
- LOC metrics paradoxically give decreased values as the level of the language gets higher, so the most powerful, advanced languages appear to be less productive than the more primitive, low-level languages. This paradox is due to the definition of productivity (*productivity=size/effort*), which generates a lower productivity when the size of the code decreases.

In spite of these problems, LOC is a widely used metric due to its simplicity, ease of application, and absence of alternative size measures. Moreover, there are many empirical studies that demonstrate the usefulness of LOC. One study found that effort correlated better with LOC than with Halstead's metrics and was at least as good as McCabe's Cyclomatic Complexity (Basili 1980). Basili's study and other studies (Evangelist 1983) demonstrate that LOC is better or at least as good as any other metric for measuring some attributes of software. LOC have been used for a variety of tasks in software development including planning, monitoring the progress of projects, predicting (e.g., the effort estimating model, COCOMO, developed by Boehm in 1981), and they have been used as a

baseline for evaluation in almost all empirical studies on metrics, including function points (Albrecht 1983).

4.6.1.3.2 Halstead metrics. Maurice Halstead's measures of complexity (Halstead 1977) were proposed shortly after McCabe presented his cyclomatic number in 1975. Halstead stated that any software program could be measured by counting the number of operators and operands, and from them, he defined a series of formulas to calculate the vocabulary, length, and volume of the software program:

- Program Vocabulary. Halstead visualized programs as sequences of tokens, with each token being classified as an operator or an operand. The vocabulary *n* of the program is

$$n = n_1 + n_2 \tag{4.10}$$

where

n_1 is the number of unique operators in the program, and n_2 is the unique number of operands. *n* is the total number of unique tokens used to construct the program.

- Program Length. Program length *N* is the total number of tokens in the program

$$N = N_1 + N_2 \tag{4.11}$$

where

N_1 is the total number of operators, and N_2 is the total number of operands. *N* is a measure of the size of the program and is derivable from the program; however, the distinction between operators and operands may be non-trivial.

- Program Volume. Another program size measure is the program volume, defined by Halstead as

$$V = N \times \log_2 n . \tag{4.12}$$

Since N is a number, the unit of V can be interpreted as bits, so V is a measure of the storage volume required to represent the program. Empirical studies show that values of LOC, N, and V appear to be linearly related and equally valid as relative measures of program size (Christensen and others 1981; Elshoff 1976; Li and Cheung 1987).

One of the problems associated with Halstead's model is how to define operands and operators. We already touched upon this issue when we discussed the dependence upon programming language. Halstead asserted that all commands in a program can be separated into operators and operands, but the problem is that even if we have succeeded in categorizing the components of one language, we have to do it all over again when we begin to use another language. Moreover, the definition of operators and operands may differ between users since there is no standard, which makes comparisons between software developers difficult (Ohlsson 1996).

The psychological assumptions that Halstead makes in his measurement of effort and programming time have also been criticized. Opponents have claimed that the postulation that the human mind is capable of eighteen mental discriminations per second is too exact and does not have enough empirical evidence to be accepted as scientific (Ohlsson 1996).

In spite of all these problems, Halstead metrics have gained wide attention because they constitute the first attempt to define a metric for software.

4.6.1.3.3 Complexity metrics. Complexity metrics are used to determine the complexity of program flow. Many of the approaches are based on representing programs in flowgraphs. There are numerous complexity metrics in the literature. Zuse's surveys (1990, 1997) listed eighteen categories of complexity metrics for software. The study that has been referenced the most has been McCabe's Cyclomatic Complexity (1976). The most frequently used complexity metrics are explained in the next section.

4.6.1.3.4 Cyclomatic Complexity. The software metric known as Cyclomatic Complexity was proposed by McCabe (1976), who suggested representing the program as a graph and then determining the number of different paths through the graph. A program control flow can be represented by a graph that has unique entry and exit nodes, and in which all nodes are reachable from the entry and the exit is reachable from all nodes. McCabe's idea was to measure the complexity by considering the number of paths in the control graph of the program. Even for simple programs containing at least one cycle, the number of paths is infinite. He therefore considers only the number of independent paths, that are complete paths (paths that go from the starting node to the end node of the graph), such that their linear combinations can produce the complete path of a program, as seen in Figure 4.2.

In the graph in Figure 4.2, there are four possible paths: aceg, acfh, bdeg, and bdfh. We can represent each path as a vector with eight positions, one for each arc, containing the number of times that the arc is present in the path. In this case we would see that, of the four vectors, only three are independent, while the fourth is always a linear combination of the other three. Therefore, the number of independent paths is three. In graph the-

ory, it can be demonstrated that in a strongly connected graph (one in which each node can be reached from any other node) the number of independent paths is given by *e-n+1*, where *e* is the number of nodes and *n* is the number of arcs.

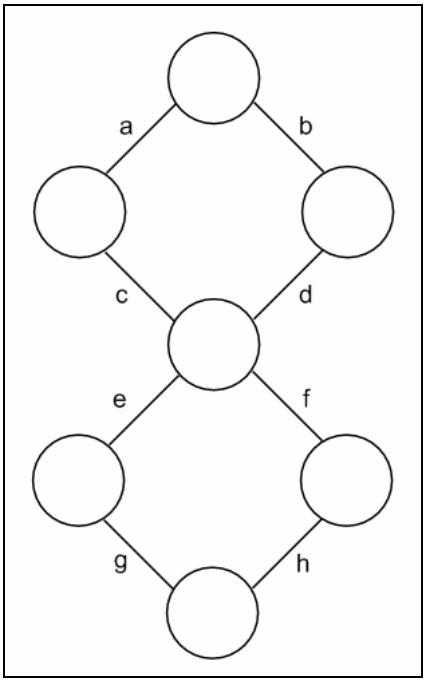


Figure 4.2 Example graph for Cyclomatic Complexity

We cannot apply this formula directly to the control flowgraph of a program, because it may not be strongly connected. We can make it strongly connected by adding an arc from the end node to the start node of the program. In this way, we increase by one the number of arcs, and therefore the number of independent paths (as a function of the original graph) is given by

$$V(G) = e - n + 2 \tag{4.13}$$

and this is the cyclomatic number as defined by McCabe (1976). This formula has an interesting simplification; we can calculate the cyclomatic number only by knowing the number of choice points *d* in the program (Mills Theorem)

$$V(G) = d + 1. \tag{4.14}$$

The Cyclomatic Complexity metric is an accurate study of control flow complexity. McCabe partially validated his metric by showing that there is a degree of correlation between the cyclomatic number and some quantities that surely influence control flow complexity, such as reliability (McCabe 1976). From this experimental work, McCabe derived the empirical rule that the cyclomatic number of a module should not be bigger than ten.

However, when we analyzed the metric from the point of view of measurement theory, we have seen the demonstration by Fenton (1991) that it is not possible to define a single metric for control flow complexity and, therefore, the cyclomatic number as a measure of that attribute is not valid. Even if a single metric for control flow complexity (and of complexity in general) does not exist, we can still consistently measure single aspects of complexity, such as the number of independent paths, for which the cyclomatic number is a valid measure.

Other objections that have been raised against McCabe's measure are that it is insensitive to, among other things, the frequency and types of input and output activity, the size of purely sequential programs, the number of variables in the program, and the intensity of fatal operations in the program, i.e., things that are captured by other measures (Ohlsson 1996).

4.6.1.3.5 Information Flow. Information Flow metric was first proposed by Kafura and Henry to measure the complexity of a software module (Kafura and Henry 1987). Their method counts the number of local information flows entering (fan-in) and exiting (fan-out) each procedure. The procedure's complexity is then defined as

$$C = procedure\ length \times (fan\_in \times fan\_out)^2 . \quad (4.15)$$

The merit of this measure is that it addresses a higher level of complexity than does McCabe's metrics. To get an overall picture of complexity one need also study the interactions between modules, not just the transactions within the modules. Modules with either a low fan-in or low fan-out are, in a sense, isolated from the system and hence have low complexity. This phenomenon is recognized by Kafura and Henry's measure, and they justify the multiplication of fan-in and fan-out by referring to this observation.

However, from a measurement theory perspective, researchers have questioned the basis for this action. In particular, they are concerned about the lack of complexity for many modules, as the multiplication leads to a value of zero complexity for any module in which either the fan-in or fan-out is zero. Many experts have also questioned the length factor, partly because it is a separate attribute and partly because it contradicts measurement theory. Consequently, Martin Shepperd studied Kafura and Henry's measure in-depth and identified a number of theoretical problems (Shepperd 1988). He proposed to refine the measure by excluding the length factor.

4.6.1.4 Quality Metrics

Early examples of work on quality metrics were discussed by Boehm, McCall, and others (Boehm and others 1976; McCall and others 1977). The user community has

since sought a single model for depicting and expressing quality. The advantage of a universal model is clear; it makes the comparison easier between one product and another. In 1992, IEEE proposed standard 1061, called “Standard for a Software Quality Metrics Methodology,” which builds on and further develops the ISO 9126 model (“Software Product Evaluation: Quality Characteristics and Guidelines for their Use”) (ISO 1991). In the IEEE standard, software quality is defined as “the degree to which software possesses a desired combination of attributes (e.g., reliability, interoperability)” (IEEE 1992). Quality is then decomposed into six factors (some of which are defined differently in our model of software complexity):

- Functionality implies the existence of certain properties and functions that satisfy stated or implied needs of users.
- Reliability is the capability of software to maintain its level of performance under stated conditions for a stated period of time.
- Efficiency is the relationship of the level of performance to the amount of resources used under stated conditions.
- Usability is the evaluation of results and the individual assessment of use by users.
- Maintainability is the effort needed for specific modifications.
- Portability is the ability of software to be transferred from one environment to another.

Software quality can be measured at any phase of development. The quality metrics are given in the following sections.

4.6.1.4.1 Defect metrics. Defect metrics help to follow-up on the number of defects found in order to have a record of the number of design changes, the number of errors detected by code inspections, the number of errors detected during integration testing, the number of code changes required, and the number of enhancements suggested for the system. Defect metrics' records from past projects can be saved as history for further reference. Those records can be used later in the development process or in new projects. Companies usually keep track of the defects on database systems specially designed for this purpose.

4.6.1.4.2 Reliability metrics. These metrics are used to determine the probability of software failure or the rate at which software errors occur. This type of estimation can be done if data collected on software defects is examined as a function of time.

4.6.1.4.3 Maintainability metrics. Many efforts have been made to use metrics to measure or predict the maintainability of software (Yau and Collofello 1980). For example, Halstead metrics were used to predict the psychological complexity of software maintenance tasks. Assuming such predictions can be made accurately, complexity metrics can be used to reduce the cost of software maintenance (Harrison and others 1982).

4.6.2 Process Metrics

These metrics measure process in terms of time a project will take, the cost, the methodology followed, and how the experience of the team members can affect these values. Before proceeding to classify process metrics, we need to define related key con-

cepts, such as process improvement and process management. In the context of improving organizational maturity, process management is defined as establishing a process infrastructure to uniformly support and guide a project's work. Process improvement, on the other hand, specifically refers to improvements in process capability and performance. A process management system facilitates process improvements by providing process metrics (and empirical data) to assess process capability and performance.

4.6.2.1 Empirical Models

One of the earliest models used to project the cost of large-scale software projects was described by Wolverton (1974). The method compares a proposed project to similar projects for which historical cost data are available. It is assumed that the cost of the total life cycle effort in the new project can be estimated using this historical data.

4.6.2.2 Statistical Models

Walston and Felix (1977) from IBM used data from sixty software projects to develop a simple model of software development. The LOC metric was assumed to be the principal determinant of the development effort. The relationship that they defined was

$$E = aL^b \tag{4.17}$$

where

L is the number of LOC, and E is effort. Regression analysis was used to define a and b. Walston and Felix also developed a productivity index based upon evaluations of twenty-nine project variables (Walston and Felix 1977).

4.6.2.3 Composite Models

A number of other process models have used some combination of intuition, statistical analysis, and expert judgment. These have been labeled as composite models by Conte and others (1986). The most used is the COCOMO model. This model was developed by Boehm (1981), and is the widely used cost-estimation model. It provides three levels of models: basic, intermediate, and detailed. Boehm defines three modes of product development that aid in determining the difficulty of the project: organic, semidetached, and embedded. The developmental effort equations are of the form

$$E = aS^b m \tag{4.18}$$

where

a and *b* are constants determined for each mode and model level, *S* is the value of source LOC, and *m* is a composite multiplier determined from fifteen cost-driver attributes. Boehm suggests that the detailed model will provide cost estimates that are within 20 percent of actual values 70 percent of the time (Boehm 1981). Other composite models worth mentioning are Softcost by Tausworthe (1981), the SPQR model by Jones (1986), COPMO by Thebaut (1984), and ESTIMACS by Rubin (1987).

4.6.2.4 Reliability Models

A number of dynamic models of software defects have been developed. These models describe the occurrence of defects as a function of time, allowing one to define the reliability *R* and Mean Time To Failure (MTTF). One example is the model described by Musa (1975), which, like most others of this type, makes six basic assumptions:

- Test inputs are random samples from the input environment.

- All software failures are observed.
- Failure intervals are independent of each other.
- Times between failures are exponentially distributed.
- Failure rate is proportional to the number of faults remaining.
- Development execution time rate of change of the number of faults corrected is proportional to the failure rate.

Based upon these assumptions, the following relationship can be derived

$$d(t) = D\ (1 - e^{bct}) \tag{4.19}$$

where

D is the total number of defects; *b*, *c* are constants that must be determined from historical data for similar software; *d*(*t*) is the number (cumulative total) of defects discovered at time *t*. As in many other software models, the determination of *b*, *c*, and *D* is a non-trivial yet vitally important task for the success of the model (Ruston 1979; Musa 1975).

4.6.3 Resource Metrics

Resource metrics measure the resource characteristics. The developers' outputs, the hardware reliability and performance, and the time elapsed for the project time are examples of common resource metrics. Although there are many resource metrics that businesses use, there is not an established taxonomy of resource metrics in the literature. Human resources metrics, hardware metrics, and network metrics are typically used in order to predict cost, effort, and efficiency.

Table 4.2 Metrics for Component-Based Software

Metric	Measures
Productivity	The amount of product delivered per unit time. A basic productivity metric is the ratio of total source lines of code to the total development hours for the project.
Cost	Total software development expenditure, including costs of component acquisition, integration, and quality improvement.
Product Stability	Level of changes to established software requirements. A basic stability metric is the ratio of change requests to the total number of requirements.
Reuse	The ratio of the product newly created by the developer to the components created and maintained by other parties.
Time to Market	Elapsed time between development start and component acquisition to software delivery.
System Resource Utilization	Use of target computer resources as a percentage of total capacity.
Performance	Response time and throughput in transaction bound systems.
Adaptability	Integrated system's ability to adapt to requirement changes.
Reliability	Probability of failure-free system operation over a specified period of time.
Customer Satisfaction	Degree to which the software meets customer expectations and requirements.

4.7 Metrics for Component-Based Systems

As CBD is maturing, the need for measurement and metrics in this area is greater than ever. Various metrics, including complexity, cost, and quality, are used frequently in CBSE. The difference in the methods for applying metrics arises from components' black-box nature, which masks their internal structure. CBS metrics do not depend on

source code size, which is generally not known. Measuring the number of use cases is an alternative measure to size measurement in CB systems. Component development time is also strongly affected by high levels of reuse when compared to traditional software projects. Component-based metrics have been researched by Poulin (2001) and Sedigh-Ali and others (2001). Table 4.2 on the preceding page lists metrics that could be used for CBS.

Summary

In this chapter, the need for software metrics was explained by means of schedule and cost. A definition of software metrics was given based on Goodman's definition (1993). The benefits of metrics were explained from the perspective of organizations and individual developers. Answers were given regarding the question of how the metrics should be applied and what the metrics should measure. The next section is a thorough summary of the classification of software metrics. Software metrics have been examined in three classes: product, process, and resource metrics. The last section of this chapter includes a list of metrics that can be applied to component-based systems.

CHAPTER 5

ENTROPIC MEASUREMENT FRAMEWORK

This chapter introduces the entropic measurement framework that is the main focus of this dissertation. The framework is based on two concepts, information theory and flowgraphs. The framework is utilized to quantify several aspects of software, such as complexity, coupling, and cohesion.

The chapter begins with a review of efforts to use entropy in software measurement. This section summarizes the novelties and problems faced in examining software through measurement of information content and entropy. It continues with a section about flowgraph representations and how to represent a flowgraph using a connectivity matrix. This section is describes the transition from software to flowgraphs and from flowgraphs to matrix expressions.

The following section explains the measurement of the information content of software in reference to Halstead's (1977) software metrics which are also based on information theory. The chapter continues with the concepts to be measured, cohesion and coupling. It summarizes how the information-theoretical approach may help quantify some of the fundamental software engineering concepts. Chapter six includes a detailed case study in which the methods explained in this chapter are applied and thoroughly analyzed in order to complete the framework.

5.1 Entropy Measurement in Software

Entropy has been defined in terms of the information content of software and used to measure software code complexity (Harrison 1992; Torres and Samadzadeh 1991; Davis and LeBlanc 1988). Harrison utilized entropy to measure the information content of C based procedural programs. Davis and LeBlanc used entropy to measure syntactic complexity in FORTRAN and COBOL programs, and they also used entropy to predict certain effects attributed to the psychological complexity of programs. Torres and Samadzadeh (1991) explored the relationship between entropy and software reusability and showed that entropy provides a good measure of program control complexity and reuse. They also showed an inverse relationship between the information content of software subsystems and their reusability. One of the earliest works in evaluating software code using information content or entropy showed that operator usage in programs follows a natural probability distribution, and thus could be used to measure a product's information content (Zweben and Halstead 1979). Operators, in all of the works mentioned, were expressed as special symbols (mathematical operators), reserved words, or function calls. The works by Berlinger (1980), Davis and LeBlanc (1988), and Harrison (1992) used this empirical distribution of operators to compute code entropy in procedural programs (written in FORTRAN, C, COBOL, Ada, and Assembly).

In procedural programming, publications of Berlinger (1980) and Davis and LeBlanc (1988) address the total information content of the program. Berlinger implies that a higher level of information content yields a more complex program. Similarly, Davis suggests that as information content increases, error density increases. On the other hand, Harrison's measure (1992) addresses the average information content per symbol.

He based the measure on an empirical distribution of operators within a program, where operators are special symbols, reserved words, or function calls. The results demonstrate that as the average information content increases, error density decreases, implying less complex programs. Harrison (1992) attributed the contradiction between his results and those of Berlinger (1980) and Davis (1988) to the fact that he used an average information content per symbol rather than the total information content of the program. While average information content metrics are not affected by program size, total information content metrics are highly affected by program size (Abd-El-Hafiz 2001).

5.2 Representation of a Flowgraph Using a Connectivity Matrix

5.2.1 Introduction

A flowgraph can be represented using a matrix $A = [a_{ij}]$ of order *n*. Each row or column in the matrix is represented by a node and is labeled by one of the integers from 1 to *n*, such that the node labeled *k* is associated with the *k*th row (column) of *A*. If $a_{ij} \neq 0$, then there is an edge (i, j) directed from nodes *i* to *j* with associated weight a_{ij}. For a more compact description of a flowgraph, the notation of 3-tuple $G(V, E, f)$ is used where V is a set of nodes, *E* is a set of edges, and *f* is a mapping function from *E* to the complex field, such that $f((i, j)) = a_{ji}$ for all *i* and *j* in *V*. It is also convenient to extend the mapping function *f* from a single edge (i, j) in *E* to any subgraph *R* of $G(V, E, f)$, such that $f(R) = \prod f((t, k))$ where the product is taken over all possible edges (t, k) in *R*.

For any *i* in *V*, the symbols $\rho(i)$ and $\rho^*(i)$ are used to denote the numbers of the sets of edges of $G(V, E, f)$ having *i* as initial and terminal nodes, respectively. A subgraph *S* of $G(V, E, f)$ is a regular graph of degree *k* if $\rho(i) = \rho^*(i) = k$ for each node *i* in *S*. If *A* is a subset of *V*, the sectional graph, denoted by $G[A]$, is a subgraph whose node set is *A*. The edges of the sectional graph are all *E* that connect two nodes in *A*. When $A = V$, the sectional graph is $G(V, E, f)$ itself. Two subgraphs are disjoint when they have no nodes in common. If *S* is a subgraph of $G(V, E, f)$ and each node of $G(V, E, f)$ is a node of *S*, then *S* is a spanning subgraph of $G(V, E, f)$. A component of $G(V, E, f)$ is a maximal connected subgraph of $G(V, E, f)$. A connection of $G(V, E, f)$ is a spanning subgraph of $G(V, E, f)$, which is a regular of degree 1. A subgraph of $G(V, E, f)$, denoted by H_{ij}, is defined as a one-connection from nodes *i* to *j* if it contains:

1. a directed path from nodes *i* to *j* or
2. a set of disjoint, directed circuits that include all nodes in *V* except for those contained in condition 1.

5.2.2 Adjacency Matrix

The adjacency matrix $A = \{a_{ij}\}$ of a labeled graph *G* with *p* points is the $p \times p$ matrix in which $a_{ij} = 1$ if v_i is adjacent to v_j and $a_{ij} = 0$ otherwise. Thus, there is a one-to-one correlation between labeled graphs with *p* points and $p \times p$ symmetric binary matrices with zero diagonals (Harary 1972).

The following example is constructed to demonstrate how to find the adjacency matrix of a graph. The graph in Figure 5.1 has 9 nodes and 11 edges.

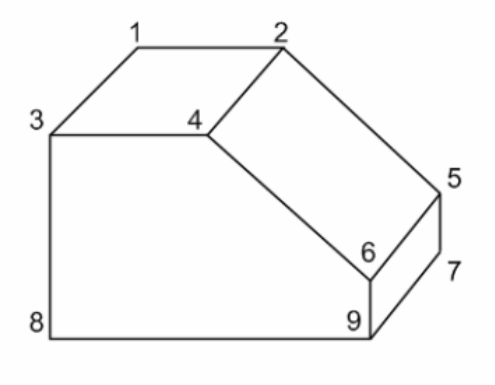


Figure 5.1 Graph *G*

Figure 5.2 illustrates the adjacency matrix of the graph *G* in Figure 5.1.

	1	2	3	4	5	6	7	8	9
1	0	1	1	0	0	0	0	0	0
2	1	0	0	1	1	0	0	0	0
3	1	0	0	1	0	0	0	1	0
4	0	1	1	0	0	1	0	0	0
5	0	1	0	0	0	1	1	0	0
6	0	0	0	1	1	0	0	0	1
7	0	0	0	0	1	0	0	0	1
8	0	0	1	0	0	0	0	0	1
9	0	0	0	0	0	1	1	1	1

Figure 5.2 The adjacency matrix of graph *G*

The number of 1’s in an adjacency matrix is equal to the number of edges in a graph that it represents.

Definition 5.1 (Reachability Matrix): In the reachability matrix *R*, $r_{ij} = 1$ if v_j is reachable from v_i, and 0 otherwise.

Assuming the following Boolean operation is carried out on adjacency matrix *A*: identity matrix *I* is added to matrix *A*, and Boolean addition and multiplication applied until a power *k* is reached, such that

$$(A+I)^{k-1} \neq (A+I)^{k} = (A+I)^{k+1} = R. \tag{5.1}$$

This matrix, denoted by *R*, to which the successive powers of $(A+I)$ converge, is called a reachability matrix. When $R_{ij} = 1$, unit *j* can be reached from unit *i*, either directly or indirectly, through one or more units. The adjacency matrix is also called a one-hop matrix, especially in studies of arbitrary networks. The adjacency matrix can be manipulated in order to obtain the connectivity matrix $C = \{c_{ij}\}$, for which the entry at (i, j) lists the minimum number of hops needed to connect node *i* to node *j*. Obtaining a connectivity matrix using an adjacency matrix is explained in section 5.2.3.

Definition 5.2 (Connectivity Matrix): A connectivity matrix is a characterization of a graph that offers information about the relationships between all vertices in the graph. If graph *G* is a graph with the vertex set $(v_1, v_2, ..., v_n)$, then the connectivity matrix *C* is the $n \times n$ matrix, $C(G) = (c_{ij})$, where $c_{ij} = d(v_i, v_j)$ (c_{ij} indicates the length of the shortest path from vertex *i* to vertex *j*). Since the graph *G* is connected, every vertex is reachable from every other vertex; therefore, all elements except for the diagonal elements in the connectivity matrix, is non-zero.

The connectivity matrix of the graph *G* in Figure 5.1 is illustrated in Figure 5.3.

	1	2	3	4	5	6	7	8	9
1	0	1	1	2	2	3	3	2	3
2	1	0	2	1	1	2	2	3	3
3	1	2	0	1	3	2	3	1	2
4	2	1	1	0	2	1	3	2	2
5	2	1	3	2	0	1	1	3	2
6	3	2	2	1	1	0	2	2	1
7	3	2	3	3	1	2	0	2	1
8	2	3	1	2	3	2	2	0	1
9	3	3	2	2	2	1	1	1	1

Figure 5.3 The connectivity matrix of graph *G*

Definition 5.3 (Maximum distance of a graph): The maximum distance of a graph is the longest distance between any vertices on the graph. In terms of the connectivity matrix, the maximum distance *k* is equal to the largest element c_{ij} in the matrix

$$k = \max(i, j) \tag{5.2}$$

where

$1 \le i, j \le n$.

Referring to Figure 5.3, the maximum distance of graph *G* is equal to the largest element in the connectivity matrix, which is 3.

5.2.3 Obtaining a Connectivity Matrix Using an Adjacency Matrix

The adjacency matrix of an *N*-node network is defined as $A = \{a_{ij}\}$, where

$$a_{ij} = \begin{cases} 1, \ link \ i \to j \text{ exists} \\ 0, \ link \ i \to j \text{ does not exist.} \end{cases} \tag{5.3}$$

The connectivity matrix of an *N*-node network is defined as $C^{(m)} = \left\{ c_{ij}{}^{(m)} \right\}$, where $C_{ii}{}^{(m)} = 0$, and

$$c_{ij} = \begin{cases} h, i \text{ connected to } j \text{ by } h \le m \text{ hops} \\ 0, \text{ otherwise.} \end{cases} \tag{5.4}$$

According to this definition, the adjacency matrix for the *N*-node network is the first iteration in calculating the connectivity matrix. That is, $C^{(1)} = A$. Considering the elements of the square of matrix *A*, which we denote $A^2 = \left\{ A_{ij}{}^{(2)} \right\}$, these elements can be written as

$$a_{ij}{}^{(2)} = \sum_{k=1}^{N} a_{ik} a_{kj} = \begin{cases} 0, \text{ no path } i \to k \to j \text{ exists} \\ > 0, \text{ at least one path } i \to k \to j \text{ exists.} \end{cases} \tag{5.5}$$

Note that every node having at least one neighbor would have a two-hop circular path back to itself. There may also be more than one two-hop path from *i* to *j*, and it is possible for a two-hop path to exist between nodes whose shortest connection is one-hop. So, to obtain the second iteration in calculating the connectivity matrix, we first modify $a_{ij}{}^{(2)}$ as

$$b_{ij}{}^{(2)} = \begin{cases} 0, \ i = j \text{ (eliminate looping paths)} \\ 0, \ a_{ij} = 1 \text{ (eliminate two-hop path when there is already a one-hop path)} \\ 2, \ a_{ij} > 0, \text{ when } i \ne j \\ 0, \text{ otherwise} \end{cases} \tag{5.6}$$

and then $c_{ij}{}^{(2)}$ is

$$c_{ij}{}^{(2)} = a_{ij}{}^{2} + b_{ij}{}^{(2)} = c_{ij}{}^{(1)} + b_{ij}{}^{(2)}. \tag{5.7}$$

The following example is used to demonstrate how to obtain the adjacency and connectivity matrices for the mesh network shown in Figure 5.4.

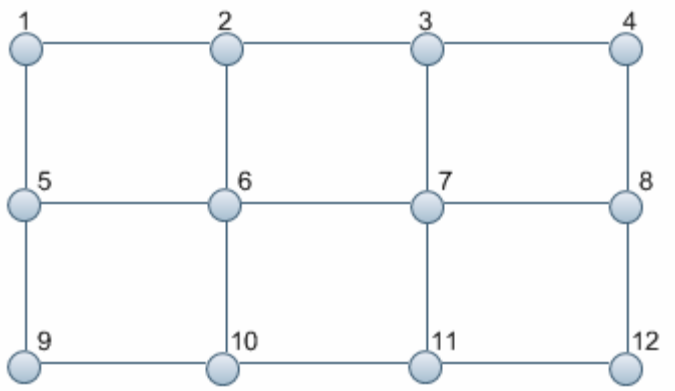


Figure 5.4 A mesh network

The adjacency matrix for this 3×4 mesh network is

$$B^{(2)} = \begin{bmatrix} 0 & 0 & 2 & 0 & 0 & 2 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 2 & 0 & 2 & 0 & 0 & 2 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 2 & 0 & 2 & 0 & 0 & 2 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 2 \\ 0 & 2 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 2 & 0 & 0 \\ 2 & 0 & 2 & 0 & 0 & 0 & 0 & 2 & 2 & 0 & 2 & 0 \\ 0 & 2 & 0 & 2 & 2 & 0 & 0 & 0 & 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 2 & 0 \\ 2 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 2 & 0 & 0 & 2 & 0 & 2 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 2 & 0 & 0 & 2 & 0 & 2 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 2 & 0 & 0 & 2 & 0 & 0 \end{bmatrix}$$

and the square of this matrix is

$$A^2 = \begin{bmatrix} 2 & 0 & 1 & 0 & 0 & 2 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 3 & 0 & 1 & 2 & 0 & 2 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 3 & 0 & 0 & 2 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 2 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 3 & 0 & 1 & 0 & 0 & 2 & 0 & 0 \\ 2 & 0 & 2 & 0 & 0 & 4 & 0 & 1 & 2 & 0 & 2 & 0 \\ 0 & 2 & 0 & 2 & 1 & 0 & 4 & 0 & 0 & 2 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 3 & 0 & 0 & 2 & 0 \\ 1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 2 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 2 & 0 & 2 & 0 & 0 & 3 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 2 & 0 & 2 & 1 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 2 \end{bmatrix}.$$

Applying (5.4) to A^2, we obtain the matrix for only two-hop connections

$$B^{(2)} = \begin{bmatrix} 0 & 0 & 2 & 0 & 0 & 2 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 2 & 0 & 2 & 0 & 0 & 2 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 2 & 0 & 2 & 0 & 0 & 2 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 2 \\ 0 & 2 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 2 & 0 & 0 \\ 2 & 0 & 2 & 0 & 0 & 0 & 0 & 2 & 2 & 0 & 2 & 0 \\ 0 & 2 & 0 & 2 & 2 & 0 & 0 & 0 & 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 2 & 0 \\ 2 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 2 & 0 & 0 & 2 & 0 & 2 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 2 & 0 & 0 & 2 & 0 & 2 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 2 & 0 & 0 & 2 & 0 & 0 \end{bmatrix}.$$

Adding $B^{(2)}$ to $C^{(1)}$ creates the intermediate calculation for the connectivity matrix for $m = 2$

$$C^{(2)} = C^{(1)} + B^{(2)} = \begin{bmatrix} 0 & 1 & 2 & 0 & 1 & 2 & 0 & 0 & 2 & 0 & 0 & 0 \\ 1 & 0 & 1 & 2 & 2 & 1 & 2 & 0 & 0 & 2 & 0 & 0 \\ 2 & 1 & 0 & 1 & 0 & 2 & 1 & 2 & 0 & 0 & 2 & 0 \\ 0 & 2 & 1 & 0 & 0 & 0 & 2 & 1 & 0 & 0 & 0 & 2 \\ 1 & 2 & 0 & 0 & 0 & 1 & 2 & 0 & 1 & 2 & 0 & 0 \\ 2 & 1 & 2 & 0 & 1 & 0 & 1 & 2 & 2 & 1 & 2 & 0 \\ 0 & 2 & 1 & 2 & 2 & 1 & 0 & 1 & 0 & 2 & 1 & 2 \\ 0 & 0 & 2 & 1 & 0 & 2 & 1 & 0 & 0 & 0 & 2 & 1 \\ 2 & 0 & 0 & 0 & 1 & 2 & 0 & 0 & 0 & 1 & 2 & 0 \\ 0 & 2 & 0 & 0 & 2 & 1 & 2 & 0 & 1 & 0 & 1 & 2 \\ 0 & 0 & 2 & 0 & 0 & 2 & 1 & 2 & 2 & 1 & 0 & 1 \\ 0 & 0 & 0 & 2 & 0 & 0 & 2 & 1 & 0 & 2 & 1 & 0 \end{bmatrix}.$$

The general rule for the intermediate calculations is

$$c_{ij}^{(m)} = c_{ij}^{(m-1)} + b_{ij}^{(m)},\ m \geq 2 \tag{5.8}$$

where

$$b_{ij}^{(m)} = \begin{cases} 0,\ i = j \\ 0,\ c_{ij} > 0 \\ m,\ \sum_{k=1}^{N} c_{ik}^{(m-1)} a_{kj} > 0 \ \text{ when } i \neq j \text{ and } c_{ij}^{(m-1)} = 0 \\ 0,\ \text{otherwise.} \end{cases} \tag{5.9}$$

5.2.4 Warshall's Algorithm to Compute a Connectivity Matrix from an Adjacency Matrix

Warshall's Algorithm (1962) applies to directed graphs and computes the transitive closure of a directed graph. It follows the logic that if there is a vertex from node *A* to node *B*, and a vertex from node *B* to node *C*, then there is a path from node *A* to node *C*. The algorithm can be used construct a connectivity matrix from an adjacency matrix. It runs in $O(n^3)$ time. It is an effective graph connectivity algorithm when the graph is dense, or it is a static structure with few updates and a several queries. The algorithm is explained as:

- Define two $N \times N$ matrices *B* and *C*, in addition to the $N \times N$ adjacency matrix *A*.
- Initially, set $C = A$ and set $B = 0$.
- For $m = 2$ to the longest hop distance (or until the update matrix *B* equals zero).
    - For all node pairs $(i, j) = (1,1)$ to (N, N).
        - If $i = j$, skip to the next node pair.
        - If $c_{ij} > 0$, skip to the next node pair.
            - For $k = 1$ to *N*.
                - If $c_{ik} > 0$ *and* $a_{kj} > 0$ for some *k*.
                - Set $b_{ij} = m$.
- Exit the loop and go to the next node pair (i, j).
- Set $C \leftarrow C + B$, then $B = 0$.

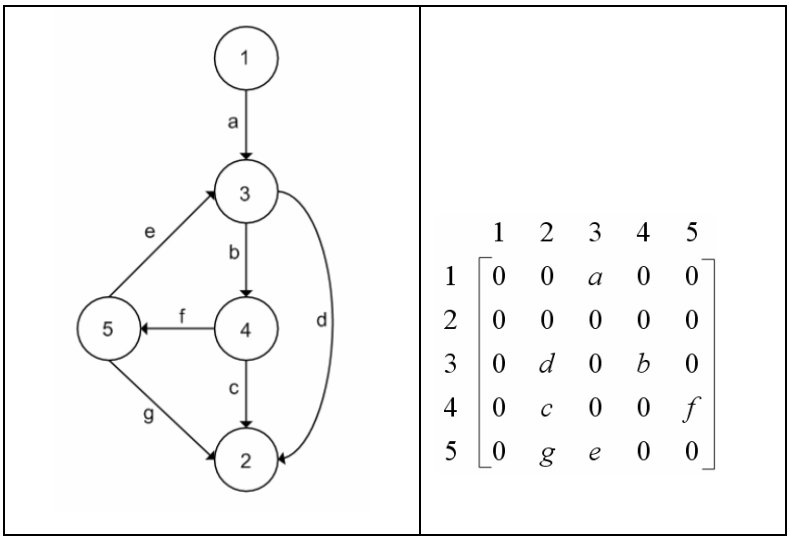
At the end of these calculations, *C* is the connectivity matrix. The sum of all the elements of *C*, divided by $N(N-1)$, equals the average hop distance.

5.2.5 Flowgraph Notation

A flowgraph is a notation used to represent the control flow of a system. The structured constructs of a program can be represented using flowgraph notations, such as sequence, if, while, until, and case. Each structured construct has a corresponding flow-graph symbol. Each circle, called a flowgraph node, represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows in the flowgraph, called edges or links, represent the flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the

node does not represent any procedural statements. Areas bound by edges and nodes are called regions. When counting regions, we include the area outside the graph as a region. In the C language the conditional constructs include if, while, and case. When compound conditions are encountered in a procedural design, the generation of a flowgraph becomes more complicated. A compound condition occurs when one or more Boolean operators are present in a conditional statement. A separate node is created for each of the conditions *a* and *b*, in the statement if *a* or *b*. Each node that contains a condition, is called a predicate node and is characterized by two or more edges emanating from it.

Flowgraphs can be represented with connectivity and adjacency matrices. A weighted adjacency matrix is a square matrix whose size (e.g., number of rows and columns) is equal to the number of nodes on the flowgraph. Each row and column correspond to an identified node, and matrix entries correspond to connections (edges) between nodes. An example of a flowgraph and its corresponding weighted adjacency matrix is shown in Figure 5.5.



	1	2	3	4	5
1	0	0	*a*	0	0
2	0	0	0	0	0
3	0	*d*	0	*b*	0
4	0	*c*	0	0	*f*
5	0	*g*	*e*	0	0

Figure 5.5 A flowgraph and its corresponding weighted adjacency matrix

Referring to the Figure 5.5, each node on the flowgraph is identified by numbers, while each edge is identified by letters. A letter entry is made in the matrix to represent a connection between two nodes. For example, node 3 is connected to node 4 by edge *b*. An adjacency matrix can be defined as a tabular representation of a flowgraph. However, by adding a link weight to each matrix entry, the adjacency matrix becomes a powerful tool for evaluating program control structure during testing. The link weight provides additional information about control flow. In its simplest form, the link weight equals 1 (a connection exists) or 0 (a connection does not exist), but link weights can be assigned other properties:

- The probability that a connection (edge) will be executed.
- The processing time expended during traversal of a link.
- The memory required during traversal of a link.
- The resources required during traversal of a link.

As an example, we use the simplest weighting to indicate connections (0 or 1). The weighted adjacency matrix in Figure 5.5 is converted to an adjacency matrix in Figure 5.6.

	1	2	3	4	5
1	0	0	1	0	0
2	0	0	0	0	0
3	0	1	0	1	0
4	0	1	0	0	1
5	0	1	1	0	0

Figure 5.6 Adjacency matrix of the flowgraph shown in Figure 5.5

Each letter has been replaced with a 1, indicating that a connection exists (zeros have been excluded for clarity). It is possible to apply metrics to the adjacency matrix to determine the various properties of software. For this matrix, we will use the Cyclomatic Complexity metric (McCabe 1976) as an example.

In the connectivity matrix, each row with two or more entries represents a predicate node; therefore, performing the procedure below provides us a method for determining Cyclomatic Complexity. The control flowgraph can be implemented using the adjacency matrix data structure. The procedure for creating the adjacency matrix from a control flowgraph is:

1. A two-dimensional array is declared. Its size is equal to the number of nodes in the flowgraph.
2. The adjacency matrix is constructed. If there is an edge from node *i* to node *j*, then $A_{ij} = 1$. If there is not an edge from node *i* to *j*, then $A_{ij} = 0$.

The complexity can be determined from this adjacency matrix using the following procedure:

1. Count the number of 1's in each row. Subtract 1 from these values and save the results.
2. Add the results from Step 1. This value added by 1 gives the Cyclomatic Complexity.

For the example given in Figure 5.6, the Cyclomatic Complexity calculation is demonstrated in Figure 5.7:

	1	2	3	4	5	RS-1
1	0	0	1	0	0	0
2	0	0	0	0	0	0
3	0	1	0	1	0	1
4	0	1	0	0	1	1
5	0	1	1	0	0	1

Figure 5.7 Adjacency matrix and RS-1 (RS denotes the row sum)

The sum of the values of the last column in the matrix in Figure 5.7 is 3. According to procedure mentioned before, the Cyclomatic Complexity of this flowgraph is $n+1=3+1=4$.

5.3 Entropy and Information Content of Software

5.3.1 Entropy of Software

Entropy is a fundamental concept in information theory-based measurement. Observing the interactions among software components leads to the measurement of complexity, coupling, cohesion, and other software engineering concepts. Software measurement relates the information content of the software to the complexity of the software, thus the quantification of the amount of information is used to assess the functional complexity of the system and the required quality improvements (Alagar and others 2000).

Entropy is a measure of the average information rate of a message or language. The output emitted by a discrete source during every unit of time could be a string of symbols drawn from a finite alphabet. The probability of the occurrence of the symbols defines the information content of the message $I = \log_2 p_i$. The amount of information increases as the number of symbols increase in a system. Based on the additivity princi-

ple of entropy, explained in Chapter 2, it could be said that the amount of information conveyed by two symbols is the sum of their individual information contents. The average information, provided by a memoryless source, can be computed using several entropies. For software measurement, Shannon entropy is the most suitable, since it satisfies many properties, such as nonnegativity, subadditivity, and expansibility.

Much of the current measurement work based on entropy relates to the complexity of software. Complexity is quantified using entropy by means of an abstraction of the interactions among software components (Abran and others 2004). The information exchange between the software and the environment, and within the software components, occurs through messages. Messaging between the user and the outside environment consists of user inputs and software responses. In OO programming, class complexity is determined by measuring the information flow in a class, based on the information-passing relationship among member data and member functions. The inter-object complexity of a program measures information flow between objects. Total complexity is measured by class complexity and inter-object complexity (Abd-El-Hafiz 2001).

Other aspects of software that can be measured by means of information theory include cohesion and coupling. Seker's research (Seker and Tanik 2004) focused on modeling CBS systems as noiseless channels, and thereby assessed coupling and cohesion in CBS systems. This dissertation follows an information-theoretic approach by measuring the entropy of messages among the software components. We define a set of metrics to measure the complexity, coupling, and cohesion of the system, based on the entropies of the intermodule and intramodule components. Before this method is thor-

oughly explained, the information content of software systems and related terms, such as coupling and cohesion, need to be clearly defined.

5.3.2 Information Content of Software Systems

It is possible to recognize source code as text, but it is usually misleading to visualize it solely as expressions generated by a grammar. Source code consists of independent and interrelated texts, and its semantics are determined by a compiler or a formal semantic theory (Malton and others 2001). Source code is independent if maintained as itself, rather than being dependent on prior data. It is also interrelated because all of the links among the code arise through abstraction, including lexical and semantic links. A lexical analysis of source code determines the attributes of software by a single pass through the code. Using this approach, it is possible to classify code into words and symbols. It is possible to further this classification by sorting words into two groups, operands and operators. All programming languages, specification languages, natural languages, and pseudocode documents can be examined using this method. It is possible to represent source code as a stream of characters drawn from the designated character set of the language; however, this representation would be of limited interest, except for the purpose of storing and transmitting the code (Edwards 2003). A word-based textual model is preferred over other methods, as words refer to tangible elements of computation and can be looked up in a dictionary using lexical analysis.

The results of lexical analysis form the basis of many software measurement methods. Lines of code, statement count, and number of operands and operators (Hal-

stead 1977) are examples of lexical measures. These measures are often observed as highly correlated (Fenton and Pfleeger 1998).

5.3.3 Measurement of Information Content of Software

There are several methods for measuring the information content of software. An important distinction among these methods is the way that the software is modeled prior to the computation stage. Several methods use lexical measures and define the source code as text before applying the measures. The other methods are based on modeling the software using graph theory. These approaches use control flowgraphs or data dependency graphs and apply the measurement methods to the graphs, rather than applying them to the source code itself. The measurement method suggested by this dissertation can be classified under this approach. It is possible to compute the individual component's entropy using a lexical approach when appropriate; combined with the information-theoretic methods, this measurement framework can be defined as a composite framework. This type of composite framework can be advantageous to individual measures due to their limitations. The composite measures can be derived from statistical analysis of empirical data, or from theoretical models of expected interactions between different factors.

The next section explains how an individual software component's average information content, and the total information content of a system, can be calculated using entropy. Halstead's (1977) software science is an established measure using a similar approach to calculate programming volume and effort, so it will be compared to entropy-based measurement using an example in the next section.

5.3.4 How to Measure a Software Component's Entropy

The black-box nature of components masks their internal structure, including their source code; however, the developer of the component can use measurement methods to test the component, which may include scanning the source code. Source code can be viewed as a stream of words and symbols that lead the way to a lexical analysis. Let $\{a_1,...,a_n\}$ be the vocabulary of the source code and let $\{p_1,...,p_n\}$ be the assigned probabilities of occurrence to $\{a_1,...,a_n\}$. Then the information content of a_i is $-\log_2 p_i$. This expression implies that the less probable, more surprising, message has higher information content.

The entropy of the system (component) is described as the sum of the probability-weighted average information content

$$H = \sum_{i=1}^{n} -\log_2 p_i \text{ in units of bits per symbol.} \tag{5.10}$$

At this point, let us review Halstead's definitions of vocabulary and volume. Let n_1 be the number of unique operators, n_2 the number of unique operands. $n = n_1 + n_2$ is called Halstead's vocabulary. Let N_1 be the number of operators in a program and let N_2 be the number of operands in a program. $N = N_1 + N_2$ is called Halstead's length. The volume is defined as $V = N \log_2 n$. Halstead's volume equation combines length and vocabulary to reflect the complexity of reading the text and accessing each symbol in a dictionary or table, assuming $O(\log n)$ equals the average time to access an optimized table of size n (Edwards 2003).

If the symbols in question are the operators and operands of a program, then these probabilities can be estimated by frequencies of occurrence in the program, the whole

system, or previous programs, and entropy can be calculated. This operation gives the average information content per symbol and the result can be multiplied by the length in symbols to give the information content for the program. This procedure is a refinement of Halstead's volume, and if all symbols are assumed to be equally likely, then *H* reduces to $\log_2 n$, and information content reduces to *V* (Berlinger 1980; Harrison 1992).

An example is given to perform these calculations; the results are given in Table 5.1, and also in the following section. A lexical analysis is applied to the following text and then the counts, probabilities, and information contents of words and punctuation marks are found (Edwards 2003):

"This monograph deals with plausible techniques for analyzing software documents to extract significant information relevant to the software development process. A "software document" may possibly be specification, design, or code, and possibly written in a programming language, a specification language, pseudo-code, or even natural language. However, it must describe a computation process in sufficient detail for the level of interest of the analyst, and it must be written in a language of sufficiently formal syntax for algorithmic analysis. The goal of this analysis is assumed to be the support of the software development process - informing the manager, the programmer, the tester, or quality analyst in answering questions necessary to decision making, evaluating work done and allocating resources for work yet to be done. The analysis may also have the basic scientific goal of understanding the process itself - how people write and understand software systems and what regularities may be expected in the creation, testing, and use of software systems. Questions to be answered may involve the function and runtime behavior of the software: Is this program a correct implementation of this specification? Will it fail to terminate or terminate badly on some inputs? Which parts of the code are most used? Are some parts of the code never used? Can some some data items be used without being defined, or defined and never used? Is this program, or system, "efficient" in its use of resources? What is the "quality" of this system or its parts? Or they may apply to the development process: What is the expected development time and effort of this (not yet implemented) system as specified? How understandable is this system? What is the expected testing time of this (implemented) system? How should testing or maintenance effort be distributed among the parts of this system? The answers to such questions are assumed to depend on relevant attributes of the software document, which are immediate products of analysis, and usually, but not always, numeric - software measures or "metrics". Since they are extracted from the "products" of the development process, they are what are usually called *product measures*, and relate to *process* measures such as development time and number of errors, and to *resource* measures such as number of persons and system time."

Table 5.1 Lexical Analysis for Calculating the Information Content

Symbols	# Count	p_i	$\log_2 p_i$	Symbols	# Count	p_i	$\log_2 p_i$
the	22	0.0608	4.0404	may	5	0.0138	6.1779
,	20	0.0552	4.4779	some	4	0.0110	6.4998
of	13	0.0359	4.7994	for	4	0.0110	6.4998
to	10	0.0276	5.1779	language	4	0.0110	6.4998
and	9	0.0249	5.3299	are	4	0.0110	6.4998
“	8	0.0221	5.4998	a	4	0.0110	6.4998
or	8	0.0221	5.4998	analysis	4	0.0110	6.9149
software	8	0.0221	5.4998	parts	3	0.0083	6.9149
be	8	0.0221	5.4998	is	3	0.0083	6.9149
?	7	0.0193	5.6925	they	3	0.0083	6.9149
this	6	0.0166	5.9149	it	3	0.0083	6.9149
process	6	0.0166	5.9149	used	3	0.0083	6.9149
in	6	0.0166	5.9149	system	3	0.0083	6.9149
development	5	0.0138	6.1779	specification	3	0.0083	6.9149

Halstead’s metrics can be calculated based on the lexical analysis results given in Table 5.1:

- The length of the text is 1690 symbols/words. The size of the vocabulary is 45.
- $N = 1690$, $n = 45$, $\log_2 n = 5.4918$.
- Halstead’s volume is $V = N \log_2 n = 9281.2317$.
- Entropy of the text is $H = \sum_{i=1}^{n} -\log_2 p_i = 4.3775$.
- Information content of the text is $IC = N \times H = 7398.0022$.

Halstead’s (1977) work is similar to the entropic approach, since they both measure the information content of modules. It is an efficient method to measure the overall size of the software when the source code is available; however, the nature of CBD dictates the rule of examining the components only through their interfaces, not through their source code. This rule makes Halstead’s method ineffective when the user does not

have access to all of the source code for the components used. This lack of access to source code is a common case in CBD, so a measurement approach based on component interactions would be more useful. Coupling and cohesion analysis are common approaches used to understand the relationship of the components within a module and among several modules. The measurement method proposed in this dissertation examines coupling and cohesion as well, though with a unique approach utilizing entropy. Entropy has been used mostly for complexity analysis in the natural sciences, and even in software engineering for the last fifteen years since Harrison's work (1992). The method used in this dissertation differs from most approaches since the source code's entropy is not calculated, but the entropy analysis is made according to component interactions.

The next section briefly revisits control flowgraphs and cubic control flowgraphs, as they are the tools used to represent the software in this framework. The measurement approach is explained using a detailed example representing some of the possible constructs in CBD. The results of this example are compared to some of the established software metrics' results in Chapter 6. Chapter 6 and the Appendix also provide empirical validation of the metrics.

5.4 Representation of Programs with Control Flowgraphs

Control flowgraphs are useful in representing the logic flow throughout the program and the control structure (Seker 2002). There have been various measures that use flowgraphs, and certain metrics have been used in flowgraphs to assess quality attributes of programs. Flowcharts have been commonly used to represent programs and algorithms; however, they do not support graph-theoretical methods (two edges can join with-

out needing a vertex which rules out flowcharts being defined as graphs). Converting flowcharts to flowgraphs provides a wide range of tools from graph theory. A flowchart can be converted into a directed graph by introducing junction nodes where two lines meet without a vertex. The resulting graph is called a control flowgraph.

A control flowgraph of a program module is contructed from a directed graph representation of the program module that can be defined as (Munson 2003): A directed graph, $G = (N, E, s, t)$ consists of a set of nodes, *N*, a set of edges *E*, a distinguished node *s*, the start node, and a distinguished node *t*, the exit node. An edge is an ordered pair of nodes (*a, b*). The in-degree $I(a)$ of node *a* is the number of entering edges to *a*. The out-degree $O(a)$ of node *a* is the number of exiting edges from *a*.

The flowgraph representation of a program, $F = (E', N', s, t)$, is a directed graph that satisfies the following properties:

- A set of edges E' connecting the elements of N'.
- There is a unique start node *s* such that $I(s) = 0$.
- There is a unique exit node *t* such that $O(t) = 0$.
- All other nodes, $n \leftarrow N'$, are members of exactly one of the following three categories.
    - Processing node. It has one entering edge and one exiting edge.
    - Predicate node. It represents a decision point in the program as a result of if statements, case statements, or any other statement that causes an alteration in the control flow.
    - Receiving node. It represents a point in the program where two or more control flows join, for example, at the end of a while loop.

Definition 5.4 (Path): A path *P* in a flowgraph *F* is a sequence of edges, $\langle a_1a_2, a_2a_3, ..., a_{n-1}a_n \rangle$, where all $a_i (i = 1, ..., N)$ are elements of N'. *P* is a path from node a_1 to node a_N. An execution path in *F* is any path from *s* to *t*.

Definition 5.5 (Cycle): A program may contain iterative structures created by if and while statements. These are called cycles.

Definition 5.6 (Loop): A loop is a cycle of length one (one vertex and one edge).

Flowgraph connectedness:

- A flowgraph *F* is weakly connected if any two nodes *a* and *b* in the flowgraph are connected by a sequence of edges.
- A flowgraph *F* is strongly connected if *F* is weakly connected and each node of *F* lies on a cycle.

Any flowgraph can potentially contain weakly connected subsets of nodes that are flowgraphs in their own right. To examine this potential hierarchical structure of the flowgraph representation, the notion of subflowgraph is essential:

- A subflowgraph $F' = (E'', N'', s', t')$ of a flowgraph $F = (E', N', s, t)$ is a flowgraph if the out-degree of every node in F' is the same as the out-degree of the corresponding node in *F*, with the exception of the nodes s' and t'. Further, all nodes in the subflowgraph are weakly connected only to nodes in N''.
- A subflowgraph of *F* is a subflowgraph with the property that the cardinality of $N'' > 2$ and $F' \neq F$. That is, the sub flowgraph must contain more nodes than the start and exit nodes and cannot be the same flowgraph.

A flowgraph is an irreducible flowgraph if it contains no proper subflowgraph. A flowgraph is a prime flowgraph if it contains no proper subflowgraph for which the prop-

erty $I(s') = 1$ holds. A prime flowgraph cannot be built by sequencing or nesting of other flowgraphs and contains a single entrance and single exit structure. The primes are the primitive building blocks of a program control flow (Munson 2003).

Cubic control flowgraphs are a special subset of control flowgraphs that have been used in software measurement. Prather (1995) used CCFG properties in designing and analyzing software metrics. CCFGs can be decomposed into their components using the methods Prather identified. The decomposition process of CCFGs has been explained in Chapter 2.

5.5 Measurement of the Structural Complexity of Software

Structural complexity can be defined as the organization of program elements within a program (Gorla and Ramakrishnan 1997). A software design decision could change the structural complexity, which could lead to changes in effort and quality. Measures of structural complexity are internal attributes that are specific to the software code artifact (Darcy and others 2005). There exist a great number of metrics that measure different aspects of structural complexity. They are sometimes referred to as inter-modular metrics, design metrics, or system metrics because they express different types of relationships between modules in the system and can often be calculated already during the design phase of a project (Krusko 2004).

Coupling and cohesion are fundamental concepts of software that indicate structural complexity. The OO programming paradigm emphasized theoretically-based measures for software, including coupling and cohesion measures. The general consensus has been that lower coupling and higher cohesion leads to better software designs (Fenton

and Pfleeger 1998). In general, cohesiveness indicates that the entire module's parts work toward some common goal or function. That means that a module with good cohesion will use all of its parts to the solution of a particular function when it is invoked.

Yourdon and Constantine (1979) proposed classes of cohesion that provide an ordinal scale of measurement:

- Functional. The module performs a single well-defined function.
- Sequential. The module performs more than one function, but they occur in an order prescribed by the specification.
- Communicational. The module performs multiple functions, but all on the same body of data (which is not organized as a single type of structure).
- Procedural. The module performs more than one function, and they are related only to a general procedure affected by the software.
- Temporal. The module performs more than one function, and they are related only by the fact that they must occur within the same timespan.
- Logical. The module performs more than one function, and they are related only logically.
- Coincidental. The module performs more than one function, and they are unrelated.

These categories of cohesion are listed from most desirable (functional) to least desirable.

Coupling is the degree of interdependence between modules (Yourdon and Constantine 1979). Usually coupling expresses interdependence between two modules, rather

than between all modules in the system. Global coupling is the measure of coupling for the whole system and can be derived from the coupling among the possible pairs

Given modules *x* and *y*, we can create a classification for coupling, defining six relations on the set of pairs of modules:

- No coupling relation R_0. Modules *x* and *y* have no communication; they are totally independent of one another.
- Data coupling relation R_1. Modules *x* and *y* communicate by parameters, where each parameter is either a single data element or homogeneous set of data items that incorporate no control element. This type of coupling is necessary for any communication between modules.
- Stamp coupling relation R_2. Modules *x* and *y* accept the same record type as a parameter. This type of coupling may cause interdependency between otherwise unrelated modules.
- Control coupling relation R_3. Modules *x* passes a parameter to *y* with the intention of controlling its behavior, that is, the parameter is a flag.
- Common coupling relation R_4. Modules *x* and *y* refer to the same global data. This type of coupling is undesirable; if the format of the global data must be changed, then all common-coupled modules must also be changed.
- Content coupling relation R_5. Modules *x* refers to the inside of *y*; that is, it branches into, changes data in, or alters a statement in *y*.

The relations are listed from the least dependent at the top to most dependent at the bottom, so that $R_i > R_j$ for $i > j$. Modules *x* and *y* are considered loosely coupled

when *i* is 1 or 2 and tightly coupled when *i* is 4 or 5. Coupling reflects the degree of relationship that exists between modules. The more tightly coupled two modules are, the more dependent on each other they become. Coupling is closely associated with the fact that a change or fault in a module could affect the other modules it is coupled with.

5.6 Using Entropy to Examine Structural Complexity

As described in the previous section, coupling and cohesion are the primary measures of structural complexity. The information flow throughout the system has also been measured for determining structural complexity. The information flow complexity metrics using length, fan-in, and fan-out by Kafura and Henry (1987), and the Shepherd complexity (1993) which concentrates on the flow information of a module, are other metrics that have examined structural complexity. Belady and Lehman (1976) examined the decrease of the structuredness of software with increasing entropy. They observed that the entropy can result in severe complications when the project is modified, causing maintenance issues.

As large-scale software grows, there is an added level of difficulty analyzing the complexity. The tools that exist in software engineering do not sufficiently address the decomposition and representation issues in large networks. Information theory, specifically entropy, could provide the methods to deal with the structural complexity of software. The framework presented here uses the best known entropy, Shannon entropy, to find the complexity of a module and also its coupling and cohesion. To understand the applicability of entropy to software modules, the following example is given by Munson

(2003). In software, if all modules execute with an equal probability, the point of maximum entropy will occur, in which case

$$h_{\max} = \sum_{i=1}^{m} q_i \log_2 q_i \ . \tag{5.11}$$

It is a point of maximum surprise that all modules are executed in precisely the same amount of time under a given input scenario. At the other extreme, the point of minimum entropy, 0, is the point at which the program will execute in only one of its modules. The probability of executing in only one module will be 1. The probability of executing the other modules will be 0. The entropy of a software module or systems is proportional to its complexity, in other words, the degree of disorder. By measuring the entropy of each module, we found a complexity metric that should be the basis of two other metrics, cohesion and coupling metrics based on entropy. In the next chapter, the measurement framework is explained with an example, and the results are compared to some known software metrics, such as Cyclomatic Complexity, Lack of Cohesion, and Efferent Coupling.

Summary

In this chapter, the previous efforts relating to the use of entropy as a measure for software were summarized. The necessary mathematical background relevant to adjacency and connectivity matrices were explained. A representation of flowgraphs using adjacency and connectivity matrices were described with examples. The chapter continued with a summary of the measurement of information content in software and related issues, such as entropy measurement, lexical analysis, and Halstead (1977) metrics. Measurement of coupling and cohesion, utilizing CCFGs, were discussed from a struc-

tural complexity perspective. The last section introduced the idea of how the entropy concept helps deal with the structural complexity issues in software engineering. The next chapter explains the entropic measurement framework using an application consisting of several components.

CHAPTER 6

CASE STUDY, CONCLUSION, AND FUTURE WORK

The sample program given in Figure 6.1 is used to apply the entropy-based metrics. The program was initially used in Seker's dissertation (2002) to find the pervasive Shannon metrics. The sample algorithm is used in this dissertation to make the metrics comparable to Seker's study, since they have common left and right eigenvectors based on the Perron-Frobenius theorem. However, in this dissertation, the measurement is performed to obtain entropy-based metrics for the system (related to its information content, complexity, coupling, and cohesion). In addition, a modularized Java implementation is written according to the algorithm in order to perform the baseline metric calculations, such as Cyclomatic Complexity (McCabe 1976), Coupling Between Objects, Lack of Cohesion, and others, within the Java Eclipse environment.

Measures based on the entropy concept (information content, complexity, cohesion, and coupling) are applicable to any flowgraph; however, the case study is limited to CCFGs due to their established relationship to component-based software. The CCFGs are labeled according to the parameter of interest. In this study, the parameter of interest is time.

The initial step in the measurement process is to represent the program with its CCFG. The CCFG can be decomposed, based on the user's needs, in a hierarchical structure. The next step is the representation of the CCFG using its adjacency matrix, where

the weights are assigned or measured for the edges. As we have values corresponding to the edges (including the hidden components along them), we will use a weighted adjacency matrix for representing the CCFG. Perron-Frobenius theorem provides the methods to determine the edge and node probabilities, based on the adjacency matrix. The sample program is represented with its control flowgraph in Figure 6.1.

```
Program Sample( )
begin
A;
  if P1 then
  B;
  else
  C;
  endif
D;
  while P2 do
  repeat
  E;
  F;
  until P3
    if P4 then
    G;
      if P5 then
      H;
      else
      J;
      endif
      K;
    else
    L;
    endif
   M;
   endwhile
N;
end
```

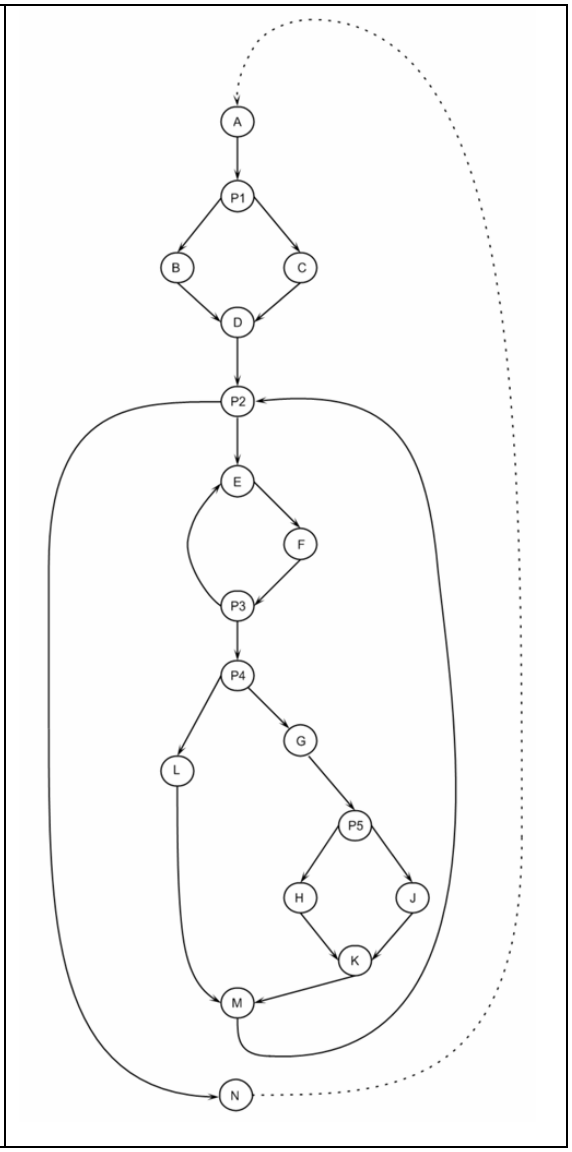


Figure 6.1 The sample program and its control flowgraph

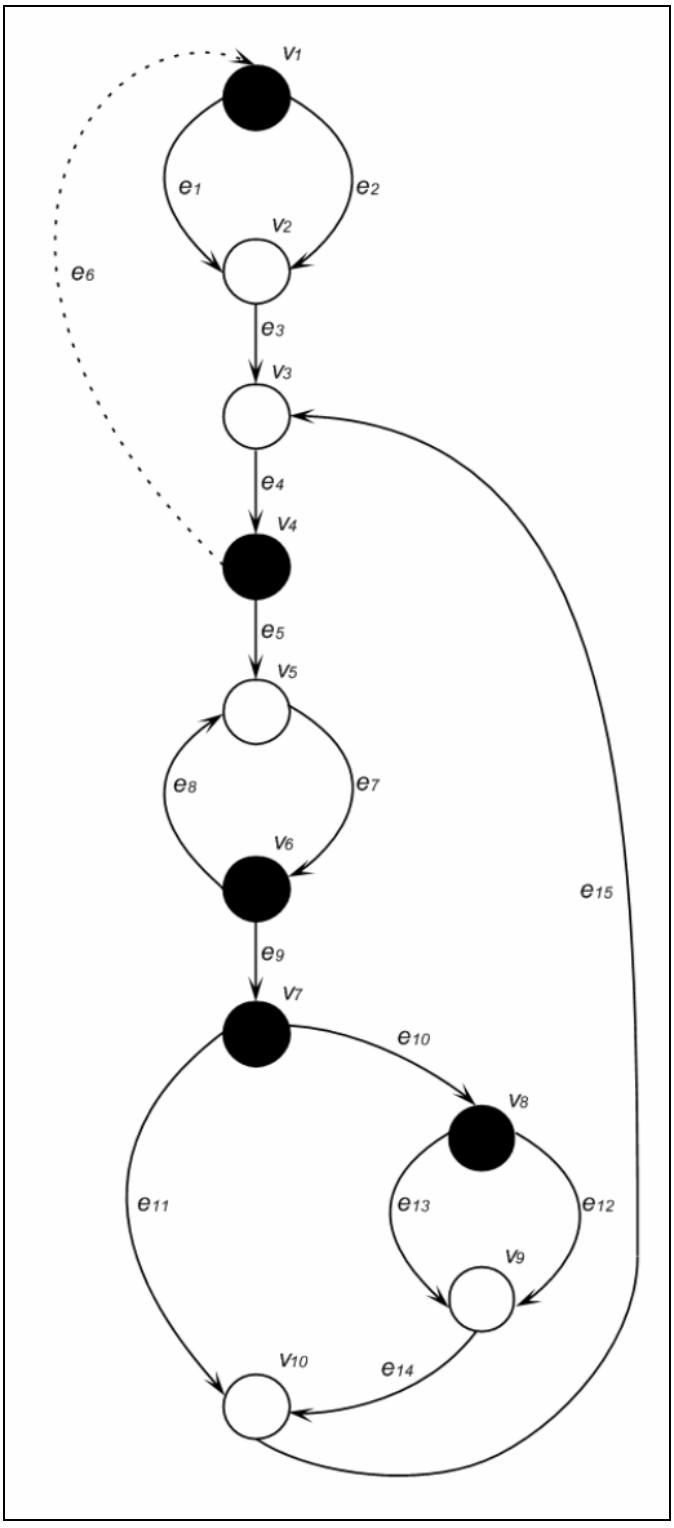


Figure 6.2 CCFG of the program

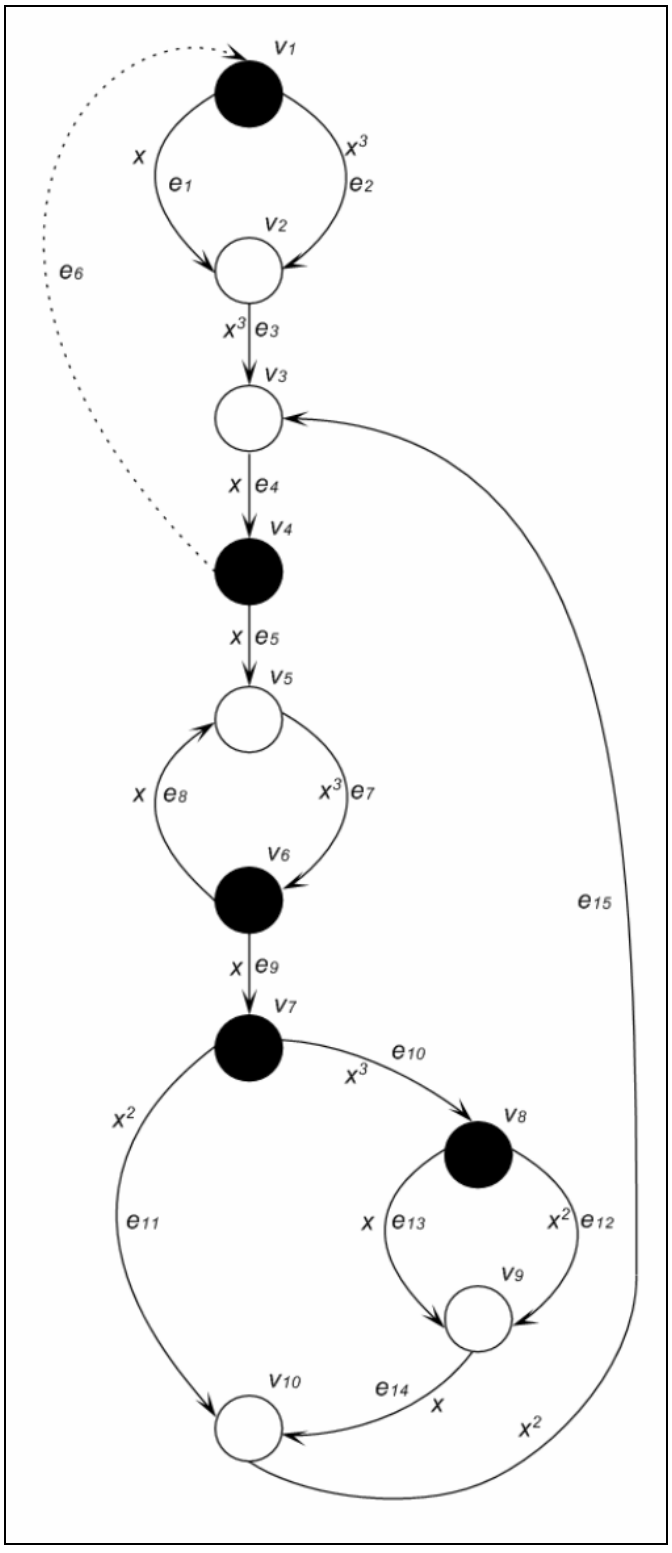


Figure 6.3 CCFG of the program with edge weights

6.1 Calculations of Edge Probabilities and Stationary Probability Distribution for the Nodes Using Perron-Frobenius Theorem

Suppose $A \in R^{n \times n}$ is a nonnegative and regular matrix, i.e., $A^k > 0$ for some *k*. Perron-Frobenius Theorem states:

- There is an eigenvalue λ_{pf} of *A* that is real and positive, with positive left and right eigenvectors.
- For any other eigenvalue λ, we have $|\lambda| < \lambda_{pf}$.
- The eigenvalue λ_{pf} is simple, i.e., has multiplicity one, and corresponds to a 1×1 Jordan block. The eigenvalue λ_{pf} is called the Perron-Frobenius eigenvalue of *A*.

Considering that CCFGs are directed graphs, and that each has associated with a nonnegative matrix with all entries 0 or 1 following:

- a_{ij} = 1 if there is an arc from vertex *i* to vertex *j*.
- a_{ij} = 0 if there is no arc from vertex *i* to vertex *j*.

If the associated matrix is irreducible, then one can get from any vertex to any other vertex (perhaps in several steps), whereas if it is reducible (like the Markov Chain case), there are vertices from which one cannot travel to all other vertices. The former case, in the realm of graph theory, is called a strongly connected graph. The irreducible matrix $P(x_0)$ has left and right eigenvectors, ξ and η, that are unique such that

$$\xi P(x_0) = \xi, \tag{6.1}$$

$$P(x_0)\eta = \eta, \tag{6.2}$$

$$\xi \cdot \eta = 1. \tag{6.3}$$

Then, the edge probabilities are

$$p_b^* = \frac{\eta_{dest(b)}}{\eta_{orig(b)}} x_0^{\tau(b)} \tag{6.4}$$

where

dest(b) is the destination node for arc *b*, and *orig(b)* is the starting node (origin) of arc b.

The corresponding stationary probability distribution for the nodes is

$$\pi_{v_i}^* = \xi_{v_i} \eta_{v_i}, \; v_i \in V \tag{6.5}$$

where

V is the set of vertices (nodes) of the digraph.

In the following example, the Perron-Frobenius theorem is applied to the CCFG of the program shown in Figure 6.3. The theorem is applied according to the weights of the components hidden on the edges of the CCFG. The weighted adjacency matrix for the CCFG is

$$P(x) = \begin{bmatrix} 0 & x+x^3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & x^3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 0 & 0 \\ x & 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & x^3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & x & 0 & x & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & x^3 & 0 & x^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x+x^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x \\ 0 & 0 & x^2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Using the Perron-Frobenius theorem, the corresponding left and right Perron eigenvectors ξ and η are

$$\xi = \begin{bmatrix} 0.27290481 \\ 0.37357203 \\ 0.40668715 \\ 0.33314694 \\ 0.49646093 \\ 0.27290481 \\ 0.22355612 \\ 0.12288891 \\ 0.18313103 \\ 0.30003181 \end{bmatrix}^T \quad \text{and} \quad \eta = \begin{bmatrix} 0.29386395 \\ 0.21467583 \\ 0.39053238 \\ 0.47674009 \\ 0.28811369 \\ 0.52412851 \\ 0.35171309 \\ 0.31991339 \\ 0.21467583 \\ 0.26206425 \end{bmatrix}.$$

The corresponding edge and node probabilities can be computed by using (6.6) and (6.7)

$$p_b^* = \frac{\eta_{dest(b)}}{\eta_{orig(b)}} x_0^{\tau(b)} \tag{6.6}$$

$$\pi_{v_i}^* = \xi_{v_i} \eta_{v_i} . \tag{6.7}$$

The edge and node probabilities calculated according to the left and right eigenvectors are given in Tables 6.1 and 6.2. For the first edge, e_1, based on (6.6), the probability is

$$\frac{\eta_{v_2}}{\eta_{v_1}} x_0^1 = \frac{0.29386395}{0.21467583} 0.81917251 = 0.59842842 .$$

For the second edge, e_2, based on (6.6), the probability is

$$\frac{\eta_{v_2}}{\eta_{v_1}} x_0^3 = \frac{0.29386395}{0.21467583} \left(0.81917251\right)^3 = 0.40157157 .$$

Table 6.1 Edge Probabilities for the CCFG

Edge	Edge Probability
e_1	0.59842842
e_2	0.40157157
e_3	1
e_4	1
e_5	0.49505972
e_6	0.50494027
e_7	1
e_8	0.45029952
e_9	0.54970048
e_{10}	0.5
e_{11}	0.5
e_{12}	0.45029951
e_{13}	0.54970046
e_{14}	1
e_{15}	1

For the first node, v_1, based on (6.7), the node probability is

$$\pi_{v_1} = \xi_{v_1}\eta_{v_1} = 0.272904081 \cdot 0.29386395 = 0.08019689 .$$

For the second node, v_2, based on (6.7), the node probability is

$$\pi_{v_2} = \xi_{v_2}\eta_{v_2} = 0.37357203 \cdot 0.21467583 = 0.08019689 .$$

Table 6.2 Node Probabilities for the CCFG

Node	Node Probability
v_1	0.08019689
v_2	0.08019689
v_3	0.15882450
v_4	0.15882450
v_5	0.14303719
v_6	0.14303719
v_7	0.07862761
v_8	0.03931380
v_9	0.03931380
v_{10}	0.07862761

The edge and node probabilities are the key components of the measurement framework. As entropy is directly related to probability, the edge and node entropies are computed using (6.8) based on the probabilities given in Tables 6.1 and 6.2. The program represented by the CCFG is composed of four modules, as illustrated in Figure 6.4. The modularity of the application allows us to study the application using a modern programming approach. The relationships between the modules are examined using the Interactive Coupling metrics, and the relationships between the nodes within the modules are examined using the Interactive Cohesion metrics.

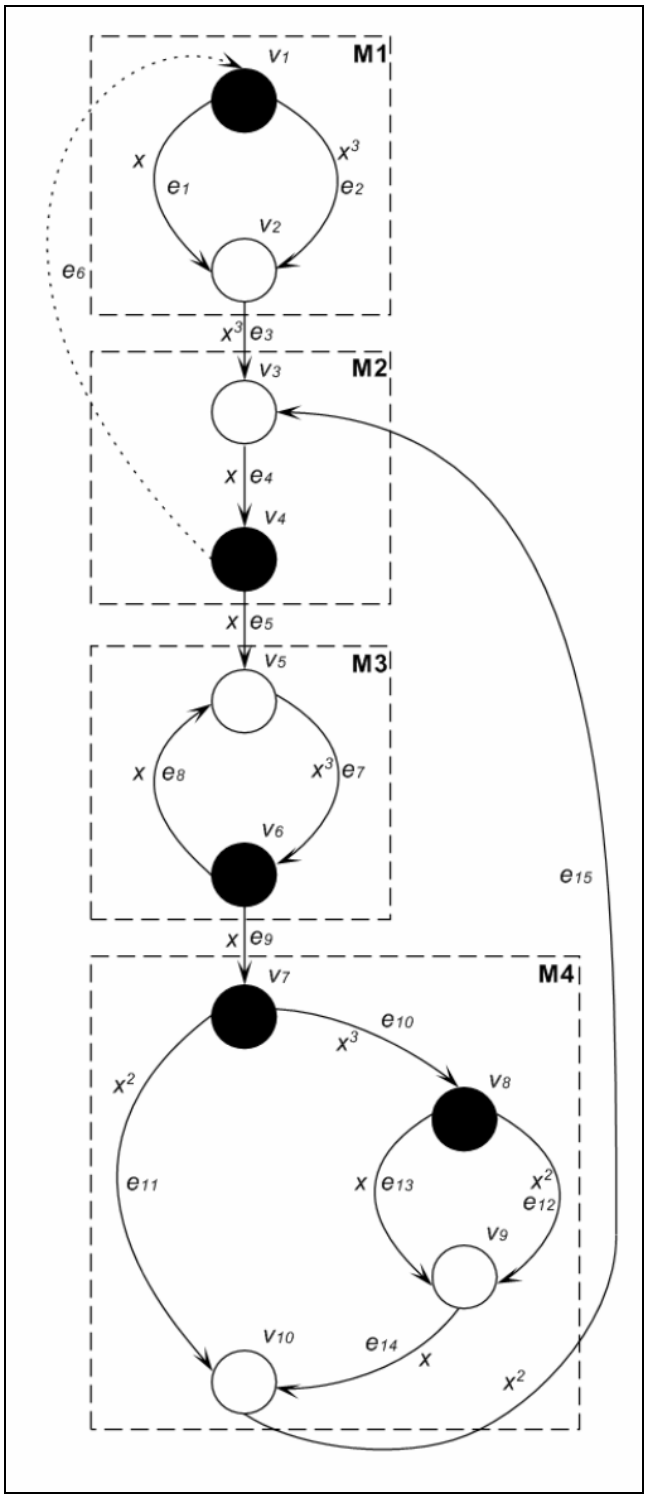


Figure 6.4 Modules of the CCFG

Before entropy-based metrics are applied, it is necessary to calculate the edge and node entropies of the CCFG shown in Figure 6.2 using the probabilities obtained in Tables 6.1 and 6.2. Shannon Entropy is calculated using

$$H_n(P) = -\sum_{i=1}^{n} p_i \log_2 p_i \tag{6.8}$$

and the resulting entropies are listed in Tables 6.3 and 6.4.

Table 6.3 Node Entropies for the CCFG

Node	p_{v_i}	$\log_2 p_i$	H_{v_i}
v_1	0.08019689	3.642	0.291
v_2	0.08019689	3.642	0.291
v_3	0.15882450	2.654	0.421
v_4	0.15882450	2.654	0.421
v_5	0.14303719	2.805	0.401
v_6	0.14303719	2.805	0.401
v_7	0.07862761	3.669	0.288
v_8	0.03931380	4.669	0.018
v_9	0.03931380	4.669	0.018
v_{10}	0.07862761	3.669	0.288

Table 6.4 Edge Entropies for the CCFG

Edge	p_{v_i}	$\log_2 p_i$	H_{v_i}
e_1	0.59842842	0.7408	0.4432
e_2	0.40157157	1.3165	0.5285
e_3	1	0	0
e_4	1	0	0
e_5	0.49505972	1.0144	0.5021
e_6	0.50494027	0.9858	0.4977
e_7	1	0	0
e_8	0.45029952	1.1513	0.5183
e_9	0.54970048	0.8632	0.4745
e_{10}	0.5	1	0.5
e_{11}	0.5	1	0.5
e_{12}	0.45029951	1.1513	0.5183
e_{13}	0.54970046	0.8632	0.4745
e_{14}	1	0	0
e_{15}	1	0	0

6.2 Calculation of the Metrics

6.2.1 Design Information Content (δ)

Information content concept has been used frequently in software measurement in the past, as explained in Chapter 5. As Alagar and others (2000) defined it, software measurement relates the information content of the software to the complexity of it, thus the quantification of the amount of information is used to assess the functional complexity of the system and the required quality improvement. Some of the methods to compute the information content are lexical methods and assume the source code as text before applying the measures. The others are based on modeling the software using graph theory. These approaches use control flowgraphs or data dependency graphs and apply the measurement methods on the graphs rather than the source code itself. The method used here employs the CCFG to represent the software and apply the computation to individual modules represented in the CCFG. Since the CCFG represents the design structure of the software, the metric is called the design information content metric.

The method for computing the metrics involves finding the total entropy of the nodes. The node probabilities are also stationary state probabilities in Markov Chains. From the node probability table, it is possible to determine the probability of a message in a program passing through a node. Due to the additivity of entropy, all node entropies are added to find the total entropy of the system. As known in information theory, information content is proportional to entropy. In section 5.3.4, the information content is defined as a product of entropy times the length of the program. For the CCFG it would be the entropy times the total number of nodes. The information content of the CCFG is thus proportional to the number and probabilities of the nodes. In the case of all nodes having

the same stationary state values, the entropy and information content would be the maximum. Entropy is the average information per node in CCFG.

The Design Information Content metric (δ) is the multiplicative product of the total number of nodes and the summation of their entropies

$$\text{Design Information Content} = \text{Total Node Entropy} \times \text{Number of Nodes}. \quad (6.9)$$

The stationary state probabilities for the nodes are computed based on all nodes, so it is not possible to use δ metric for each individual module. Each node's probability is calculated considering the whole system as the state space, thus δ metric should be computed for the whole system.

For the CCFG given in Figure 6.3, the total number of nodes is 10. The node probabilities and their entropies are given in Table 6.3. The total node entropy is

$$H_{VN} = H_{V1} + H_{V2} + ... + H_{V10}. \quad (6.10)$$

The following example is constructed to demonstrate how to find the Design Information Content of a CCFG. We calculate the total node entropy of the example given in Figure 6.2 based on (6.10). The first step is to find the total node entropy

$$H_{VN} = 0.291 + 0.291 + 0.421 + ... + 0.288 = 2.838 \text{ bits}.$$

The second step is to calculate the Design Information Content based on (6.9)

$$\delta = 2.838 \cdot 10 \text{ nodes} = 28.38 \text{ bits}.$$

6.2.2 Interaction Complexity (χ)

A CCFG is composed of nodes and edges that provide an abstraction of the software system and the subgraphs representing software modules. The edges represent the interaction between components and modules, thus the complexity of the software is af-

fected by this interaction. The Interaction Complexity metric (χ) of a module is the total entropy of the edges within that module

$$\chi = \sum H_e \tag{6.11}$$

where

H_e is the entropy of the edges.

As the number of edges increase within a module, the metric has a higher value indicating a higher interaction complexity. As the entropy of the edges increases, corresponding to a higher uncertainty of the system, we observe a higher interaction complexity.

In the following example, the Interaction Complexity metric is calculated for each module of the CCFG given in Figure 6.2. The metric calculations are based on (6.11) and the edge entropies (listed in Table 6.4). The results of the calculations are listed in Table 6.5. The results are compared to counting metrics in Table 6.6.

For Module 1, as seen in Figure 6.5, the edges, e_1, e_2, e_3, and e_6 ,are included in the calculation.

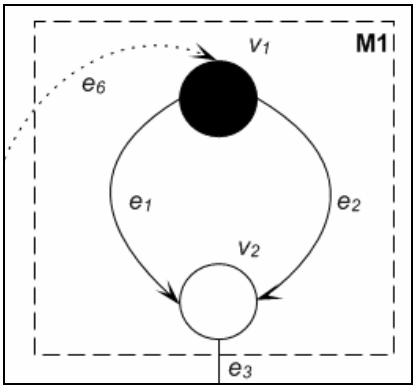


Figure 6.5 Module 1

The Interaction Complexity of Module 1 is

$$\chi_{m_1} = 0.4432+0.5285+0+0.4977=1.4694 .$$

For Module 2, as seen in Figure 6.6, the edges, e_3, e_4, e_5, e_6 and e_{15}, are included in the calculation.

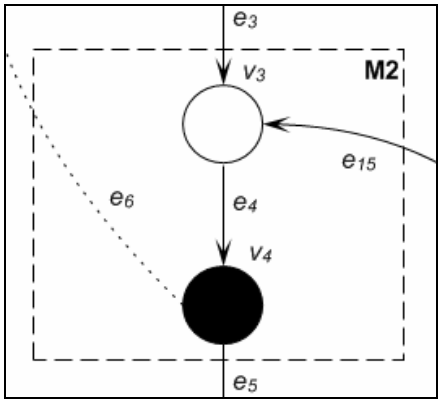


Figure 6.6 Module 2

The Interaction Complexity of Module 2 is

$$\chi_{m_2} = 0+0+0.5021+0.4977+0=1 .$$

For Module 3, as seen in Figure 6.7, the edges e_5, e_7, e_8, and e_9, are included in the calculation.

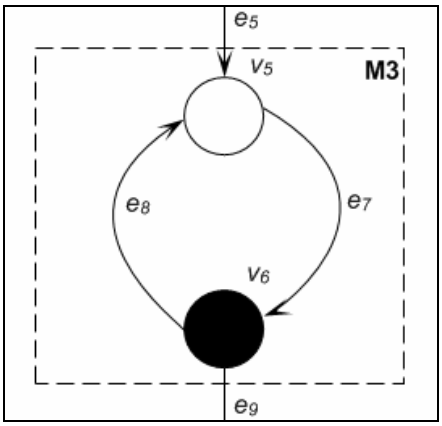


Figure 6.7 Module 3

The Interaction Complexity of Module 3 is

$$\chi_{m_3} = 0.5021+0+0.5183+0.4745=1.4949.$$

For Module 4, as seen in Figure 6.8, the edges, e_9, e_{10}, e_{11}, e_{12}, e_{13}, e_{14}, and e_{15}, are included in the calculation.

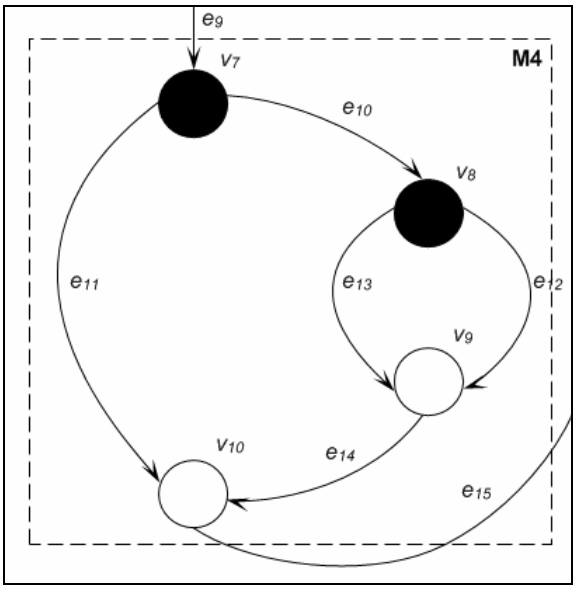


Figure 6.8 Module 4

The Interaction Complexity of Module 4 is

$$\chi_{m_4} = 0.4745+0.5+0.5+0.5183+0.4745+0=2.4673.$$

Table 6.5 Interaction Complexities of the Modules

Module	Interaction Complexity
M1	1.4694
M2	1
M3	1.4949
M4	2.4673

6.2.3 Interaction Cohesion (Ω)

Cohesion is one of the fundamental measures of structural complexity, as explained in Chapter 5. In this work, an information-theoretical approach is used to compute the cohesion of a software system. The measurement can be applied to all component-based systems without the need for source code. Briand and Morasca's (1997) work for measuring cohesion for component-based systems has been the main source for identifying the properties of cohesion. The properties of cohesion of a module are:

- Nonnegativity and Normalization. Cohesion of a module belongs to a specified interval, $Cohesion\ (mk|MS) \in [0, Max]$.
- Null Value. Cohesion of a module is null if its set of intramodule edges is empty.
- Monotonicity. Adding an intramodule edge to a module does not decrease its cohesion.
- Merging of modules. If two unrelated modules, m_1 and m_2, are merged to form a new module, $m_{1\cup 2}$, that replaces m_1 and m_2, then the module cohesion of $m_{1\cup 2}$ is not greater than the maximum of the module cohesion of m_1 and m_2.

In an intramodule edges-only graph representation of a software module, Interaction Cohesion (Ω) is the ratio of the entropy sum of the intramodule edges to the Interaction Complexity (χ) of the module

$$\text{Interaction Cohesion } (\Omega) = \frac{\text{Total entropy of intramodule edges}}{\text{Interaction complexity}}. \qquad (6.12)$$

This ratio concept is similar to the counting cohesion of a modular system (ratio of the number of intramodule edges to the total number of edges); however, the entropy measurement would provide more precise values compared to the classical counting approach.

In the following example, the Interaction Cohesion metric is calculated for each module of the CCFG given in Figure 6.2. The metric calculations are based on (6.12) and the edge entropies (listed in Table 6.4). The results of the calculations are listed in Table 6.7 in comparison to the counting metrics.

For Module 1, as seen in Figure 6.9, the intramodule edges, e_1 and e_2, are included in the calculation.

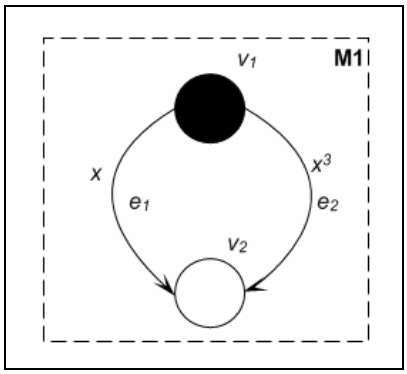


Figure 6.9 Intramodule edges of Module 1

The Interaction Cohesion of Module 1 is

$$\Omega_{M_1} = \frac{0.4432+0.5285}{1.4694} = 0.6805.$$

For Module 2, as seen in Figure 6.10, the only intramodule edge included in the calculation is e_4.

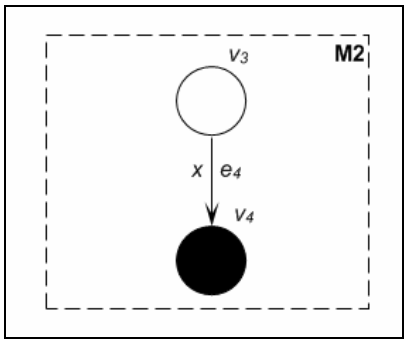


Figure 6.10 Intramodule edges of Module 2

The Interaction Cohesion of Module 2 is

$$\Omega_{M_2} = \frac{0}{1} = 0 .$$

For Module 3, as seen in Figure 6.11, the intramodule edges, e_7 and e_8, are included in the calculation.

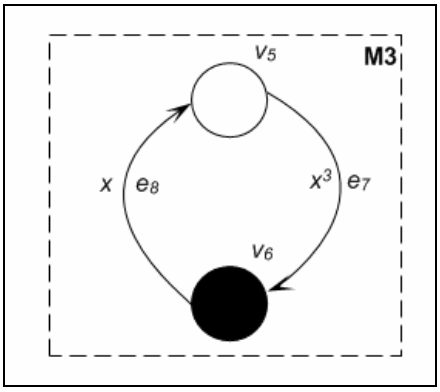


Figure 6.11 Intramodule edges of Module 3

The Interaction Cohesion of Module 3 is

$$\Omega_{M_3} = \frac{0+0.5183}{1.4949} = 0.3467 .$$

For Module 4, as seen in Figure 6.12, the intramodule edges, e_{10}, e_{11}, e_{12}, e_{13}, and e_{14}, are included in the calculation.

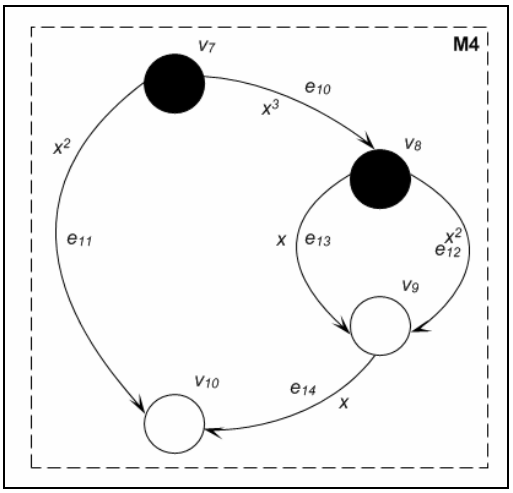


Figure 6.12 Intramodule edges of Module 4

The Interaction Cohesion of Module 4 is

$$\Omega_{M_4} = \frac{0.5+0.5+0.5183+0.4745+0}{2.4673} = 0.8106\,.$$

6.2.4 Interaction Coupling (ϒ)

Coupling is one of the fundamental measures of structural complexity, as explained in Chapter 5. In this work, an information-theoretical approach is taken to compute the coupling of a software system. The measurement can be applied to all component-based systems without the need for source code. Briand and Morasca's (1997) work for measuring coupling for component-based systems has been the main source for identifying the properties of coupling. The properties of coupling of a module are:

- Nonnegativity. Coupling of a module is nonnegative.

- Null Value. Coupling of a module is null if its set of intermodule edges is empty.
- Monotonicity. Adding an intermodule edge to a module does not decrease its coupling.
- Merging of modules. If two modules, m_1 and m_2, are merged to form a new module, $m_{1\cup 2}$, that replaces m_1 and m_2, then the module coupling of $m_{1\cup 2}$ is not greater than the sum of the module coupling of m_1 and m_2.
- Disjoint module additivity. If two modules, m_1 and m_2, which have no intermodule edges between nodes in m_1 and nodes in m_2, are merged to form a new module, $m_{1\cup 2}$,that replaces m_1 and m_2, then the module coupling of $m_{1\cup 2}$ is equal to the sum of the module coupling of m_1 and m_2.

In an intermodule edges-only graph representation of a software module, Interaction Coupling (Υ) is the ratio of the entropy sum of the intermodule edges to the Interaction Complexity (χ) of the module

$$\text{Interaction Coupling } (\Upsilon) = \frac{\text{Total entropy of intermodule edges}}{\text{Interaction complexity}}. \qquad (6.13)$$

This ratio concept is similar to the counting coupling of a modular system (ratio of the number of intermodule edges to the total number of edges); however, the entropy measurement would bring more precise values compared to the classical counting approach.

In the following example, the Interaction Coupling metric is calculated for each module of the CCFG given in Figure 6.2. The metric calculations are based on (6.13) and the edge entropies (listed in Table 6.4). The results of the calculations are listed in Table 6.8 in comparison to the counting metrics.

For Module 1, as seen in Figure 6.13, the intermodule edges, e_3 and e_6, are included in the calculation.

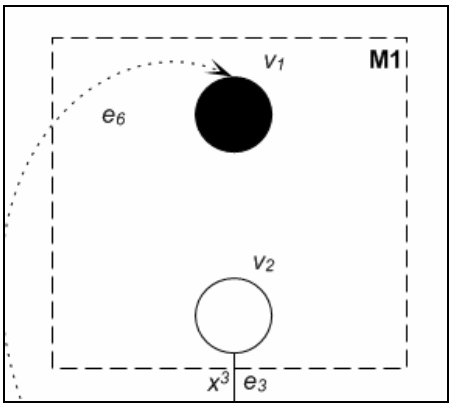


Figure 6.13 Intermodule edges of Module 1

The Interaction Coupling of Module 1 is

$$\Upsilon_{M_1} = \frac{0+0.4977}{1.4694} = 0.3387 .$$

For Module 2, as seen in Figure 6.14, the intermodule edges e_3, e_5, e_6, and e_{15} , are included in the calculation.

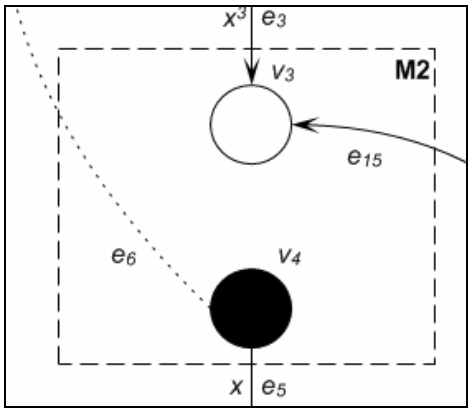


Figure 6.14 Intermodule edges of Module 2

The Interaction Coupling of Module 2 is

$$\Upsilon_{M_2} = \frac{0+0.5021+0.4977+0}{1} = 1.$$

For Module 3, as seen in Figure 6.15, the intermodule edges, e_5 and e_9, are included in the calculation.

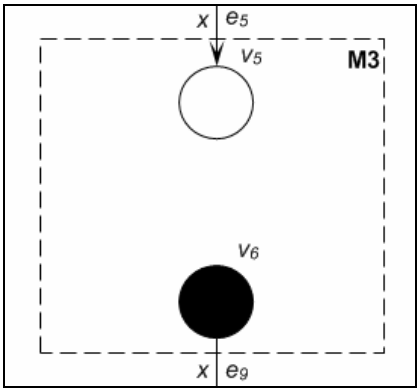


Figure 6.15 Intermodule edges of Module 3

The Interaction Coupling of Module 3 is

$$\Upsilon_{M_3} = \frac{0.5027+0.4745}{1.4949} = 0.6536.$$

For Module 4, as seen in Figure 6.16, the intermodule edges, e_9 and e_{15}, are included in the calculation.

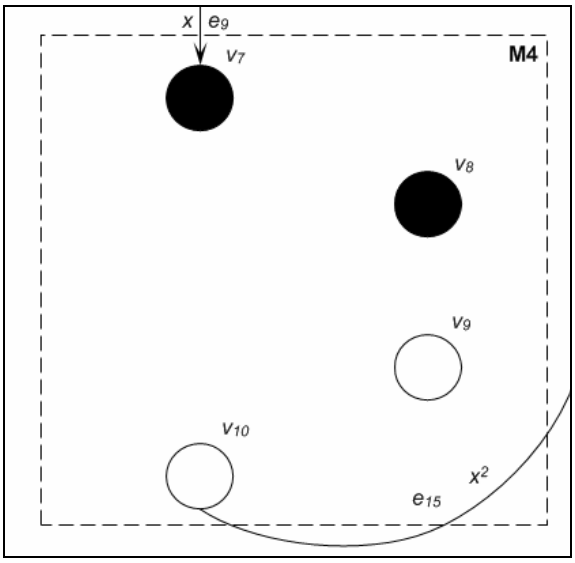


Figure 6.16 Intermodule edges of Module 4

The Interaction Coupling of Module 4 is

$$\Upsilon_{M_4} = \frac{0.4745+0}{2.4673} = 0.1917 .$$

6.3 Comparison of Information Theory-Based Metrics to Counting Metrics for Graphs

This section applies the metrics examined in the previous section: Design Information Content, Interaction Complexity, Interaction Cohesion, and Interaction Coupling, to traditional counting metrics that are comparable to them, graph size, complexity, cohesion, and coupling. The counting metrics are based on the number of edges and nodes in each module and do not consider the probabilities of the nodes or edges. By using stationary state probabilities at the nodes and branch probabilities at the edges, information theory metrics are based on the entropies of the nodes and edges of the CCFG. As explained in the section about the Design Information Content metric, the whole system needs to be taken into account, so it will be compared to the size metric of the whole system. In Tables 6.6, 6.7, and 6.8, the comparisons are made with the comparable entities

between counting and information theory metrics according to the modules. The Design Information Content of the system was calculated as 28.38 bits in the example with ten nodes.

Table 6.6 Comparison of the Interaction Complexity Metrics to the Counting Metrics

Modules	Interaction Complexity	# of edges (Counting)
M1	1.4694	4
M2	1	5
M3	1.4949	4
M4	2.4673	7

Table 6.6 includes a result that shows the precision of the Interaction Complexity metric over the counting metric related to the number of edges. Even though Module 2 has more edges than Module 1, the Interaction Complexity of Module 2 is smaller than Module 1. This result is caused by the uncertainty of the branches in the modules. The edges leaving the decision node in Module 1 have similar probabilities, leading to a higher uncertainty resulting in a higher Interaction Complexity. On the other hand, Module 2 edges have less uncertainty, leading to a simpler structure with less complexity despite having more edges. The highest complexity level occurs in Module 4 that has the largest number of edges and branching levels.

Table 6.7 Comparison of the Interaction Cohesion Metrics to the Counting Metrics

Modules	Interaction Cohesion	Counting metric of cohesion
M1	0.6805	0.5
M2	0	0.25
M3	0.3467	0.5
M4	0.8106	0.71

Table 6.7 lists the Interaction Cohesion metric values vs. the counting metrics of cohesion. The counting metrics are simply the ratio of intramodule edges to the total number of edges; however, the Interaction Cohesion metric gives more precise results based on the entropy of the intramodule edges.

Table 6.8 Comparison of the Interaction Coupling Metrics to the Counting Metrics

Modules	Interaction Coupling	Counting metric of coupling
M1	0.3387	0.5
M2	1	0.6
M3	0.6536	0.5
M4	0.1917	0.28

Table 6.8 lists the Interaction Coupling metric values vs. the counting metrics of cohesion. The counting metrics are the ratio of intermodule edges to the total number of edges. However the Interaction Cohesion metric again gives more precise results based on the entropy of the intermodule edges. The comparisons between the entropy-based metrics and counting metrics are discussed further in conclusion.

6.4 The Program Used in the Case Study

The program examined in the previous section has been implemented as a Java application with four class files. The program and its metrics are listed in Table 6.9.

Module 1:

```
public class M1 {
static int k=0;

        public static void V1V3(int i) {
          if(k>30)
                        return;
                System.out.println("V1");
                if (i % 2 == 0) {// V1 predicate
                        System.out.println("E1");// E1 Edge
                } else {
                        System.out.println("E2");// E2 Edge
                }
                System.out.println("V2");// V2 Node
                System.out.println("E3");// E3 Edge
                M2.V3V4(k);
        }

        public static void main(String[] args) {
                V1V3(0);
        }
}
```

Module 2:

```
public class M2 {

        public static void V3V4(int i){
                System.out.println("V3");// V3 Node
                System.out.println("E4");// E4 Edge
                System.out.println("V4");
                if (i % 8==0) {// V4 prdeicate
                        M1.k++;
                        i=M1.k;
                        M1.V1V3(i);
                } else{
                        M3.V5V6(i);
                }
        }
}
```

Module 3:

```
public class M3 {
        public static void V5V6(int i){
```

```
                System.out.println("E5");// E5 Edge
                int k = -1;
                while (true) {
                        k++;
                        System.out.println("V5");// V5 node
                        System.out.println("E7");// E7 node
                        if (k == 1) {
                                break;
                        } else {
                                System.out.println("E8");// E8 edge
                        }
                }
                System.out.println("V6");
                System.out.println("E9");
                M4.V7V10(i);
        }
}
```

Module 4:

```
public class M4 {
        public static void V7V10(int i) {
                System.out.println("V7");// V7 Node
                if (i % 2 == 0) {
                        System.out.println("e10"); // E10 edge
                        System.out.println("V8");// V8 Node
                        if (i % 3 == 0) {
                                System.out.println("e12");// e12 edge
                        } else {
                                System.out.println("e13");// e13 edge
                        }
                        System.out.println("V9");// V9 Node
                        System.out.println("E14");// E14 edge
                } else {
                        System.out.println("e11");// e11 edge
                }
                System.out.println("V10");// V10 node
                System.out.println("e15");// e15 edge
                M1.k++;
                i=M1.k;
                M2.V3V4(i);
        }
}
```

Table 6.9 The Results of Metric Analysis of the Modules by Analyst4j (2007)

Metric	Module 1	Module 2	Module 3	Module 4
Halstead volume of the file.	277.0	192.0	244.0	387.0
Halstead effort of the file.	3698.0	2032.0	3715.0	5612.0
Average cyclomatic complexity of a method in the file.	2.5	3.0	4.0	5.0
Average weighted method of a class in the file.	2.0	1.0	1.0	1.0
Average weighted method complexity of a class in the file.	5.0	3.0	4.0	5.0
Average response for class in the file.	3.0	3.0	2.0	2.0
Average lack of cohesion of methods of a class in the file.	1.0	0.0	0.0	0.0
Average coupling between objects of a class in the file.	2.0	3.0	2.0	3.0
Maintainability index of the file.	128.9	133.4	126.7	131.0
Average essential complexity of a method in the file.	1.5	1.0	4.0	1.0

Table 6.9 lists the results for the metric analysis performed by the code analyzing software Analyst4j. The analysis programs performing metric computations use source codes for analysis. This process usually includes a scanning of the code by a static code analysis. Static code analysis is performed without actually executing programs built from that software. The metrics defined within the program are calculated according to the results of the static code analysis, such as counting the number of number of input data parameters or input control parameters for measuring the coupling between modules.

The results listed in Table 6.12 are based on static code analysis, making the comparison of the results to the entropy information metrics less meaningful. In the entropy-based measurement framework, the dynamic factors that would lead to different results than a static analysis of the program are considered, such as execution time. The static code analysis tools would thus not provide comparable results to the framework to provide an empirical validation. There is another branch of code analysis called dynamic code analysis, which is the analysis of computer software through executing programs on a real or virtual processor. Such tools may require the loading of special libraries or even the recompilation of program code; however, current dynamic code analysis software does not support metric analysis to make comparisons to our results. Dynamic code analysis software has current uses, such as memory debugging, memory leak detection, and profiling, and its use may include software metrics in the near future. Entropy-based metrics may have potential uses in dynamic software analysis.

6.5 Validation

Kitchenham and others (1995) propose a validation framework for software measurement by defining a measurement structure model, measurement process, and five other models involved in measurement. The framework can help to understand how to validate a measure, how to assess the validation work of others, and when to apply a measure. According to their validation framework, there are two methods of testing the validity of measures:

- Theoretical validation. It confirms that the measurement does not violate any necessary properties of the elements of measurement or the definition models.

- Empirical validation. It corroborates that measured values of attributes are consistent with values predicted by models involving the attribute.

Theoretical methods of validation allow us to say that a measure is valid with respect to certain defined criteria; empirical methods provide corroborating evidence of validity or invalidity (Kitchenham and others 1995).

For the entropy-based framework, all of the metrics we defined are directly based on information theory and graph theory. Both areas are well established and the concepts that we used, entropy and CCFG, are already mathematically proven entities. As with most of the information theory-based metrics, entropy-based metrics also are valid measures according to theoretical methods; however, an empirical validation suite would prove useful in testing the application methods.

According to the criteria that Kitchenham established, it is possible to show that:

- Different modular systems may have different amounts of Interaction Complexity, Interaction Coupling, and Interaction Cohesion.
- Interaction Coupling represents relationships among modules.
- Interaction Coupling represents relationships inside the modules.
- Different modular systems are allowed to have the same value of Interaction Coupling and Interaction Cohesion.
- Interaction Cohesion and Coupling are compatible with the ratio scale type, and the measurement algorithm is a valid instrument for an information theory-based measure.
- Given a CCFG graph, the protocol for abstracting CIUs is unambiguous and self-consistent.

Designing a complete validation framework for measurement methods is a highly complex challenge. A partial empirical validation is offered in this chapter during the comparison of entropy-based metrics to counting metrics, and further research would validate the usefulness of the metrics.

6.6 Conclusion

According to the summary of the efforts to use entropy-based metrics, we reach to the following conclusion. There is a need for a measurement framework:

- To support quantitative decomposition and representation.
- To increase the applicability of entropy based metrics.
- To lead the way in developing measurement software, which uses entropy-based methods.

Our motivation to create an entropy-based measurement framework is based on several factors that emphasize its advantages over other measurement methods:

- Entropy metrics promise important improvements in our insights about design quality and design complexity (Bianchi and others 2001).
- Entropy metrics provide a quantitative means of measuring information flow in software systems (Masri and Podgurski 2006).
- Entropy metrics can be applied to CBS without requiring source code, based on previous results of information theory metrics.

The main contribution of this dissertation work is the creation of a dynamic entropy-based measurement framework supported by information theory that represents and decomposes complex software systems in a hierarchical structure with analytical methods.

The framework examines the structural complexity of the design by quantifying the interaction of components and modules. The end results of this framework are our entropy-based metrics, Design Information Content (δ), Interaction complexity (χ), Interaction Cohesion (Ω), and Interaction Coupling (Υ).

The concept of entropy, as a measure of information is used to compute the information content of a system and determine the structural complexity of the system by measuring the Interaction Complexity, Interaction Coupling and Cohesion of modular systems. All of the interactions are examined by using the edge structures and entropies in CCFG. The metrics are applied without using source code to CCFGs that are obtained from UML diagrams and flowcharts. To measure the communication among components, by the component definition, we use component interfaces to read information. The measured parameter of interest could be any value, such as execution time, composition ratio, or reuse ratio. In the case study used in this chapter, execution time is the parameter. The results of the case study are compared to the related counting metrics in Chapter 6. There are similar results in most cases; however, the higher precision of the entropy metrics is visible in the cases, in which the counting metrics give the same results due to the same number of edges. In these cases, entropy metrics provide more accurate values for the parameter of interest. It needs to be mentioned that the measurement procedure is independent of the measurement environment or subjective decisions of the designer, as the computation relies on solely quantitative information, and also the representation/decomposition process is based on graph theory.

6.7 Future Work

The recent work we completed in relation to software metrics focused on the application of software metrics to Content Management Systems (CMS) in a systematic way in order to provide quantitative and qualitative values to project managers (Aktunc and Tanik 2007). The metrics were used to measure system results, such as response times, user rankings, and content creators' satisfaction levels with the system. Measuring the success of the business processes using metrics allows the success of the project to be assessed and provides a basis for calculating an ROI. Metrics are also a valuable way to track the health of a CMS, and they help identify issues quickly and systematically for effective solutions. It is possible to represent the CMS the same way that we represented the software modules in this chapter; the application of entropy metrics to CMS is a viable option that may provide other perspectives to the project managers/designers.

As we mentioned in section 6.4, a common problem of metrics software is that it is based on static code analysis. Static code analysis directly supports software comprehension, and various methods already exist to perform static code analysis of software systems. However, a dynamic code analysis would provide better results regarding the dynamic behavior of the system, especially the logic flow and the interaction of components. Unfortunately, the current dynamic code analysis software does not support metric analysis to provide these results. Entropy-based metrics have potential for use in dynamic software analysis.

Another improvement to this work would be to unify the measurement framework with semantic methods. A graph-theoretic approach to represent a software system using semantic and structural information inferred from UML diagrams and designers' flow-

charts would follow our measurement approach. With this model, a number of analysis methods and tools could be used in conjunction to obtain various types of information about the software being analyzed (e.g., static structural information, dynamic information, and semantic information). The combined representation of these types of information would allow the user to define new measures and metrics that reflect different aspects of the software.

REFERENCES

Abd-El-Hafiz SK. 2001. Entropies as measures of software information. In: International conference on software maintenance; Florence, Italy: IEEE Computer Society. p 110-117.

Abran A, Ormandjieva O and Abu Talib M. 2004. Information theory-based functional complexity measures and functional size with COSMIC-FFP. In: 14th international workshop on software measurement IWSM-Metrikon 2004; Magdeburg, Germany. p 457-471.

Aczel J and Daroczy Z. 1975. On measures of information and their characterization. New York: Academic Press.

Aktunc O. 2002. The role of component technologies on enterprise engineering [Masters Thesis]. Birmingham, AL: University of Alabama at Birmingham.

Aktunc O and Tanik MM. 2007. A metrics approach to content management systems. In: The tenth world conference on integrated design and process technology; Antalya, Turkey. p 59-64.

Alagar VS, Ormandjieva O and Zheng M. 2000. Managing complexity in real-time reactive systems. In: The sixth IEEE international conference on engineering of complex computer systems; Tokyo, Japan. p 12-24.

Albrecht A. 1979. Measuring application development productivity. In: Proceedings of joint SHARE/GUIDE/IBM applications development symposium; Monterey, CA. p 83-92.

Albrecht A and Gaffney J. 1983. Software function, source lines of code, and development effort prediction: A software science validation. IEEE Transactions on Software Engineering. 9(6):639-648.

Allen EB, Gottipati S and Govindarajan R. 2007. Measuring size, complexity, and coupling of hypergraph abstractions of software: An information-theory approach. Software Quality Control. 15(2):179-212.

Allen TFH and Starr TB. 1982. Hierarchy: Perspectives for ecological complexity. Chicago: University of Chicago Press.

Analyst4j. 2007. A Codeswat company product. Code quality assessment for Java using software metrics. Available from: http://www.codeswat.com.

Baker AL, Beiman JM, Fenton N, Gustafson DA, Melton A and Whitty R. 1990. A philosophy of software measurement. Journal of System Software. 12(3): 277-281.

Basili VR. 1980. Qualitative software complexity models: A summary in tutorial on models and methods for software management and engineering; Los Alamitos, CA: IEEE Computer Society Press.

Basili VR, Caldiera G and Rombach HD. 1994. Goal question metric paradigm. In: Encyclopedia of Software Engineering: John Wiley & Sons. p 646-661.

Berander P and Jonsson P. 2006. A goal question metric based approach for efficient measurement framework definition. In: Proceedings of the 2006 ACM/IEEE international symposium on empirical software engineering; Rio de Janeiro, Brazil: ACM Press. p 316-325.

Berlinger E. 1980. An information theory based complexity measure. In: Proceedings of the national computer conference; Reston, VA: AFIPS Press. p 773-779.

Bianchi A, Caivano D, Lanubile F and Visaggio G. 2001. Evaluating software degradation through entropy. In: The seventh international software metrics symposium; London, England: IEEE Computer Society Press. p 210.

Boehm B. 1981. Software engineering economics. Englewood Cliffs, NJ: Prentice Hall.

Boehm BW, Brown JR and Lipow M. 1976. Quantitative evaluation of software quality. In: 2nd international conference on software engineering; San Francisco, CA: IEEE Computer Society Press. p 592-605.

Booch G, Maksimchuk RA, Engel MW, Young BJ, Conallen J and Houston KA. 2007. Object-oriented analysis and design with applications. 3rd ed.: Addison-Wesley Professional.

Briand LC and Morasca S. 1997. Software measurement and formal methods: A case study centered on TRIO+ specifications. In: Proceedings of the 1st international conference on formal engineering methods: IEEE Computer Society. p 315-326.

Brooks FP. 1987. No silver bullet: Essence and accidents of software engineering. The Computer Journal. 20(4):10-19.

Card DN and Glass RL. 1990. Measuring software design quality. Englewood Cliffs, NJ: Prentice Hall.

Cardoso AI, Araujo T and Crespo ZC. 2001. An alternative way to measure software. In: Proceeding of FESMA; Heidelberg, Germany. p 225-236.

Chapin N. 2002. Entropy-metric for systems with COTS software. In: The eighth IEEE symposium on software metrics; Ottawa, Canada: IEEE Computer Society. p 173-181.

Chidamber SR and Kemerer CF. 1991. Towards a metrics suite for object-oriented design. In: Conference on object-oriented programming systems languages and applications; Phoenix, Arizona: ACM. p 197-211.

Christensen K, Fitsos GP and Smith CP. 1981. A perspective on software science. IBM Systems Journal of System Software. 20(4):372-387.

Conte SD, Dunsmore HE and Shen VY. 1986. Software engineering metrics and models. Redwood City, CA: The Benjamin/Cummings.

Coulter N, Cooper RB and Solomon MK. 1987. Information-theoretic complexity of program specifications. The Computer Journal. 30(3):223-227.

Darcy DP, Kemerer CF, Slaughter SA and Tomayko JE. 2005. The structural complexity of software: An experimental test. IEEE Transactions on Software Engineering. 31(11):982-995.

Davis A. 1993. Identifying and measuring quality in software requirements specification. In: Proceedings of software metrics symposium; Los Alamitos, CA: IEEE CS Press. p 137-150.

Davis JS and LeBlanc RJ. 1988. A study of the applicability of complexity measures. IEEE Transactions on Software Engineering. 14(9):1366-1372.

DeMarco T. 1982. Controlling software projects: Management, measurement & estimation. New York: Yourdon Press.

Dumke R. 1998. An object-oriented software measurement and evaluation framework. In: CONQUEST 98; Nurenberg, Germany. p 52-61.

Edwards WR. 2003. Outline of the analysis of software structure and information: The Center for Advanced Computer Studies. University of Louisiana at Lafayette.

Ejiogu L. 1991. A systematic methodology for software metrics. ACM SIGPLAN Notices. 26(1):124-132.

Elshoff JL. 1976. Measuring commercial PL/I programs using Halstead's criteria. ACM SIGPLAN Notices. 11(5):38-46.

Evangelist M. 1983. Software complexity metric sensitivity to program structuring rules. Journal of Systems and Software. 3(3):231-243.

Fenton NE. 1991. Software metrics: A rigorous approach. London, UK: Chapman & Hall.

Fenton NE. 1994. Software measurement: A necessary scientific basis. IEEE Transactions on Software Engineering. 20(3):199-206.

Fenton NE and Pfleeger SL. 1998. Software metrics: A rigorous and practical approach, revised. London: International Thomson Computer Press.

Florac W. 1992. Software quality measurement: A framework for counting problems and defects. CMU/SEI-92-TR-22. Carnegie Mellon University. Software Engineering Institute. Pittsburgh, PA.

Ford G. 1993. Lecture notes on engineering measurement for software engineers. CMU/SEI-93-EM-9. Carnegie Mellon University. Software Engineering Institute. Pittsburgh, PA.

Goodman P. 1993. Practical implementation of software metrics. London: McGraw Hill.

Gorla N and Ramakrishnan R. 1997. Effects of software structure attributes using software development productivity. Journal of Systems and Software. 36(2):191-199.

Grady RB and Caswell DL. 1987. Software metrics: Establishing a company-wide program. Englewood Cliffs, NJ: Prentice-Hall.

Halstead MH. 1977. Elements of software science. New York: Elsevier North-Holland.

Harary F. 1972. Graph theory. Reading, MA: Addison-Wesley.

Harrison R, Counsell SJ and Nithi RV. 1998. An evaluation of the MOOD set of object-oriented software metrics. IEEE Transactions on Software Engineering. 24(6):491-496.

Harrison W. 1992. An entropy-based measure of software complexity. IEEE Transactions on Software Engineering. 18(11):1025-1029.

Harrison W, Magel K, Kluczny R and DeKock A. 1982. Applying software complexity metrics to program maintenance. The Computer Journal. 15(9):65-79.

Heinemann GT, Loyall J and Schantz R. 2004. Component technology and QoS management. In: Component-based software engineering: 7th international symposium; Edinburgh, UK: Springer. p 249-263.

Henderson-Sellers B. 1996. Object-oriented metrics–measures of complexity. New Jersey: Prentice Hall.

IEEE. 1992. IEEE standard for a software quality metrics methodology. IEEE Std 1061-1992. Available from: http://standards.ieee.org/reading/ieee/std_public/description/se/1061-1992_desc.html.

ISO. 1991. International Standard ISO/IEC 9126. Information technology–software product evaluation–quality characteristics and guidelines for their use. International Organization for Standardization. International Electrotechnical Commission, Geneva.

Jones TC. 1986. Programming productivity. New York: McGraw Hill.

Kafura D and Henry S. 1987. The evaluation of software systems' structure using quantitative software metrics. Software–practice and experience. 14(6):561-573.

Kearney KJ, Sedmeyer RL, Thompson WB, Gray MA and Adler MA. 1986. Software complexity measurement. Communications of the ACM. 29(11).

Kelvin WT. 1891. Popular lectures and addresses. London, New York: Macmillan and Co.

Khoshgoftaar TM and Allen EB. 1994. Applications of information theory to software engineering measurement. Software Quality Control. 3(2):79-103.

Kitchenham BA, Pfleeger SL and Fenton NE. 1995. Towards a framework for software measurement validation. IEEE Transactions on Software Engineering. 21(12):929-944.

Korner J. 1971. Coding of an information source having ambiguous alphabet and the entropy of graphs. In: Transactions of the 6th Prague conference on information theory; Prague: Academia. p 411-425.

Krusko A. 2004. Complexity analysis of real-time software–using software complexity metrics to improve the quality of real-time software [Masters Thesis]. Stockholm: Royal Institute of Technology.

Lehman M and Belady LA. 1985. Program evolution–processes of software change. London: Academic Press.

Li HF and Cheung WK. 1987. An empirical study of software metrics. IEEE Transactions on Software Engineering. 13(6):697-708.

Lorenz M and Kidd J. 1994. Object-oriented software metrics. Upper Saddle River, New Jersey: Prentice Hall.

Malton A, Cordy JR, Cousineau D, Schneider KA, Dean TR and Reynolds J. 2001. Processing software source text in automated design recovery and transformation. In: IEEE 9th international workshop on program comprehension; Toronto, Canada: IEEE. p 127-134.

Masri W and Podgurski A. 2006. An empirical study of the strength of information flows in programs. In: 2006 international workshop on dynamic systems analysis; Shanghai, China: ACM Press. p 73-80.

Mathematica. A Wolfram company product. Version 4.1. Available from: http://www.wolfram.com/products/mathematica/index.html

Maxwell TT, Ertas A and Tanik MM. 2002. Harnessing complexity in design. Transactions of SDPS: Journal of Integrated Design and Process Science. 6(3):63-74.

McCabe T. 1976. A complexity measure. IEEE Transactions of Software Engineering. SE-2(4):308-320.

McCall JA, Richards PK and Walters GF. 1977. Factors in software quality. Technical report. Accession no: ADA049014: National Technical Information Service; Sunnyvale, CA.

Mills E. 1988. Software metrics. SEI curriculum module SEI-CM-12-1.1. Carnegie Mellon University. Software Engineering Institute. Pittsburgh, PA.

Morowitz H. 1995. The emergence of everything. The Complexity Journal. 1(4).

Mowshowitz A. 1968. Entropy and the complexity of graphs: I. An index of relative complexity of a graph. Bulletin of Mathematical Biophysics. 30:175-204.

Munson JC. 2003. Software engineering measurement. Boca Raton, Florida: CRC Press.

Musa JD. 1975. A theory of software reliability and its application. IEEE Transactions on Software Engineering. 1(3):312-327.

Nystedt S and Sandros C. 1999. Software complexity and project performance [Masters Thesis]. University of Gothenburg.

Ohlsson N. 1996. Software quality engineering by early identification of fault prone modules [Dissertation]. Linköping: Linköping University.

Pandian R. 2004. Software metrics. Boca Raton: CRC Press.

Papoulis A and Pillai SU. 2002. Probability, random variables and stochastic processes. New York: McGraw Hill Higher Education.

Parnas DL. 1972. On the criteria to be used in decomposing systems into modules. Communications of the ACM. 15(12):1053-1058.

Parnas DL. 1975. The influence of software structure on reliability. In: International conference on reliable software; Los Angeles, CA. p 358-362.

Podgorelec V and Hericko M. 2007. Estimating software complexity from UML models. ACM Press. 32(2).

Poulin J. 2001. Measurement and metrics for software components. In: Component based software engineering–Putting the pieces together: Addison-Wesley.

Prather RE. 1995. Design and analysis of hierarchical software metrics. CM Computing Surveys. 27(4):497-518.

Prather RE. 1996. Convexity and independence in software metric theory. Software Engineering Journal. 11(4):238-246.

Pressman RS. 2007. Software engineering: A practitioner's approach. 5th ed. New York: McGraw Hill.

Riguzzi F. 1996. A survey of software metrics. Report 96-010. Bologna, Italy. Available from: http://citeseer.ist.psu.edu/550098.html.

Roberts F. 1979. Measurement theory with applications to decisionmaking, utility, and the social sciences. Addison-Wesley.

Rubey RJ and Hartwick RD. 1968. Quantitative measurement program quality. In: Proceedings of the 23rd ACM national conference; New York: ACM. p 671-677.

Rubin HA. 1987. A comparison of software cost estimation tools. System Development. 7(5):1-3.

Ruston H. 1979. Complexity and cost: An assessment of the state of the art. In: The workshop on quantitative software models for reliability; New York. p 152-158.

Schneidewind NF. 1977. Modularity considerations in real-time operating structures. In: IEEE computer software and applications conference COMPSAC; Chicago. p 397-403.

Sedigh-Ali S, Ghafoor A and Paul RA. 2001. Software engineering metrics for COTS-based systems. The Computer Journal. 34(5):44-50.

Seker R. 2002. Component-based software modeling based on Shannon's information channels [Dissertation]. Birmingham, AL: University of Alabama at Birmingham.

Seker R, Aktunc O, Ozaydin B, Tanik MM and Jololian L. 2004. Pervasive Shannon metrics and component-based software. International Journal of Computer and Information Science. 5(11). p 62-73.

Seker R and Tanik MM. 2004. An information-theoretical framework for modeling component-based systems. IEEE Transactions on Systems, Man, and Cybernetics. 34(4):475-484.

Shannon CE. 1949. A mathematical theory of communication. The Bell System Technical Journal. 27:379-423.

Shepperd M. 1988. A critique of cyclomatic complexity as a software metric. Software Engineering Journal. 3(2):30-36.

Simon H. 1969. The sciences of the artificial. Cambridge, Massachusetts: MIT Press.

Sommerville I. 2007. Software engineering. 8th ed. Essex, England: Addison-Wesley.

Stevens WP, Myers GJ and Constantine LL. 1976. Structural design. IBM Systems Journal of System Software. 13(2):113-129.

Szyperski C. 1998. Component software: Beyond object-oriented programming. New York: ACM Press/Addison-Wesley.

Tang Y. 1999. A methodology for component-based system integration [Dissertation]. New Jersey Institute of Technology.

Tang Y, Dogru AH, Kurfess FJ and Tanik MM. 2001. Computing cyclomatic complexity with cubic graphs. Journal of Systems Integration. 10(4):395-409.

Tausworthe RC. 1981. Deep space network software cost estimation model. The telecommunications and data acquisition progress report, TDA PR 42-61. Pasadena, CA. p 39-57.

Thebaut SM and Shen VY. 1984. An analytic resource model for large-scale software development. Information Processing and Management. 1(2):293-315.

Torres WR and Samadzadeh MH. 1991. Software reuse and information theory based metrics. IEEE Transactions on Software Engineering. p 437-446. Available from: http://ieeexplore.ieee.org/iel2/332/3857/00143916.pdf?tp=&isnumber=&arnumber=143916.

USAF. 1987. Management quality insight. Report AFSCP800-14. Air Force Systems Command, Andrews AFB. Washington, D.C.

USC. 2001. Software metrics guide. In: CSCI 577b: Software engineering II class notes. Available from: http://sunset.usc.edu/classes/cs577b_2001/metricsguide/metrics.html.

Van Emden MH. 1971. An analysis of complexity [Dissertation]. Amsterdam: Mathematisches Zentrum.

Walston CE and Felix CP. 1977. A method of programming measurement and estimation. IBM Systems Journal. 16(1):54-73.

Warshall S. 1962. A Theorem on Boolean Matrices. Journal of the ACM. 9(1):11-12.

Whitmire S. 1997. Object-oriented design measurement. John Wiley and Sons.

Wolverton RW. 1974. The cost of developing large-scale software. IEEE Transactions on Computer. C-23(6):615-636.

Yau SS and Collofello JS. 1980. Some stability measures for software maintenance. IEEE Transactions on Software Engineering. 6(6):545-552.

Yourdon E and Constantine L. 1979. Structured design: Fundamentals of a discipline of computer programming and design. Prentice Hall.

Zuse H. 1990. Software complexity measures and methods. Walter de Gruyter, New York.

Zuse H. 1997. A framework of software measurement. Walter de Gruyter, New York.

Zweben SH and Halstead MH. 1979. The frequency distribution of operators in PL/I programs. IEEE Transactions on Software Engineering. 5(2): 91-95.

APPENDIX

EMPIRICAL VALIDATION

In this section, the program written for Mathematica (2007) to calculate the eigenvectors is given with the outputs. Another example, following the methods in Chapter 6, is also given as a validation to the framework.

The example given in Chapter 6 has the weighted adjacency matrix

$$P(x)=\begin{bmatrix} 0 & x+x^3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & x^3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 0 & 0 \\ x & 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & x^3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & x & 0 & x & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & x^3 & 0 & x^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x+x^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x \\ 0 & 0 & x^2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

The Mathematica program used to calculate the positive unique root of the matrix is given below.

$$A = \begin{Bmatrix} 0 & x+x^3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & x^3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 0 & 0 \\ x & 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & x^3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & x & 0 & x & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & x^3 & 0 & x^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x+x^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x \\ 0 & 0 & x^2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{Bmatrix}$$

```
R[x_] := Det[IdentityMatrix[First[Dimensions[A]]] - A];
Print["det[1-A]=", R[x]];
RS = x /. NSolve[R[x] == 0];
Print[RS]
{{0, 1.36887, 0, 0, 0, 0, 0, 0, 0, 0},
 {0, 0, 0.5497, 0, 0, 0, 0, 0, 0, 0},
 {0, 0, 0, 0.819173, 0, 0, 0, 0, 0, 0},
 {0.819173, 0, 0, 0, 0.819173, 0, 0, 0, 0, 0},
 {0, 0, 0, 0, 0, 0.5497, 0, 0, 0, 0},
 {0, 0, 0, 0, 0.819173, 0, 0.819173, 0, 0, 0},
 {0, 0, 0, 0, 0, 0, 0, 0.5497, 0, 0.671044},
 {0, 0, 0, 0, 0, 0, 0, 0, 1.49022, 0},
 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0.819173},
 {0, 0, 0.671044, 0, 0, 0, 0, 0, 0, 0}}
det[1-A]=2.47542 × 10^-8
0.819173
```

The left and right eigenvectors of the weighted adjacency matrix are calculated using the Mathematica program below. The right eigenvector is referred in general as eigenvector. The left eigenvector is the eigenvector of the transpose of the matrix.

```
C = Eigenvectors[Transpose[A]]
D = Eigenvectors[A]
x = 0.81917251
Print[D]
Print[C]
```

The left and right eigenvectors of this matrix were already listed in Chapter 6.

Validation of the framework with another example that has 6 nodes

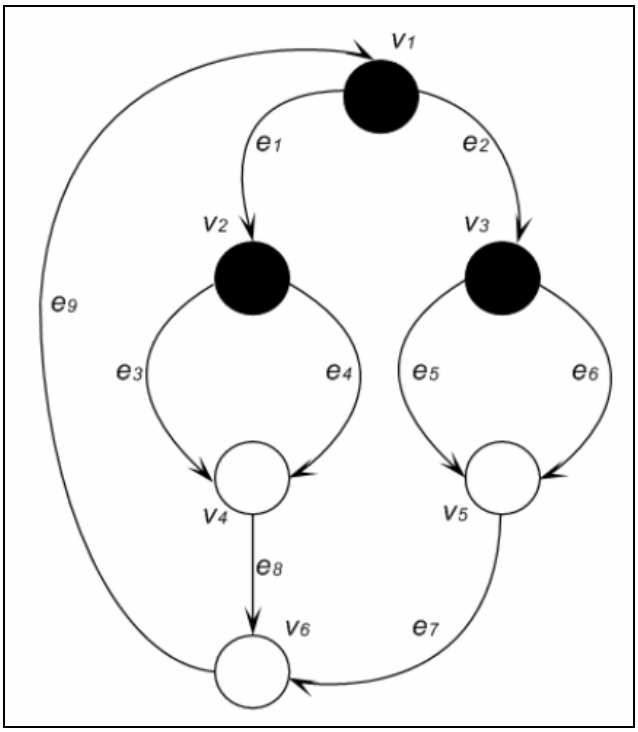


Figure A.1 A CCFG with 6 Nodes

For the CCFG in Figure A.1, the edge weights are listed in Table A.1.

Table A.1 Edge Weights

Edges	Weights
e_1	x^2
e_2	x^3
e_3	x
e_4	x
e_5	x
e_6	x^4
e_7	x^2
e_8	x
e_9	x^2

The weighted adjacency matrix for this CCFG is

$$P(x)=\begin{bmatrix} 0 & x^2 & x^3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2x & 0 & x^2 \\ 0 & 0 & 0 & 0 & x & 0 \\ 0 & x & 0 & 0 & 0 & x^2 \\ 0 & 0 & x^4 & 0 & 0 & x \\ x^2 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

It is possible to decompose the CCFG into its prime components. The decomposed CIUs are illustrated in Figure A.2.

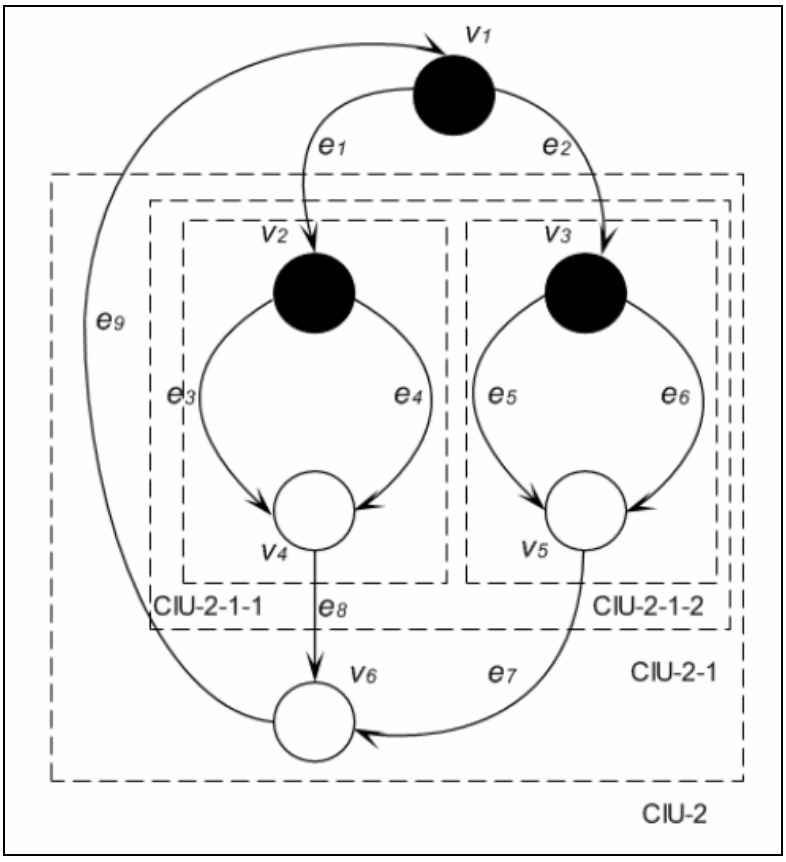


Figure A.2 The CIUs of the CCFG

The weighted adjacency matrix representing the CCFG is constructed in Mathematica

$$\mathbf{B} = \begin{pmatrix} 0 & x^2 & x^3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2x & 0 & x^2 \\ 0 & 0 & 0 & 0 & x & 0 \\ 0 & x & 0 & 0 & 0 & x^2 \\ 0 & 0 & x^4 & 0 & 0 & x \\ x^2 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

The unique root of this matrix is $x = 0.643643$. The left and right eigenvectors of the adjacency matrix are

$$\xi = \begin{bmatrix} 0.2383 \\ 0.5759 \\ 0.0714 \\ 0.7414 \\ 0.0459 \\ 0.5753 \end{bmatrix} \quad \eta = \begin{bmatrix} 0.3288 \\ 0.7529 \\ 0.0634 \\ 0.5410 \\ 0.0985 \\ 0.1362 \end{bmatrix}.$$

The node and edge probabilities are calculated using the eigenvectors above and listed in Tables A.2 and A.3.

Table A.2 Node Probabilities for the CCFG

Node	Node Probability
v_1	0.0783
v_2	0.4335
v_3	0.0045
v_4	0.4010
v_5	0.0045
v_6	0.0783

Table A.3 Edge Probabilities for the CCFG

Edge	Edge Probability
e_1	0.9486
e_2	0.0514
e_3	0.8955
e_4	0.1044
e_5	0.9999
e_6	0.0001
e_7	1
e_8	1
e_9	1

The entropies of the nodes and edges are calculated using Shannon's Entropy and listed in Table A.4 and A.5.

Table A.4 Node Entropies for the CCFG

Node	p_{v_i}	$\log_2 p_i$	H_{v_i}
v_1	0.0783	3.6748	0.2877
v_2	0.4335	1.2058	0.5227
v_3	0.0045	4.4739	0.0201
v_4	0.4010	1.3183	0.5286
v_5	0.0045	7.7958	0.0350
v_6	0.0783	3.6748	0.2877

Table A.5 Edge Entropies for the CCFG

Edge	p_{v_i}	$\log_2 p_i$	H_{v_i}
e_1	0.9486	0.0761	0.0721
e_2	0.0514	4.2820	0.2200
e_3	0.8955	0.1592	0.1425
e_4	0.1044	3.2598	0.3403
e_5	0.9999	0.0001	0.00001
e_6	0.0001	13.2877	0.0013
e_7	1	0	0
e_8	1	0	0
e_9	1	0	0

The entropy-based metrics for this example are computed in the following section.

1. Design Information Content (δ)

The total node entropy is

$$H_{v_{1N}} = H_{v_1} + H_{v_2} + ... + H_{v_{10}} = 0.2877 + 0.5227 + ... + 0.2877$$

$$H_{v_{1N}} = 1.6818 \text{ bits.}$$

Based on (6.9), the Design Information Content is

$$\delta = 1.6818 \cdot 6 \text{ nodes} = 10.09 \text{ bits.}$$

In the following section the Interaction Complexity, Interaction Cohesion, and Interaction Coupling metrics are computed for the four CIUs, CIU-2, CIU-2-2, CIU-2-2-1, and CIU 2-2-2.

2. Interaction Complexity (χ)

The edges that are included for the computation are

CIU-2. $e_1, e_2, \ldots, e_9$

CIU-2-1. $e_1, e_2, \ldots, e_8$

CIU-2-1-1. e_1, e_3, e_4, e_8

CIU-2-1-2. e_2, e_5, e_6, e_7.

The Interaction Complexities of the modules based on (6.11) are

$$\chi_{CIU-2} = 0.7762$$

$$\chi_{CIU-2-1} = 0.7762$$

$$\chi_{CIU-2-1-1} = 0.5549$$

$$\chi_{CIU-2-1-2} = 0.2213.$$

3. Interaction Cohesion (Ω)

The edges that are included for the computation are

CIU-2. e_3, e_4, e_5, e_6, e_7, e_8

CIU-2-1. e_3, e_4, e_5, e_6

CIU-2-1-1. e_3, e_4

CIU-2-1-2. e_5, e_6.

The Interaction Cohesion of the modules based on (6.12) are

$$\Omega_{CIU-2} = \frac{0.4841}{0.7762} = 0.6237$$

$$\Omega_{CIU-2-1} = \frac{0.4841}{0.7762} = 0.6237 .$$

Note that the edges going into the node *v6* do not have any uncertainty; thus the values are same for CIU-2 and CIU-2-1

$$\Omega_{CIU-2-1-1} = \frac{0.4828}{0.5549} = 0.8700$$

$$\Omega_{CIU-2-1-2} = \frac{0.0013}{0.2213} = 0.059 .$$

4. Interaction Coupling (Υ)

The edges that are included for the computation are

CIU-2: e_3, e_4, e_5, e_6, e_7, e_8

CIU-2-1: e_3, e_4, e_5, e_6

CIU-2-1-1: e_3, e_4

CIU-2-1-2: e_5, e_6.

The Interaction Coupling of the modules based on (6.13) are

$$\Omega_{CIU-2} = \frac{0.2921}{0.7762} = 0.3763$$

$$\Omega_{CIU-2-1} = \frac{0.2921}{0.7762} = 0.3763$$

$$\Omega_{CIU-2-1-1} = \frac{0.0721}{0.5549} = 0.1299$$

$$\Omega_{CIU-2-1-2} = \frac{0.22}{0.2213} = 0.9941.$$

Comparison of the results to the counting metrics is given in Tables A.6 through A.8.

Table A.6 Comparison of the Interaction Complexity Metrics to the Counting Metrics

Modules	Interaction Complexity	# of Edges (Counting)
CIU-2	0.7762	9
CIU-2-1	0.7762	8
CIU-2-1-1	0.5549	4
CIU-2-1-2	0.2213	4

Table A.7 Comparison of the Interaction Cohesion Metrics to the Counting Metrics

Modules	Interaction Cohesion	Counting Metric of Cohesion
CIU-2	0.6237	0.66
CIU-2-1	0.6237	0.5
CIU-2-1-1	0.8700	0.5
CIU-2-1-2	0.0590	0.5

Table A.8 Comparison of the Interaction Coupling Metrics to the Counting Metrics

Modules	Interaction Coupling	Counting Metric of Coupling
CIU-2	0.3763	0.22
CIU-2-1	0.3763	0.5
CIU-2-1-1	0.1299	0.5
CIU-2-1-2	0.9941	0.5