New Jersey Institute of Technology
Digital Commons @ NJIT

Dissertations | Theses and Dissertations

Fall 1998

A methodology for component-based system integration

Yongming Tang
New Jersey Institute of Technology

Follow this and additional works at: https://digitalcommons.njit.edu/dissertations

Part of the Computer Sciences Commons

Recommended Citation
Tang, Yongming, "A methodology for component-based system integration" (1998). *Dissertations*. 991.
https://digitalcommons.njit.edu/dissertations/991

This Dissertation is brought to you for free and open access by the Theses and Dissertations at Digital Commons @ NJIT. It has been accepted for inclusion in Dissertations by an authorized administrator of Digital Commons @ NJIT. For more information, please contact digitalcommons@njit.edu.

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

NJIT New Jersey's Science & Technology University

The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.

ABSTRACT

A METHODOLOGY FOR COMPONENT-BASED SYSTEM INTEGRATION

by
Yongming Tang

Component-based software based on software architectures is emerging to be the next generation software development paradigm. The paradigm shifts the development focus from lines-of-codes to coarser-grained components and the interconnections among them. It consists of system architecture design, architecture description, component search and system integration from components to generate a software system.

However, one of the bottlenecks in this paradigm is the integration of the individual components into the overall system. In this dissertation a methodology for component-based system integration is proposed. It is based on an architectural aggregation view, a component model, flowgraphs and cyclomatic complexity. We introduce this view, model, and new ways to compute cyclomatic complexity based on flowgraphs.

The methodology makes use of Jackson diagram to represent the detailed design of a system and decomposes the system into components and aggregations. An aggregation is a set of components glued together by one connector, and is represented as a flowgraph. Then an aggregation flowgraph is decomposed into prime flowgraphs called prime connections. An Implementation Description Language (IDL) is introduced to represent the aggregations and components. Finally a system synthesis mechanism is proposed that is responsible for translating prime connections, embedding functional units into them, and composing aggregations and the integrated system from them.

A METHODOLOGY FOR COMPONENT-BASED SYSTEM INTEGRATION

by
Yongming Tang

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

Department of Computer and Information Science

January 1999

Copyright © 1999 by Yongming Tang
ALL RIGHTS RESERVED

APPROVAL PAGE

A METHODOLOGY FOR COMPONENT-BASED SYSTEM INTEGRATION

Yongming Tang

Dr. Murat M. Tanik, Dissertation Advisor — Date
Professor of Electrical and Computer Engineering, UAB, Alabama

---

Dr. Franz J. Kurfess, Dissertation Co-advisor — Date
Assistant Professor of Computer and Information Science, NJIT

Dr. Dao Chuang Hung, Committee Member — Date
Associate Professor of Computer and Information Science, NJIT

Dr. Donald H. Sebastian, Committee Member — Date
Professor of Industrial and Manufacturing Engineering, NJIT

Dr. Ali H. Dogru, Committee Member — Date
Associate Professor of Computer Science, METU, Turkey

BIOGRAPHICAL SKETCH

Author: Yongming Tang

Degree: Doctor of Philosophy

Date: January 1999

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1999

- Master of Science in Computer Science,
  Northwest University, Xi'an, P. R. China, 1987

- Bachelor of Science in Computer Science,
  Northwest University, Xi'an, P. R. China, 1982

Major: Computer Science

Presentations and Publications:

Yongming Tang, Ali H. Dogru, and Murat M. Tanik,
"Cyclomatic Complexity Based on Cubic Flowgraphs,"
Proceedings of the Third Conference on Integrated Design and Process Technology, Berlin, Germany, IDPT Vol. 4, pp. 82-85, July 6-9, 1998.

Yongming Tang and Hua Hua,
"Hypermedia-based Structured Modeling,"
Presented as a Poster to the Seventh ACM Conference on Hypertext, Washington DC, March 16-20, 1996.

Yongming Tang,
"PRAPS: A Problem Representation System for Automatic Programming,"
Journal of Northwest University, Xi'an, P. R. China, Vol. 24, No. 3, pp. 203-206, 1994.

To my beloved family

ACKNOWLEDGMENT

I owe a deep gratitude to my dissertation advisor, Dr. Murat M. Tanik, and my dissertation co-advisor, Dr. Franz J. Kurfess, for their academic support. They are not only an endless source of information, insights and intuition on research issues, but also a source of encouragement, confidence and reassurance. I would never have made it without them. Special thanks are given to Dr. Ali H. Dogru, for his invaluable assistance, insights and cooperation on earlier work of the dissertation.

I would like to thank the other members of my dissertation committee, Dr. Dao Chuang Hung, Dr. Donald H. Sebastian and Dr. Peter Ng, for actively participating in my committee. All of them have given me bright insights and excellent comments on an earlier draft of the dissertation.

I would also like to give my heartfelt thanks to my wife, Meihua Chen, and my daughter, Di Tang. It would have been impossible for me to complete this dissertation without their support and patience. I am especially thankful to them for understanding and tolerating my unusual working way these years.

TABLE OF CONTENTS

Chapter **Page**

1 INTRODUCTION 1

1.1 Objective 2

1.2 A Blueprint of the Methodology 3

1.3 An Outline of the Dissertation 5

1.4 Summary 8

2 FLOWGRAPHS AND CUBIC GRAPHS 9

2.1 Flowgraphs 9

2.1.1 Program Flowcharts 10

2.1.2 Program Flowgraphs 10

2.1.3 Structured Flowgraphs 12

2.1.4 Regular Expressions for Flowgraphs 15

2.2 Cubic Graphs 17

2.2.1 Definitions 18

2.2.2 Properties 19

2.2.3 Decomposition and Composition of Cubic Graphs 20

2.2.4 Prime Cubic Graphs 21

2.3 Cubic Flowgraphs 22

2.3.1 The Decomposition and Composition of Cubic Flowgraphs 23

2.4 Summary 27

3 SOFTWARE ARCHITECTURES 29

3.1 Definitions 29

TABLE OF CONTENTS
(Continued)

Chapter **Page**

3.2 Software Architectural Styles 34

3.3 Architectural Views 35

3.4 Architecture Description Languages 38

3.4.1 Requirements for ADLs 39

3.4.2 Classification Framework for ADLs 40

3.5 Summary 42

4 COMPONENT-BASED SOFTWARE 44

4.1 What Is a Component? 44

4.2 Requirements on Components in Software Libraries 45

4.3 Component Representation and Retrieval 47

4.4 Component-Based Development 48

4.5 Evaluation, Analysis and Management of Component-based Software 51

4.5.1 What Problems Need to Be Solved? 51

4.5.2 Some Proposed Solutions 54

4.6 Summary 55

5 CYCLOMATIC COMPLEXITY AND CUBIC FLOWGRAPHS 57

5.1 Cyclomatic Complexity of Programs 58

5.2 Computing Cyclomatic Complexity Based on Cubic Flowgraphs 61

5.3 Computing Cyclomatic Complexity Based on Decomposition of Cubic Flowgraphs 64

5.4 Summary 67

TABLE OF CONTENTS
(Continued)

Chapter **Page**

6 FUNDAMENTALS FOR THE METHODOLOGY 68

6.1 An Aggregation View on Software Architectures 68

6.1.1 Aggregation View 69

6.1.2 Demonstration by Examples 72

6.2 Component Model 78

6.3 Summary 80

7 SYSTEM REPRESENTATION AND DECOMPOSITION 83

7.1 Conversion of Jackson Diagrams to Flowgraphs 84

7.2 Aggregation Flowgraph Decomposition 88

7.3 Applications 90

7.3.1 The KWIC System 90

7.3.1.1 Solution 1 - Pipes and Filters 90

7.3.1.2 Solution 2 - Main Program/Subroutine 92

7.3.2 The ECC System 95

7.4 Summary 100

8 SYSTEM IMPLEMENTATION DESCRIPTION 102

8.1 Prime Connections 102

8.2 Implementation Description Framework 105

8.3 Implementation Description Language 108

8.4 Summary 122

TABLE OF CONTENTS
(Continued)

Chapter **Page**

9 SYSTEM INTEGRATION 124

9.1 An Outline of System Integration 125

9.2 Preliminary Issues 129

9.2.1 Parsing IDL Descriptions 129

9.2.2 Creating the System Table 130

9.2.3 Creating the Aggregation Table 132

9.2.4 Retrieving Components and Creating Component Tables 133

9.3 System Synthesis 134

9.4 Summary 140

10 CONCLUSION AND FUTURE WORK 142

10.1 Summary of Results 143

10.1.1 Cyclomatic Complexity 143

10.1.2 Software Architectures as Aggregations 144

10.1.3 Component Model 145

10.1.4 System Representation and Decomposition 146

10.1.5 Implementation Description Language 147

10.1.6 System Integration Mechanism 147

10.2 Features of the Methodology 149

10.3 Future Work 151

10.3.1 Supporting Tools 152

10.3.2 Component-based Programming Languages 153

TABLE OF CONTENTS
(Continued)

Chapter **Page**

10.3.3 Studies of Interactions among Components 154

10.4 Epilogue 155

APPENDIX A THE GRAMMAR OF IDL 156

APPENDIX B CONVERSION OF CUBIC GRAPHS INTO FLOWGRAPHS..... 161

REFERENCES 186

LIST OF TABLES

Table		Page
3.1	Classification framework for ADLs (adapted from [49])	41
3.2	ADLs with their architectural modeling features	42
5.1	A program example	59
5.2	Independent circuit sets	60
5.3	Independent program path sets	61
6.1	The definitions for inputs and output of the ECC system	75
8.1	An aggregation description framework	106
8.2	A component description framework	107
8.3	A system description template	109
8.4	An aggregation description template	109
8.5	A prime connection description template	109
8.6	A component description template	110
9.1	A system table template	131
9.2	A system table of solution 1 for the KWIC	131
9.3	A system table of solution 2 for the KWIC	131
9.4	A system table for the ECC	131
9.5	An aggregation table template	132
9.6	Aggregation table for Aggregation Process in Figure 8.9	133
9.7	Aggregation table for Aggregation ECC in Figure 8.11	133
9.8	Aggregation table for Aggregation Initialize in Figure 8.12	133
9.9	A component table template	134

LIST OF FIGURES

Figure **Page**

1.1 A view of a component-based software system 4

1.2 An approach for system integration 5

2.1 Flowchart nodes and their shapes 10

2.2 A program flowchart 11

2.3 A program flowgraph 12

2.4 Program control structures and their cubic flowgraphs and cubic graphs 13

2.5 A mixed program control structure, its cubic flowgraph and cubic graph 14

2.6 Three control flowgraphs and their regular expressions 15

2.7 Cubic graphs 18

2.8 A cubic flowgraph 22

2.9 A cubic graph 23

2.10 A cubic flowgraph with named vertices 24

2.11 Decomposition process of a cubic flowgraph 25

2.12 Prime cubic flowgraphs 26

3.1 Two architectures 35

5.1 A program control graph 60

5.2 A cubic flowgraph of the program in Table 5.1 63

5.3 The decomposition results of the cubic flowgraph in Figure 5.2 66

6.1 A system architecture based on subsystems and modules 69

6.2 A structure of a component-based system 70

6.3 A decomposition of the structure in Figure 6.2 71

LIST OF FIGURES
(Continued)

Figure **Page**

6.4 A component-based system structure for KWIC 73

6.5 Another component-based system structure for KWIC 74

6.6 ECC data flow diagram (adapted from [71]) 76

6.7 A component-based system structure for ECC 77

6.8 A component model 79

6.9 Aggregation pyramid of software architectures 80

6.10 A development model for component-based software systems 81

7.1 A Jackson Diagram 84

7.2 The Flowgraph of the Jackson Diagram in Figure 7.1 85

7.3 Implicit nodes in the Jackson Diagram 87

7.4 Jackson Diagrams and their prime flowgraphs 88

7.5 A solution with pipes/filters for the KWIC 91

7.6 A Jackson Diagram for the KWIC with pipes/filters 91

7.7 A flowgraph for the KWIC with pipes/filters 92

7.8 A solution for the KWIC with main program/subroutine 93

7.9 Jackson Diagram for the KWIC system with main program/subroutine 94

7.10 Flowgraph for the KWIC system with main program/subroutine 94

7.11 Aggregation Jackson Diagrams and flowgraphs 94

7.12 A decomposition of the non-prime flowgraph 95

7.13 A Jackson Diagram for the ECC system 96

7.14 The flowgraph for the ECC system 97

LIST OF FIGURES
(Continued)

Figure **Page**

7.15 Aggregation ECC: Jackson Diagram and flowgraph 98

7.16 Aggregation Maintain: Jackson Diagram and Flowgraph 98

7.17 Aggregation Initialize: Jackson Diagram and flowgraph 99

7.18 Aggregation Adjust: Jackson Diagram and flowgraph 99

7.19 Prime flowgraphs contained in aggregation Initialize 100

7.20 Prime flowgraphs contained in aggregation Adjust 100

8.1 Prime connections and their regular expressions 105

8.2 An IDL description for the KWIC with pipes/filters style 111

8.3 IDL description for component Input in the KWIC with pipes/filters 112

8.4 IDL description for component Shift in the KWIC with pipes/filters 112

8.5 IDL description for component Sort in the KWIC with pipes/filters 113

8.6 IDL description for component Output in the KWIC with pipes/filters 113

8.7 IDL system description for the KWIC with main program/subroutine 114

8.8 IDL description for aggregation KWIC with main program/subroutine 114

8.9 IDL description for aggregation Process with main program/subroutine 115

8.10 An IDL system description for the ECC system 116

8.11 The IDL description for aggregation ECC 116

8.12 The IDL description for aggregation Initialize 117

8.13 The IDL description for aggregation Adjust 118

8.14 The IDL description for aggregation Maintain 119

8.15 The IDL description for component Current 119

LIST OF FIGURES
(Continued)

Figure **Page**

8.16 The IDL description for component Desired 120

8.17 The IDL description for component Last 120

8.18 The IDL description for component Brake 121

8.19 The IDL description for component Throttle 121

9.1 The functional units and their dependency relationships 126

9.2 System integration 128

9.3 System synthesis process flowchart 138

9.4 The synthesis process for solution 2 of the KWIC system 139

9.5 Two possible aggregation queues for the ECC system 139

CHAPTER 1

INTRODUCTION

The demands for software systems have never stopped since computers have been widely used. At the same time the tremendously increased demands make the size and complexity of software systems growing rapidly. The expansion of software dimensions leads to much trouble to software production, for example, exceeded budgets, missed due date, poor performance, high maintenance cost, etc. Although the software engineering community has developed many methodologies, techniques and tools to try to resolve the problems, a breakthrough still has not been achieved. Recently an emerging software development style, component-based software with software architecture, holds great potential for overcoming some of these problems. This style focuses on the reuse of software components and changes the software development style from line by line coding to system integration from well-defined components. This style usually implies that software development starts with software architecture design, then buys the COTS (Commercial Off The Shelf) components, and finally integrates the whole system from the COTS. In fact, one of the ideal objectives of this style is to search the required components and to do the system integration automatically. The latter is really a motivation of this dissertation, and the work in this dissertation focuses on automatic system integration. As an introductory chapter, Section 1.1 shows the motivation of this work for component-based system integration and the objectives of this work. Section 1.2 sketches our approach for the objectives. Section 1.3 outlines the contents in this dissertation. In Section 1.4, the topics discussed in this chapter are summarized.

1.1 Objective

An ideal component-based software paradigm suggests that a software system be automatically built through semantics-driven component search and system integration [58]. It also suggests that the required components in a system may be coded in any languages, may be developed in any environments and may run on any platforms [58]. The former means that the real automation of development of a component-based software system must be based on the semantics of the components and the system. The latter means that the system may be a heterogeneous system because it is not necessary for all components to be written in the same language, to be compiled by the same compiler or to run on same platform.

For the second one, we can say that it is not a major challenge any more because the problem has been largely solved by middleware, such as the Common Object Request Broker Architecture (CORBA), the Distributed Component Object Model (DCOM), etc. However, the first one is still a bottleneck for the applications of component-based software [18]. In other words, we can say that there are two bottlenecks, one is semantics-driven component search and the other one is system integration. In this dissertation, we work on the component-based system integration.

The component-based software paradigm requires the system design to be a set of components glued together by connectors. In reality, components are the abstractions of functionality while connectors are the abstractions of interactions among the components. The detailed design for a component-based software system is really to design implementation algorithms for connectors because it is assumed that the components are already implemented. Generally speaking, component-based system integration starts

immediately after the detailed design of a system is done and the expected components are ready.

The objective of this dissertation is to present a methodology for component-based system integration. The methodology suggests a diagrammatic way to represent the detailed design, presents a method to decompose the design, proposes a language to describe the design, and finally proposes a synthesis mechanism to accomplish the system integration task. The author does not propose a semantics-driven method for component search because it is beyond the system integration aspect. But the author proposes a component model and a component description language in order to search the expected components in a system.

1.2 A Blueprint of the Methodology

As the author mentioned before, a component-based software system looks like a set of components, a set of connectors and a number of interconnections among connectors. The interactions among the components are abstracted to be connectors. All the interconnections among the connectors represent the configuration or topology of a system. Such an overall structure of the system can be drawn as a diagram shown in Figure 1.1. The diagram suggests us a view of a component-based software system. This view is that a component-based system consists of connectors and a connector consists of components. This view is called an aggregation view. An aggregation is really a set of components connected by one connector. A composite component en facto is a body in which there is one connector that connects a number of components. For a large system, the view can be extended to decompose a system into subsystems. Therefore, an aggregation could be a composite component, a subsystem or a system.

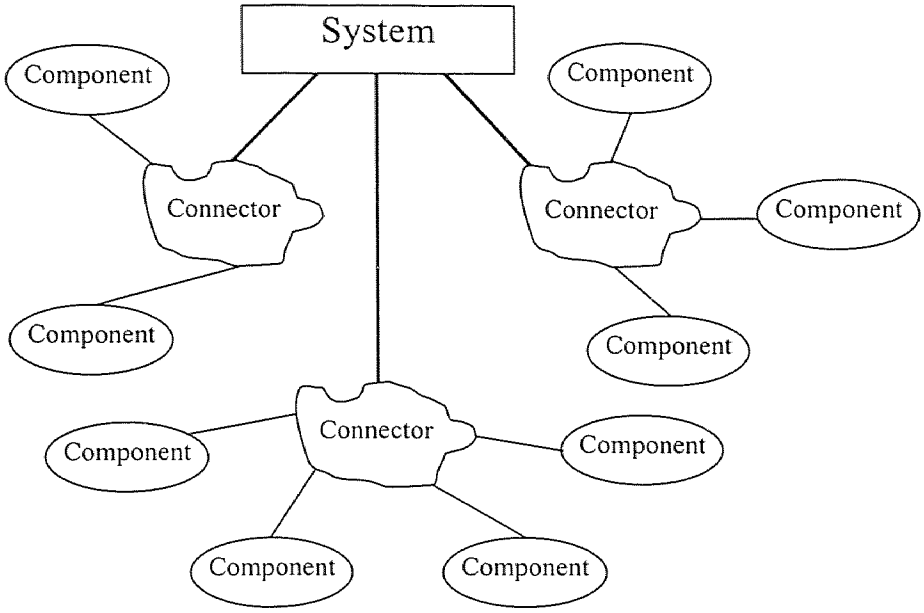

Figure 1.1 A view of a component-based software system

This view implies a special relationship between a system and a number of connectors and between a connector and a number of components. It is essentially a containing relationship. This view creates a base for our methodology. We decompose a software architecture into aggregations based on this view. Then we represent each aggregation as a Jackson Diagram [27], convert the diagram to a flowgraph called aggregation flowgraph (see Chapters 2 and 7) and decompose each aggregation flowgraph into prime flowgraphs called prime connections (see Chapter 8) that are the elementary unit for system integration. Aggregations are also the description unit in our implementation description language (see Chapter 8). Finally the integrated system is also considered as an aggregation. Therefore, the concept of an aggregation plays a very important role in our approach for the methodology of component-based system integration. The ideas illustrated above are summarized in Figure 1.2.

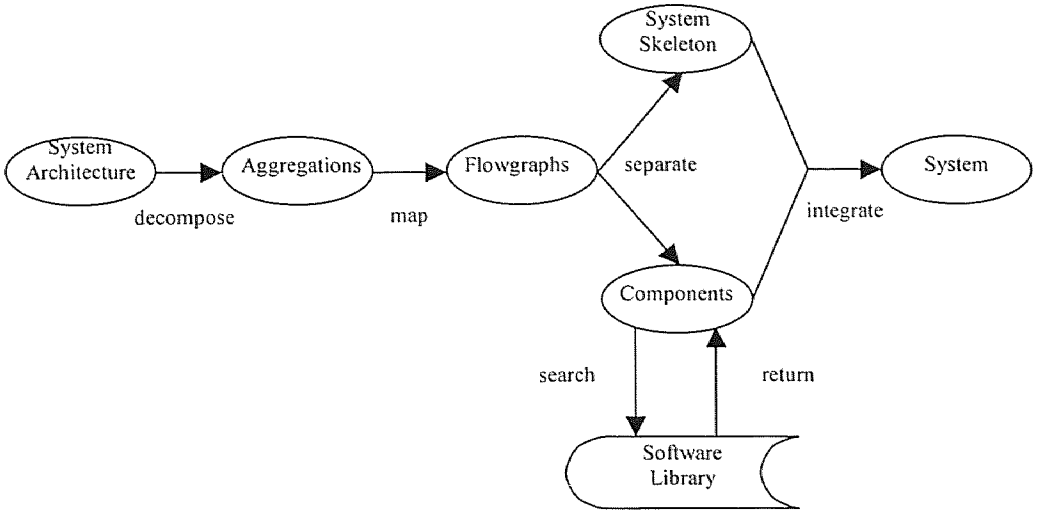

Figure 1.2 An approach for system integration

1.3 An Outline of the Dissertation

There are ten chapters and two appendices in this dissertation. Besides this chapter, the introduction, this dissertation contains nine more chapters. Chapters 2, 3, and 4 cover the background knowledge about flowgraphs, software architectures and component-based software. Chapter 5 presents new ways to compute the cyclomatic complexity of a program based on flowgraphs. The new ways are applied in our methodology. Chapters 6, 7, 8, and 9 introduce our methodology for component-based system integration. Chapter 10 serves as conclusions for the dissertation. In Appendix A, the definition of implementation description language in Extended Backus-Naur Form (EBNF) is listed. Appendix B introduces the conversion of a cubic graph to cubic flowgraphs. All the chapters are outlined in the rest of this section.

Chapter 2 is a survey of flowgraphs, cubic graphs and the relationship between flowgraphs and cubic graphs. We focus on the structured flowgraphs, their

representation, regular expressions and the decomposition and composition of flowgraphs. The features of flowgraphs are used in our methodology. Chapter 3 explores related work to our methodology on software architectures. The survey includes definitions, software architecture styles, architectural views, and architecture description languages. Chapter 4 discusses component-based software. The definitions for components, components in software libraries, component representation and search, component-based development, and the evaluation, analysis and management of component-based software are surveyed.

Chapter 5 focuses on the cyclomatic complexity of programs. The current computing methods are surveyed. Then we introduce new methods for computing the cyclomatic complexity based on cubic flowgraphs and prove two theorems about the new methods. We also compare the new methods with the current methods. The new methods are applied in our methodology for computing how many prime flowgraphs are included in an aggregation flowgraph before decomposing the aggregation flowgraph into prime ones.

Chapter 6 examines software architectures in terms of aggregate relationships among the system, subsystems, connectors and components. We call the relationships aggregations. We conclude that a component-based software system is a set of aggregations that belong to system level, subsystem level and connector level. The aggregation relationship is abstracted as a software architectural view, the aggregation view. This view plays a very important role in the integration methodology. We also propose a component model for specifying a component. This model is very comprehensive for the representation of a component in a software library. A modified

version of the component model can also be used for component search. To enhance the introduction and explanation of the methodology, the Key Word in Context (KWIC) problem and the Embedded Cruise Control (ECC) problem are discussed. The two problem examples are used through Chapter 6 to Chapter 9 to show how the methodology is applied.

Chapter 7 explains why Jackson Diagrams are used to represent a system and its aggregations. Then it is discussed how a component-based software system is represented with Jackson Diagrams and flowgraphs based on the aggregations. The conversion of a Jackson Diagram to a flowgraph is also discussed. Furthermore we discuss how an aggregation flowgraph is decomposed into prime flowgraphs. These principles are demonstrated by using the problem examples, KWIC and ECC.

Chapter 8 starts with an introduction of prime connections for prime flowgraphs. A framework for system implementation description is proposed in this chapter. The framework covers the system skeleton including a system and all its aggregations, and components. Then we propose a language for system implementation description. The description templates for a system, an aggregation and a component are introduced. A number of description examples are used to enhance the description framework.

The system integration is discussed in Chapter 9. We provide a system synthesis mechanism for system integration. The mechanism makes use of a parser, a component manager, and a synthesizer to accomplish the system integration based on system descriptions, aggregation descriptions and component descriptions.

In Chapter 10 our methodology is summarized. A number of characteristic features of the methodology is emphasized by comparing our work with some related

work. We also review the contributions made in this work. Finally we explore the future research directions based on this dissertation.

1.4 Summary

This chapter serves as an introduction to the whole work in this dissertation. The objectives and motivations of the work are introduced. Because we are going to propose a methodology for component-based system integration, the main ideas of the methodology are sketched as a blueprint. Then the main contents in other chapters are briefly discussed.

Based on the objectives and motivations introduced in Section 1.1, the primary task of this work is to propose a methodology, in order to integrate a software system from components automatically. This is a difficult task because system integration is a bottleneck of software system development [72]. The proposed methodology takes the 'divide and conquer' strategy to handle system integration. The division method is based on the decomposition theory of flowgraphs. The conquering method is based on architectural aggregation view. From this point of view, the detailed design of a system is represented in an implementation description language, then it is decomposed into aggregations, and at last, the aggregations are composed from prime connections.

CHAPTER 2

FLOWGRAPHS AND CUBIC GRAPHS

In this chapter, flowgraphs and cubic graphs are surveyed because of some useful features they have. One of the most important features is the regular expression representation of flowgraphs. In other words, every flowgraph has a regular expression, a formal representation for the flowgraph. The other feature is the decomposition and composition of flowgraphs. Since the latter one has a root in cubic graphs, cubic graphs are also discussed. The decomposition is used for dividing an entity into smaller ones. The composition is used for integrating an entity from some parts. They all are used as theoretic tools in our integration methodology. For example, flowgraphs are used for the representation of aggregations. Section 2.1 introduces flowgraphs, how to create a flowgraph, a special kind of flowgraphs, structured flowgraphs and their representation, regular expressions. Cubic graphs and some of their properties are discussed in Section 2.2. Finally the relationships between flowgraphs and cubic graphs are explored in Section 2.3.

2.1 Flowgraphs

When we design a traditional structured program, we often use a diagrammatic representation for an algorithm and then write a program according to the representation. We call this representation a program flowchart. A flowgraph is a variation of a flowchart. A flowgraph is a strongly connected directed graph. A flowchart is not a graph because two lines can be joined to become one line without a node. Their definitions are introduced in this section.

2.1.1 Program Flowcharts

Flowcharts have been introduced as a tool for programming. A flowchart consists of several geometrical shapes that are connected by directed edges. There are two distinguished shapes, the *begin* node and *end* node in a flowchart. The other shapes may be *input* nodes, *output* nodes, *process* nodes, and *decision-making* nodes in a flowchart. The different geometrical shapes for flowchart nodes are shown in Figure 2.1.

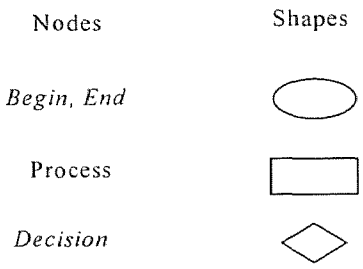

Figure 2.1 Flowchart nodes and their shapes

The input, output and process nodes represent basic functional statements in a program. For the sake of convenience, we assume that the input and output nodes are also represented as process nodes because they are special processes in programs. Two edges in a flowchart may join together to be one edge. Figure 2.2 shows a typical flowchart example, which does not necessarily represent a meaningful program. The lowercase letters represent input, output and processes. The uppercase letters represent decision-making conditions.

2.1.2 Program Flowgraphs

A flowgraph is a strongly connected directed graph with decision vertices, junction vertices, process vertices, and a distinguished arc from ending vertex to starting vertex

[61]. The vertices except junction vertices are the same as in the original flowcharts. But there are no explicit begin and end vertices. The junction vertices are the junction points indicated by (*) in Figure 2.2 where two branches are joined to one branch. We color decision vertices black and leave junction vertices white in order to distinguish them from each other. The two types of vertices are represented as circles.

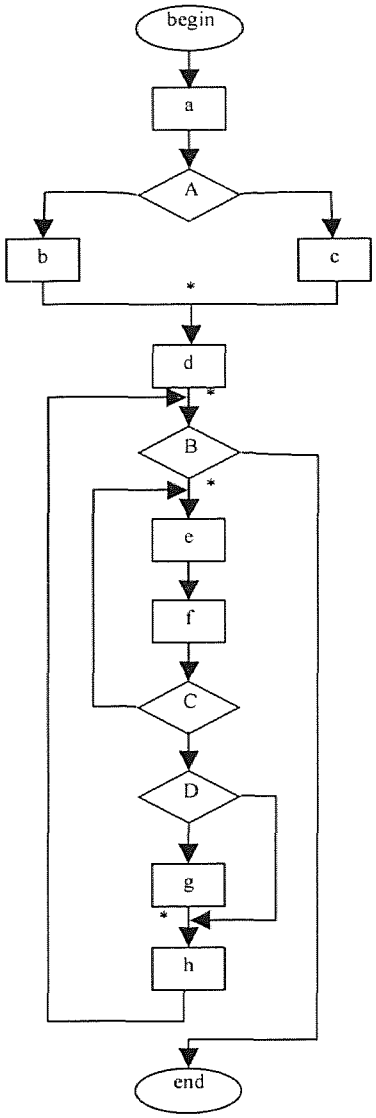

Figure 2.2 A program flowchart

The junction vertices convert a flowchart to a flowgraph. The distinguished arc makes a flowgraph strongly connected. These are the significant differences between a flowchart and a flowgraph. Basically a junction point in a flowchart originates a decision-making node because the branches would be definitely joined somewhere. Therefore it is natural for us to introduce junction vertices. There are four junction points in the example flowchart. The sample flowgraph shown in Figure 2.3 is converted from Figure 2.2.

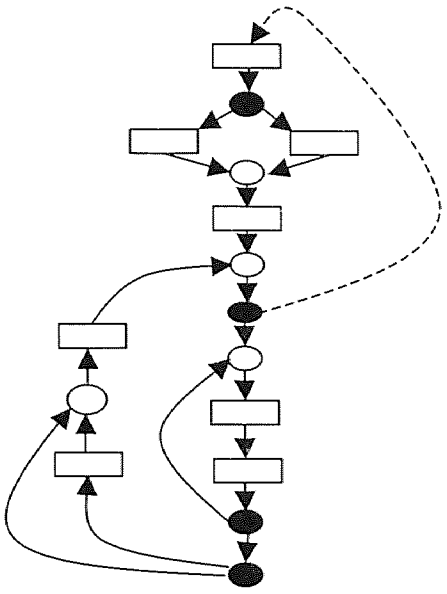

Figure 2.3 A program flowgraph

2.1.3 Structured Flowgraphs

Currently, many popular programming techniques are based on the structured programming style. This style requires that a programmer design a program using just three structures: sequential structure, selective structure and repetitive structure. The primary structure is the sequential one. Each structure has exactly one entry and one exit. The structures can be used to compose a large program sequentially or nestedly. The typical structures are shown in Figure 2.4. If a program flowgraph represents a structured program, we call the flowgraph a structured flowgraph.

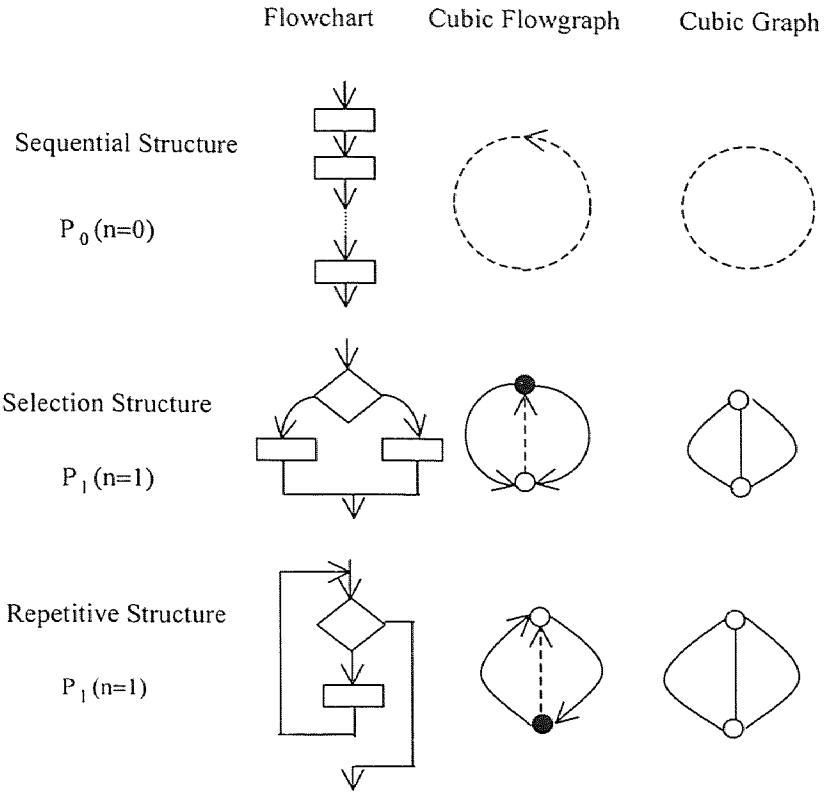

Figure 2.4 Program control structures and their cubic flowgraphs and cubic graphs

The flowcharts, cubic flowgraphs and cubic graphs of the three control structures are drawn in Figure 2.4. The cubic graphs are discussed in Section 2.2. A cubic flowgraph discussed in Section 2.3 is a flowgraph in which all the process nodes are removed. The n in Figures 2.4 and 2.5 is the number of nodes in a cubic flowgraph. A cubic flowgraph with n=0, n=1, and n=2 is named P_0, P_1 and P_2 respectively as shown in Figures 2.4 and 2.5, where the P_0 and P_1 are degenerate cases.

Based on the structured programming idea, every program is built on the three structures. Just like the program example in Figure 2.2, the whole structure of the program is a sequential one containing a selective structure and a repetitive structure. In the repetitive structure there are another selective structure and repetitive structure. The

embedding is sequential or nested. The flowgraph in Figure 2.3 is a structured flowgraph.

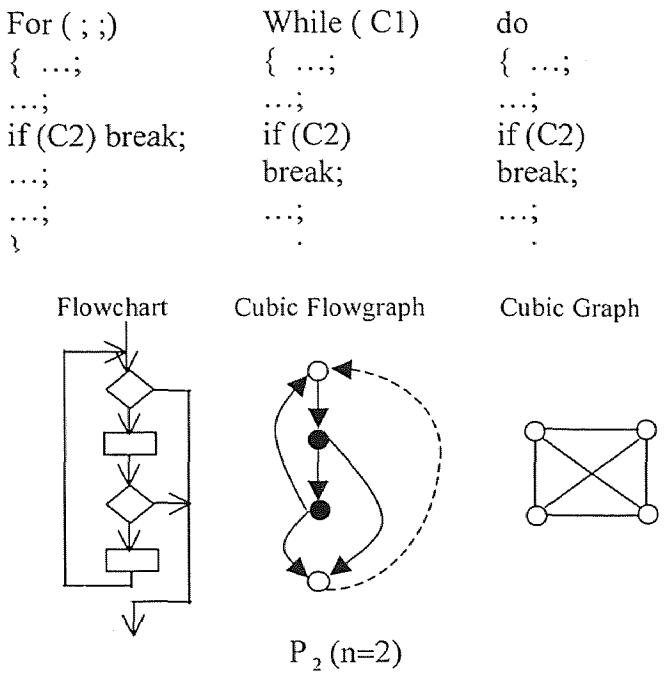

Figure 2.5 A mixed program control structure, its cubic flowgraph and cubic graph

We note that a structured program is composed of these three structures. The composition is sequential or nested. And we also note that a structured program can be decomposed into program segments that are based on the three basic structures. Besides standard control structures, a mixed control structure is often used in some programming languages. The structure can relate to program segments such as a loop structure with an exceptional break statement as shown in Figure 2.5. Although this structure violates the structured programming criteria, it is commonly used in some languages like C, C++, and Java. Thus it is considered as a special control structure here. The structure also has a flowchart, a cubic flowgraph and a cubic graph respectively shown in Figure 2.5.

2.1.4 Regular Expressions for Flowgraphs

Regular expressions are used to represent languages that are recognized by finite state automata. Here we use regular expressions to represent the computation set of any program flowgraph. It means that a program flowgraph corresponds to a regular expression. Actually the correspondance is an application of the algebra of regular expressions. The definitions for the algebra of regular expressions and the regular language $\lambda(\alpha)$ used here are based on the work in [62].

The algebra of regular expressions is defined over an alphabet Σ with distinguished elements 0 and 1. It is closed under the three binary operations, *sum*, *product* and *star* where *sum* stands for selection structure, *product* for sequential structure and *star* for repetitive structure as shown in Figure 2.6:

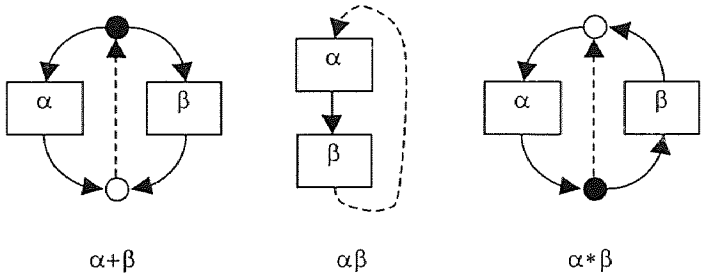

Figure 2.6 Three control flowgraphs and their regular expression

(i) sum: $\alpha+\beta$,

(ii) product: $\alpha\beta$, and

(iii) star: $\alpha*\beta$,

while satisfying the axioms:

(1a) $\alpha+\beta=\beta+\alpha$,

(1b) $\alpha+\alpha=\alpha$,

(2a) $\alpha+(\beta+\gamma)=(\alpha+\beta)+\gamma$,

(2b) $\alpha(\beta\gamma)=(\alpha\beta)\gamma$,

(3a) $\alpha(\beta+\gamma)=\alpha\beta+\alpha\gamma$,

(3b) $(\alpha+\beta)\gamma=\alpha\gamma+\beta\gamma$,

(4a) $0\alpha=0$,

(4b) $0+\alpha=\alpha$,

(5a) $1\alpha=\alpha$,

(5b) $1*\alpha=1*(1+\alpha)$,

(6a) $\alpha\beta*\gamma=\alpha(\beta*\gamma\alpha)$, and

(6b) $\alpha*\beta=(\alpha*\beta)\beta\alpha+\alpha$.

Based on the definition of the algebra of regular expressions above, the regular language can be defined. For each regular expression α over the alphabet Σ, the language $\lambda(\alpha)$ represented by α is defined.

$\lambda(\sigma) = \{\sigma\}$ for $\sigma \in \Sigma$, and

$\lambda(0) = \varnothing$ and $\lambda(1) = \{\varepsilon\}$

$\lambda(\alpha+\beta) = \lambda(\alpha) \cup \lambda(\beta)$

$\lambda(\alpha\beta) = \lambda(\alpha)\lambda(\beta)$

$$\lambda(\alpha*\beta) = \bigcup_{k=0}^{\infty} \lambda(\alpha)(\lambda(\beta)\lambda(\alpha))^k$$

Equality of regular expressions: $\alpha = \beta$ if and only if $\lambda(\alpha) = \lambda(\beta)$

Composite regular expression: For a regular expression, $\rho = \rho(\sigma)$, if σ is substituted with the regular expression μ, wherever it occurs, then a composite regular expression $\rho(\sigma(\mu))$ is obtained and $\lambda(\rho(\sigma(\mu))) = \lambda(\rho)(\sigma{:}\lambda(\mu))$. The program flowgraph in Figure 2.3 corresponds to the following regular expression:

$$a(b+c)d(1*((ef*1)(1+g)h)).$$

2.2 Cubic Graphs

Generally a graph is defined as a set of vertices V and a set of edges E which are lines and are used to connect the vertices [23]. Our notations here follow [61]. A graph has a representation, G=(V, E). The vertices connected by an edge are called end-points of the edge. If the end-points of an edge are the same, the edge is called a self-loop. If more than one edge have the same end-points, the edges are parallel. The parallel edges are different edges. A graph is a simple graph if there are no self-loop and parallel edges. A graph is a multi-graph if parallel edges are allowed. Each vertex v in V has a degree, the number of edges incident with v, written deg(v). There is a relationship between the number of edges and the sum of degrees of vertices in a graph as shown below; here we let |E| be the number of edges, |V| be the number of vertices and n=|V|:

$$2*|E| = \deg(v_1) + \deg(v_2) + \cdots + \deg(v_n)$$

The introduction of general graphs serves the purpose of introducing cubic graphs. The relationship above implies important properties of cubic graphs, which are described in Section 2.2.2. Cubic graphs are a special kind of graphs. Besides the common properties of general graphs, a cubic graph has its own properties. In this section we discuss the definition and properties of a cubic graph. The discussion of cubic graphs is to explore their relationships to program flowgraphs.

2.2.1 Definitions

In order to show the properties of cubic graphs four definitions are introduced. One is about cubic graphs and the other three are about the connectivity of a cubic graph. The first three definitions used here follow Prather's work in [61]. We introduce Definition 4 for a comprehensive understanding of the connectivity of cubic graphs. Based on the connectivity, we will survey a theory for decomposition and composition of cubic graphs [61]. Here multi-graphs are allowed. Graph G is a doubly connected cubic graph. Graphs G_1 and G_2 are triply connected. Graph G_3 is singly connected. In fact, graph G is the composition of graphs G_2 and G_1. The examples of cubic graphs are shown in Figure 2.7.

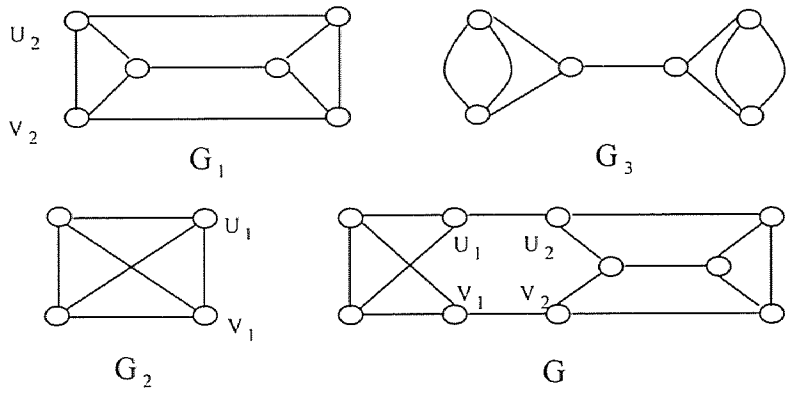

Figure 2.7 Cubic graphs

Definition 1

For a graph G=(V, E), if each vertex in V has degree 3, then the graph is a cubic graph.

Definition 2

A cubic graph is doubly connected if exactly two edges need to be removed in order to disconnect the graph.

Definition 3

A cubic graph is triply connected if at least three edges need to be removed in order to disconnect the graph.

Definition 4

A cubic graph is singly connected if exactly one edge needs to be removed in order to disconnect the graph.

2.2.2 Properties

For any graph, the number of edges is equal to half of the sum of each vertex's degree, as mentioned before. The relationship implies two properties for a cubic graph summarized below because every vertex in a cubic graph has degree 3:

1) Every cubic graph has an even number of vertices represented as 2n (n=0,1,2, ...).
2) Every cubic graph with 2n vertices has 3n edges (n=0,1,2, ...).

Generally a cubic graph is connected or consists of a number of connected components. The connectivity can be divided into three kinds according to Definitions 2, 3 and 4. Those are singly connected, doubly connected and triply connected. Therefore a property related to connectivity of cubic graphs is concluded below:

3) Every cubic graph is singly connected, doubly connected, or triply connected.

2.2.3 Decomposition and Composition of Cubic Graphs

The decomposition of a cubic graph means dividing the cubic graph into at least two cubic graphs. The essence of the decomposition is to remove some edges from a given cubic graph and to add some edges to decomposed components such that the original graph is disconnected and the resulted components are kept to be cubic graphs. The composition of two cubic graphs means combining two cubic graphs into one cubic graph. The two cubic graphs are arbitrary cubic graphs. But the composed cubic graph must be doubly connected because one edge is removed from each original cubic graph and two new edges are added to connect the cubic graphs such that the composed graph is a cubic graph.

Now we discuss the decomposition of cubic graphs. We will limit the discussion to doubly and triply connected cubic graphs because singly connected cubic graphs have no corresponding program flowgraphs according to the strong connectivity of a flowgraph. Here are the steps for decomposition and composition [61].

Decomposition steps:

a. Choose two edges and remove them such that the graph is disconnected.

b. Add one edge to each component such that the component is still a cubic graph.

Composition steps:

a. Choose one edge from each given cubic graph and remove it;

b. Add two new edges such that one end vertex of each new edge is a vertex in a graph and another end vertex of the new edge is in another graph and the resulting graph is still a cubic graph.

According to the assumption and the steps we conclude that if a cubic graph is doubly connected, then the graph can be at least decomposed into two cubic graphs and if a cubic graph is triply connected, then the graph can not be decomposed into two cubic graphs. Let us examine an example of decomposing and composing a cubic graph in Figure 2.7. When we decompose a cubic graph, we need to identify two edges in the given cubic graph, for example, the edges (U_1, U_2) and (V_1, V_2) in G. Remove the edges and add the edges (U_1, V_1) and (U_2, V_2) to the graph. In this way the original graph G is decomposed into two cubic graphs G_1 and G_2. When we compose a cubic graph from two cubic graphs, we also need to identify two edges, one from one cubic graph and one from the other. For example, edge (U_1, V_1) in G_2 and edge (U_2, V_2) in G_1. Remove the edges and connect the two graphs from U_1 to U_2 and from V_1 to V_2. We get a new cubic graph G that is the composition of G_2 and G_1.

2.2.4 Prime Cubic Graphs

The decomposition rules discussed in Section 2.2.3 can be used to classify all the cubic graphs into two kinds. One is the decomposable cubic graphs and the other is the non-decomposable cubic graphs. Based on this observation, here is a definition [61] for the two kinds of cubic graphs. The definition implies that, for a given cubic graph, either it is a prime one or it is composed of at least two prime cubic graphs. A doubly connected

cubic graph is a non-prime cubic graph and a triply connected cubic graph is a prime cubic graph.

In fact, a cubic graph is either a prime one or can be composed from several prime ones. This is why we particularly discuss prime cubic graphs. When the theory of cubic graphs is applied to flowgraphs, a prime cubic flowgraph stands for a non-decomposable program segment, such as the cubic flowgraphs for three program control structures in Figures 2.4 and 2.5.

Definition 5

A cubic graph is a prime cubic graph if it is non-decomposable; otherwise, it is a non-prime cubic graph.

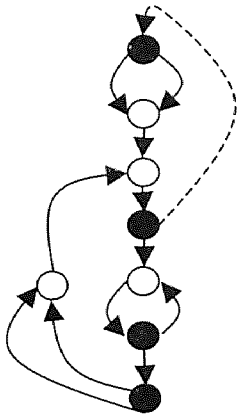

Figure 2.8 A cubic flowgraph

2.3 Cubic Flowgraphs

If all the process vertices in a flowgraph are eliminated and the two adjacent arcs around a process vertex are replaced by one arc with the same direction, a new flowgraph is obtained (Figure 2.8). Such a flowgraph has an interesting property that each vertex has degree 3. A decision vertex has indegree 1 and outdegree 2. A junction vertex has indegree 2 and outdegree 1. This kind of flowgraph is called a cubic flowgraph [61].

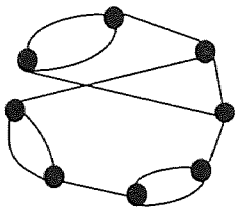

Figure 2.9 A cubic graph

A cubic flowgraph is a special kind of cubic graph. If we ignore the color of each vertex and the orientation on each edge, the cubic flowgraph is a general cubic graph. Figure 2.9 shows the cubic graph derived from the cubic flowgraph in Figure 2.8. As we discussed in Section 2.1, a structured program has a unique flowgraph. If the process nodes in a program flowgraph are ignored, it is a cubic flowgraph that corresponds to a cubic graph. The conversion from flowgraphs to cubic graphs suggests that the properties of cubic graphs can be applied to flowgraphs. The most important one is the application of decomposition and composition of cubic graphs to flowgraphs. It indicates a way for programmers to compose a large program from some smaller programs. This section explores the various aspects of decomposition and composition for cubic flowgraphs and then for programs.

2.3.1 The Decomposition and Composition of Cubic Flowgraphs

Before we discuss the details of decomposition and composition of cubic flowgraphs, the connectivity of cubic flowgraphs needs to be discussed. There are doubly connected cubic flowgraphs and triply connected cubic flowgraphs. The definitions are as same as the definitions of doubly connected and triply connected cubic graphs. There are cubic graphs that are singly connected. However, no flowgraph is singly connected because of the strong connectivity.

The decomposition and composition of cubic flowgraphs follow the same steps as the decomposition and composition of a cubic graph, in addition to considering the orientations of the removed arcs and added arcs. The added arcs will maintain the incident decision vertex with indegree 1 and outdegree 2 and the incident junction vertex with indegree 2 and outdegree 1.

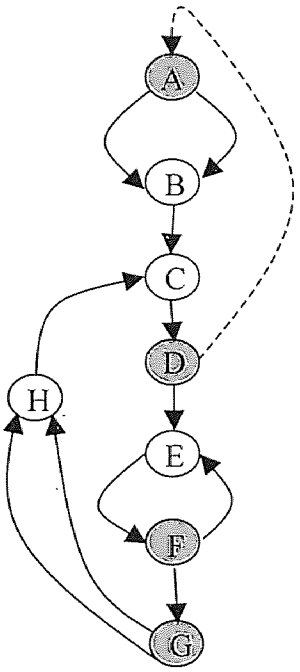

Figure 2.10 A cubic flowgraph with named vertices

An example cubic flowgraph is shown in Figure 2.8. Figure 2.10 is a copy of the cubic flowgraph in which the vertices are named and the decision nodes are colored light gray, instead of black. First, let us decompose the cubic flowgraph. The cubic flowgraph is doubly connected. It is decomposable in terms of the decomposition of cubic graphs. Now what we need to do is to choose two arcs and remove them such that it is disconnected.

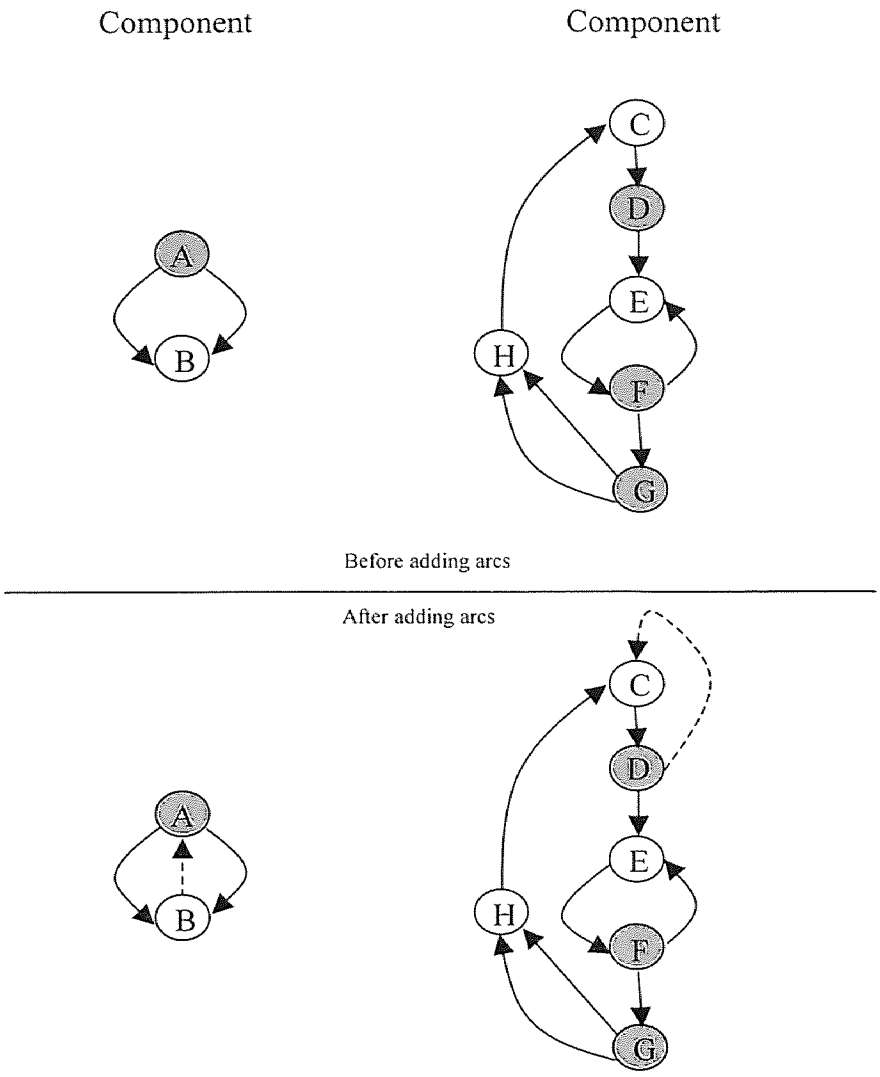

Figure 2.11 Decomposition process of a cubic flowgraph

For example, choose arc (B, C) and then arc (D, A) must be chosen because only this combination can disconnect the graph. After removing the two arcs, the graph is disconnected into two parts. Then one arc needs to be added to each component such that it is still a cubic flowgraph. In this way an arc from vertex B to vertex A is added to component 1 as a distinguished arc and an arc from vertex D to vertex C is added to component 2 as a distinguished arc too. The decomposition process is shown in Figure 2.11.

In the figure, the components are marked as before adding arcs and after adding arcs. The added arcs are distinguished arcs because they start with the ending vertex and end with the starting vertex in their components. Clearly, after adding arcs, component 1 is a triply connected cubic flowgraph and cannot be decomposed and component 2 is still a doubly connected cubic flowgraph and can be decomposed again. The decomposition can continue until no more doubly connected cubic flowgraphs exists. The final decomposition results from component 2 are shown in Figure 2.12.

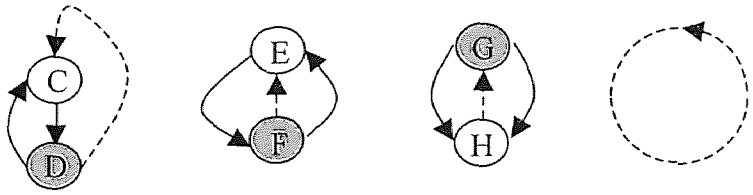

Figure 2.12 Prime cubic flowgraphs

The final decomposed results are five prime cubic flowgraphs. They belong to three program control structures. It indicates that a structured program is really composed from the three control structures and can be decomposed into the three control structures. On the other hand, the decomposition does not change the properties of structured programs.

The above decomposition process actually implies the composition process that is the reverse of the decomposition. We simply remove arc (B, A) from component 1 and arc (D, C) from component 2. Then an added arc links vertex B to vertex C. The other added arc links vertex D to vertex A. As a result, the two cubic flowgraphs in Figure 2.11 are composed into the cubic flowgraph in Figure 2.10. It is noteworthy that the added arcs must not violate the properties of a cubic flowgraph.

Based on decomposition and composition of cubic flowgraphs, we can also define a cubic flowgraph as a prime cubic flowgraph if it is non-decomposable; otherwise it is a non-prime cubic flowgraph. In fact, the composition of cubic flowgraphs follows the same rules as the composition of structured programs from the three fundamental structures with sequential or nested composition. Therefore the decomposition and composition of cubic flowgraphs do not change the properties of structured programs.

2.4 Summary

Flowgraphs, cubic graphs and cubic flowgraphs are discussed in this chapter. Simply speaking, the topics related to the integration methodology for component-based software systems proposed in this dissertation are surveyed. Flowgraphs are surveyed in Section 2.1. Cubic graphs are surveyed in Section 2.2. Cubic flowgraphs are surveyed in Section 2.3, in which the relationships between flowgraphs and cubic graphs are explored. The main topics of flowgraphs, cubic graphs, and cubic flowgraphs are summarized in this section.

Traditionally a flowchart is used to represent a computer program. A flowgraph is a variation of a flowchart. It is also used to represent a computer program. However, the difference between a flowgraph and a flowchart is that a flowgraph is a directed graph, but a flowchart is not a graph. In Section 2.1, the definitions for a flowchart and a flowgraph are introduced. The relationships between them are explored, and it is discussed how to convert a flowchart to a flowgraph. We also introduce structured flowgraphs for structured programs and a formal representation, regular expressions, for flowgraphs. It is indicated that each flowgraph has a regular expression based on three program control structures and process nodes in a program.

In Section 2.2, based on the definition of a general graph, we introduce the definition of a cubic graph. The properties of cubic graphs are explored. We also discuss how to decompose a cubic graph into other cubic graphs and how to compose a cubic graph from other cubic graphs respectively. If a cubic graph is decomposable or non-decomposable, a cubic graph can be called a prime cubic graph or a non-prime cubic graph respectively. The decomposability of a cubic graph determines if a cubic graph is singly connected, doubly connected, or triply connected.

Section 2.3 focuses on cubic flowgraphs. We discuss how to produce a cubic flowgraph from a general flowgraph. How the decomposition and composition theory of cubic graphs is applied to cubic flowgraphs is discussed in order to decompose a program into some program segments or compose a program from other programs. The decomposition and composition of cubic flowgraphs plays an important role in our methodology.

CHAPTER 3

SOFTWARE ARCHITECTURES

Generally each software system has an architecture. It consists of functional modules and the interactions among the modules. System analysts draw a diagram to represent the architecture with boxes for modules and lines for the interactions. The study of software architectures now is attracting more attention from academia, government and industries. The topics studied currently include definitions for software architectures; software architecture styles; architecture-based development and supporting tools; software architecture description and architecture description languages; formalization of software architectures; domain-specific software architectures. The primary motivation for the software community to study software architectures is to solve problems such as unsatisfactory quality, high costs, late delivery, low reliability, and low productivity, that have not been resolved yet with the current software development paradigms. In this chapter a survey is supplied for the main topics of software architectures. We survey them because they are the fundamentals of our methodology for component-based system integration.

3.1 Definitions

An architecture generally consists of a partitioning strategy and a coordination strategy [14]. The partitioning strategy is used to divide a whole system into separate components. The coordination strategy expresses the interactions among the components. What is a software architecture? Based on the general definition for an architecture, we examine the definitions for a software architecture. Although there is no

universally accepted definition for software architecture, it does not mean that we do not have any definition candidates for software architectures. There have been many definitions for software architectures from previous work. The representative definitions are discussed here. Then we propose our own definition for software architectures at the end of this section. They almost imply everything for software architectures. They supply us with a comprehensive view on software architectures.

A definition from [4] indicates that a software architecture is the structure or structures of a program or computing system composed of components and the relationships among them, with the exposed properties of those components. The properties could involve provided services, performance characteristics, fault handling, shared resource usage, and so on. They make it convenient to build a composite component or integrate systems from other components. First of all, the definition implies that architecture defines the interface of components including behavior descriptions. Then it implies that a component could be anything, such as an object, a process, a library, a database or a commercial product. It also implies that a software system must have an architecture if it is composed of components and relations among them.

Another definition for software architectures was given by Garlan and Shaw [19]. It emphasizes designing and specifying the overall system structure. The structural issues are gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and

selection among design alternatives. The system design does not focus on algorithms and data structures of the computation.

Hayes-Roth's definition [26] clearly claims that a software architecture, in reality, is composed of functional components with the description of their behaviors and interfaces and the interconnections among the components. A similar definition is found in [21], but an additional aspect considers principles and guidelines governing their design and evolution over time. In contrast to the definitions mentioned above, a definition from [7] emphasizes the importance of system stakeholders' requirement statements and their satisfaction by components, connections and constraints in a system. Kruchten's definition enhances the importance of choosing a proper architectural style in order to satisfy the major functionality and performance requirements, e.g., scalability and availability [39]. The definition given by Abowd [1] suggests that all the life-cycle issues of a software architecture should be covered.

Software architectures can be examined from different angles. The observation from each angle is a view of software architectures like we will discuss in Section 3.3. Here is a definition supplied in [68] that summarizes four views. The conceptual view on software architecture is the description of system in terms of its major design elements and the relationships among them. The modular view on interconnection covers two orthogonal structures: functional decomposition and layers. The runtime view illustrates the dynamic structure of a system. The implementation view focuses on how the source code, binaries, and libraries are organized in the development environment.

Perry and Wolf introduced a definition for software architectures by classifying the architectural elements into processing elements, data elements and connection elements

[60]. The processing elements are components that perform transformations on the data elements. The connection elements are the glue that connects every element of the architecture together. The data elements are the information that needs to be processed. The connection elements could be procedure calls, shared data, messages, and so on. Perry and Wolf used water polo as an example to show that the swimmers are the processing elements, the ball is the data element, and the water is the connection element that glue the ball and the swimmers together. Such a situation can also be applied to polo and soccer. "They all have a similar 'architecture' but differ in the 'glue' – that is, they have similar elements, shapes and forms, but differ mainly in the context in which they are played and in the way that the elements are connected together." The examples imply that the connecting elements are key factors in distinguishing one software architecture from another and play an important role in deciding the characteristics of a particular architecture or architectural style. In addition to the elements, an architecture includes properties, that are constraints on architectural elements, and relationships, that determine how the elements interact.

Moriconi defined that a software architecture is represented using the concepts: component, interface, connector, configuration, mapping, architectural style [53]. A component is defined as an object with independent existence, such as, a module, process, procedure, or variable. An interface is a typed object that is a logical point of interaction between a component and its environment. A connector is a typed object relating either interface points or components. A configuration means a collection of constraints that embed objects into a specific architecture. A mapping represents a relation between the vocabularies and the formulas of an abstract and a concrete

architecture. The architectural style is composed of a vocabulary of design elements, a set of well-formedness constraints that must be satisfied by any architecture expressed in the style, and a semantic interpretation of the connectors. A noteworthy viewpoint in this definition is that "the semantics of components is not considered part of an architecture, but the semantics of connectors is."

The definitions surveyed above imply that the kernel of software architectures is the overall structural issues of software systems, not the choices of algorithms and data structures of computation. The structural issues include common architectural elements: components, connectors, overall structures, constraints on components and connectors, the life-cycle issues and all the requirements of the whole system, such as, performance, scalability, availability, evolution. The components are abstracted from modules. The connectors are abstracted from the interactions among the modules and define communication protocols for the interactions. The overall structures are defined as system configurations or topologies [49].

Therefore we define that a software architecture is simply a topology or separate topologies consisting of independent components and connectors. Basically a connector is used to connect at least two components in the topologies. The connectors are the abstractions of interactions among components. The most important thing in the definition is the independence of the components. The independence means that any component is independent of a software architecture even if components are the design results of a system. In other words, a component can be used in many software architectures that is the reusability of components. This is also the significant difference between a traditional development and architecture-based development.

3.2 Software Architectural Styles

Many engineering disciplines have their own design styles. Every software system is actually built on an organizational style. Software architectural styles are the organization patterns of software systems. The typical examples for the patterns are client-sever systems, hierarchical organization for main program and subroutines, pipe-filter systems, and so on. The styles fall into four main groups: object-oriented architectures including information hiding; state-based architectures; feedback-control architectures; and architectures for real-time systems [66].

The styles are pipes and filters, data abstraction and object-oriented organization, event-based implicit invocation, layered system, repositories, interpreters, process control, distributed processes (client-server organization), main program/subroutine organizations, domain-specific software architectures and state transition systems [64][66]. It is possible for designers to combine some of the styles into one system. The combination produces many mixed styles called heterogeneous architectures. The classification of architectural styles focuses on the properties of components and connectors in each style. For example, the components are filters and the connectors are pipes in the style of pipes and filters. If we focus on the topologies of software architectures, the architectural styles could be hierarchies or networks.

One of the goals of exploring architectural issues is to facilitate component-based software development. Besides the independence of components, we find that connectors play an important role in software. For example, in Figure 3.1 the two architectures have the same components, but different connectors. Therefore they are different architectures even if they belong to the same styles. The boxes are components and arrows are

connectors. Actually connectors have some important properties in a software architecture. One of them is that the topology of an architecture is determined by its connectors as shown in Figure 3.1. Furthermore, the architectural styles imply that all the components and connectors can be divided into different groups. Thus we can classify the components and connectors into different types so that they can be generalized. Some types for components and connectors can be found in [36].

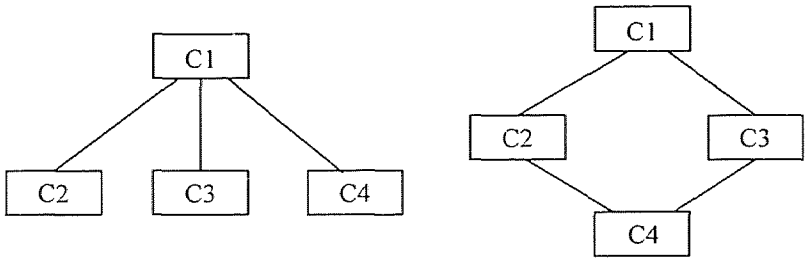

Figure 3.1 Two architectures

3.3 Architectural Views

Architectural views provide us with different ways to observe, study and understand a software architecture. The views combined together reflect all the properties of a software architecture. Clements and Northrop indicated several views [12] that are conceptual view, module view, process view, and physical view. Soni, Nord and Hofmeister defined software architectures in terms of four views that are called conceptual architecture, module interconnection architecture, execution architecture and code architecture [68]. These architectures are different views on software architectures. Shaw also defined software architectures based on four views [66]. Shaw named the views structural models, framework models, dynamic models and process models.

Although the terms are different, they actually provide us with different ways to analyze and understand software architectures. For the sake of consistency, we use views in this dissertation. The views based on the previous work mentioned above are summarized in this section.

The conceptual, or logical, architectural view defines a picture of the functional requirements of a system. It is a useful tool for the communication between system analysts and domain experts because it does not refer to any implementation decisions and instead focuses on the fact that software architecture consists of components, connections between those components, plus the configuration of whole system, architectural styles, constraints, semantics, rationale, and systematic properties. An informal representation for the view traditionally has been the block-line diagram. But currently the formal representation is a software architecture description language that facilitates the description of an architecture's components and connectors.

The module, or development, view is a product of refining the functional requirements to functional modules. It can be considered as the organization of actual software modules or subsystems that could include some modules with closed relationships and the interdependencies among the modules and the subsystems. Its primary characteristic is the tight coupling to the implementation. For a system architecture, the modules or subsystems are the usable candidates for components, while the interdependencies are the proper candidates for connectors.

If the conceptual and module views are considered to be utilized for representing the static aspects of a system, the process, or coordination, view emphasizes the runtime behavior of the system. Since the dynamic properties of a system are usually processes,

the view can also be called a process model of the system. Therefore the structural components and connectors in the view appear as processes. The view is useful to serve as a tool for evaluating, analyzing and managing the system's performance.

The physical view serves as the mapping of software onto hardware. If a hardware platform is chosen, software must be partitioned into processes that are executed on the platform. Whether the platform is a centralized computing environment or distributed computing environment, since physical configurations can vary depending upon the system is in test or deployment, the mapping of the software onto the hardware needs to be flexible and should have as little impact on the actual code as possible. It focuses on the real construction of the architecture, and the steps involved in the construction.

The framework view focuses on the structure of the whole system and often refers to specific domains or problem classes. For example, the framework view may be domain-specific software architectures, CORBA or CORBA-based architecture models and domain-specific component repositories. In fact, the framework view is a concrete application of a conceptual view to specific domains.

Clearly the views categorized as static views include conceptual view, module view and framework view, and dynamic views include process view and physical view. Each of these views provides a different perspective of same system. The conceptual view, module view and framework view show the static system structure, while the process and physical views give us the dynamic or runtime system structure. Although the views represent different system structural perspectives, the elements of one view have potential correspondences to the elements of other views. These views do not exclude each other. Put together, they are a consensus view of software architecture.

The views are very useful for domain experts, system designers and system implementers. The different-level views can supply them with communicating and understanding guidelines to finally construct a software system.

3.4 Architecture Description Languages

Usually system analysts use a diagram with boxes and lines to informally represent software architectures such as the two diagrams in Figure 3.1. The diagrammatic representation sometimes does not make sense and is very ambiguous because the boxes and lines cannot express the overall architectural properties, such as the semantics of components and connectors, their properties and the behavior of the whole system [10]. In addition, software architecture based development requires formal modeling notations, analysis and development tools that operate on architectural specifications [49]. If we can semantically describe a software architecture, we are actually close to reaching our goal: the automatic composition of a system from components. This section is designated to survey the issues related to requirements for architecture description languages (ADLs), candidates for ADLs, and what is an ideal ADL. The ADLs are actually designed for software architectures. Therefore an ADL should be comprehensive enough to cover everything involved in the definition of software architectures. The definitions surveyed in Section 3.1 directly suggest what must be covered in an ADL. In other words, what are the requirements an ADL must satisfy. First of all, we analyze the requirements.

3.4.1 Requirements for ADLs

ADLs are formal languages used for depicting or describing the architecture of a software system. As discussed before, an architecture for a software system consists of components, connectors representing the interactions among the components, and the configuration of the system. It is clear that a software architecture is a high-level abstraction of structural and behavioral issues related to a system, and does not focus on the algorithmic details of computations. The description of a software architecture serves as a bridge between a system design and system implementation based on architectural styles. Thus an appropriate ADL plays an important role in architecture based development. In order to design an ADL, it is absolutely necessary to understand the requirements of ADLs. This section emphasizes the requirements. The requirements are introduced here in terms of what an ADL should do.

- Describe the high-level structure of the overall application rather than the implementation details of any specific source module [77].
- Explicitly model components, connectors and their configurations [49].
- Provide tool support for architecture-based development and evolution [49].
- Describe the semantics of components and connectors [64] [49].
- Have appropriate syntactic and semantic elements for software architectures. The elements could be components, operators that are interconnection mechanisms, or patterns that are compositions in which code elements are connected in a particular way [64].
- Support types of elements and constraints for a system or parts of it [49].

- Support reusability of components, connectors and architectural patterns; support heterogeneity for combining multiple and heterogeneous architectural descriptions [64].

This requirements list illustrates the various aspects of what constitutes an ADL. Medvidovic and Taylor presented a definition for ADLs. The definition says that an ADL must describe components, connectors and their configurations and supply tool support for architecture-based development and evolution [49].

3.4.2 Classification Framework for ADLs

In addition, according to the definitions of software architectures in Section 3.1, an ADL should also describe the constraints on components, connectors, and the whole systems, and types of components and connectors. Medvidovic and Taylor present a classification framework [49] for ADLs introduced here in Table 3.1. Although the existing ADLs are in progress, the framework can be considered as a tool to evaluate them.

Is there any ADL that meets the requirements? It is hard to answer the question because the proposed ADLs at least partially meet the requirements. Rapide [42] models component interfaces and their externally visible behavior. Wright formalizes the semantics of architectural connections and architectural styles [2]. The UniCon system can compile architectural descriptions and modules into executable code [52]. The Aesop System emphasizes the explicit encoding and uses a lot of architectural styles [20]. Various domain-specific software architecture languages support architectural specification referred to a specific application domain [76]. Other ADLs have been proposed, for example, MetaH [78], LILEANNA [75], ArTek [74], C2 [50][48][47],

Darwin [43][44], and SADL [54], though they only satisfy partial requirements in the framework. This is not surprising because the development of ADLs is still in its infancy.

Table 3.1 Classification framework for ADLs (adapted from [49])

ADL
Architecture Modeling Features
Components
Interface
Types
Semantics
Constraints
Evolution
Connectors
Interface
Types
Semantics
Constraints
Evolution
Architectural Configurations
Understandability
Compositionality
Heterogeneity
Refinement and traceability
Scalability
Evolution
Dynamism
Tool Support
Active Specification
Multiple Views
Analysis
Refinement
Code Generation
Dynamism

Nevertheless, the previous work benefits us and helps us find what a comprehensive ADL must carry out. Based on the work of Medvidovic and Taylor [49], we use a table to show the architectural features of some existing ADLs (Table 3.2). In this table, only the architecture modeling features in the framework are shown. "√" is used to indicate that an ADL has that feature. "N/A" is used to indicate that we do not know if the ADL has that feature.

Table 3.2 ADLs with their architectural modeling features

Features \ ADL		Aesop	MetaH	LILE-ANNA	ArTek	C2	Rapide	Wright	Uni-Con	Dar-win	SADL	ACME
Components	Interface	√	√	√	√	√	√	√	√	√	√	√
	Types	√	√	√	√	√	√	√	√	√	√	√
	Semantics		√				√		√	√		
	Constraints	√	√	√	√	√	√	√	√	√	√	√
	Evolution	√				√	√					√
Connectors	Interface	√				√		√	√		√	√
	Types	√				√		√	√		√	√
	Semantics						√	√	√			
	Constraints					√		√	√			
	Evolution	√									√	
Configurations	Understandability	√	√			√	√	√	√	√	√	√
	Compositionality		√			√	√		√	√	√	√
	Heterogeneity	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	Constraints	√	√			√	√				√	
	Refinement and traceability						√				√	
	Scalability					√			√			
	Evolution	√						√			√	√
	Dynamism					√	√			√		

3.5 Summary

In this chapter, we surveyed a number of main topics in the fields of software architectures. They are definitions for software architectures, software architectural

styles, architectural views, and architectural description languages. These main topics are summarized in this section.

In Section 3.1, various definitions for software architectures are discussed. These definitions show a comprehensive framework for software architectures. The framework covers components, connectors, configurations or topologies of software architectures. Software architectural styles are surveyed in Section 3.2. In this section, different architectural styles are introduced based on previous work on software architectures. The observations on software architectures from different angles are called architectural views. A number of views are discussed in Section 3.3. Architecture description languages are one of the main research fields of software architectures. The related topics to ADLs are introduced in Section 3.4. The topics are the requirements for ADLs, a framework for ADLs, and the features the current existing ADLs have.

CHAPTER 4

COMPONENT-BASED SOFTWARE

Recently a paradigm has been emerging to develop software systems by using reusable building blocks. This paradigm is component-based software. Simply speaking, component-based software represents a development method for assembling software systems from components [57]. Sometimes it is also called component-based development, component-based software engineering, or componentware. Whatever it is called, it really makes the software community begin to shift the development focus from lines-of-code to coarser-grained components and their interconnections. A definition for component is "a unit of computation or a data store." [49]. However, this definition is too general to bridge the gap between the concept and component-based software. In this chapter, we survey the previous work on definitions of components and address other issues related to component-based software.

4.1 What Is a Component?

Components are used in many different ways and in many fields. The term, component, has been used for many years in the software community. Now the term is involved in component-based software. Obviously any discussion of the topic must answer the question, what is a component? We have collected some definitions for components.

Brown and Short define a component to be an independently deliverable set of reusable services [9]. This definition focuses on reusable services and independent delivery. Clearly the independent delivery guarantees the reusability of a component. It means that a component should be unaware of the context. A similar one to the

definition of Brown and Short is found in [37]. Krieger and Adler define software components as reusable building blocks for creating software systems. Such components feature themselves with semantically meaningful application or technical services. An important property implied in the two definitions is reusability. This property is also emphasized in [6].

A definition given in [5] simply indicates that a component is a fragment of a software module – a piece of a program, a subroutine, an object, one or more statements written in any language. It emphasizes the heterogeneity of components and the varied granularity of components. According to [1], components are characterized as active, computational entities of a system and they involve internal computation and external communication with the rest of the system in order to accomplish tasks. Bronsard et al define a component as a significant functional unit of a system with any granularity and reusability [8]. In addition, they emphasize the composite feature of a component.

The definitions explicitly indicate that a component is a unit of computation; its granularity is varied; it must be reusable, independent, and context-unaware; it has its intention and extension. The intention is its implementation or functional specification. The extension is its exposed interfaces for other components to access it. The revealed essences of a component are very important for further work on component-based software.

4.2 Requirements on Components in Software Libraries

Usually independently developed components are organized in a library called software library or component repository. It is called a catalog in [28]. Software libraries are

repositories where software components are stored and searched [51]. Jazayeri claims that components must meet four requirements if they are organized in software libraries so that the components can be reused widely [28]. The requirements refer to taxonomy, genericness, efficiency, and comprehension of components. They are discussed in the rest of this section.

First, components in a catalog must form a systematic taxonomy. This requirement is based on the experiences of some failed and successful existing software libraries. The software libraries must be organized to closely relate to well-defined domains of functionality, rather than collections of loosely related, or worse, unrelated, components. The components in a software library must be semantically classified into different catalogs such that each catalog supports a related set of concepts from a domain.

Second, components should be as generic as possible. A motivation for component development is to reduce the size and complexity of software systems. For a component library, it is better to use fewer components to support the same functionality than to use more components. To make it possible to have fewer components means that each component must be usable in several contexts, that is, it must make minimal assumptions about the context in which it is used. These kinds of components are generic components. A typical implementation for a generic facility is a C++ template. When a generic component is designed, we need to generalize all the common properties from the semantic-related specific components.

Third, components should be as efficient as possible. Basically a generic component is not an efficient one. So this requirement seems to contradict with the

previous one. However, when a generic component is used, it will be instantiated or specialized to meet a system's functional and performance requirements.

Fourth, catalogs must be comprehensive. A problem with a component catalog is that if a system designer will not find the needed components, he will stop using the catalog. Therefore, component vendors must try to do their best to keep the catalogs comprehensive.

4.3 Component Representation and Retrieval

The representation of components is necessary no matter where the components are. The components in an application architecture are represented in an ADL. However, if a component is located in a software library, how is the component represented? If the components are represented in a specific way, the retrieval of the components heavily depends on the way it is represented.

Mili et al. claim that the components must be represented as formal specifications that describe their functional properties [51]. Because such specifications may be arbitrarily abstract, they allow us to focus the description on those properties of the components that are most relevant in a retrieval operation. The formal specification is based on a logical description of the semantics of the software components. The formal specification form Mili et al use is the relational specification.

Zaremski and Wing also claim that the representation of software components must be formal specification [81]. They use Larch/ML [80], a Larch interface language for the ML programming language, to specify ML functions and ML modules. Larch provides a "two-tiered" approach to the specification [25].

Besides the representation of the semantic properties of a component, the syntactic properties and environmental properties need to be represented. The syntactic properties are interfaces' names, the number of arguments for each interface, and the data types of the arguments. The environmental properties include the working platforms, programming languages, and compilers.

Component retrieval is similar to database query. But here it means to get some software components from a software library. The key issue of component retrievals is the semantic match indicating that two components are semantically equivalent. This is why we need a formal specification for component representation.

Another issue is the retrieval algorithm. Zaremski and Wing use a theorem prover because their specification language is based on first order logic. Mili et al. classify the current algorithms into four broad families: AI-based algorithms; hypertext-based algorithms; library science/information science algorithms; and formal specification-based algorithms [51]. Mili et al. use a formal specification-based algorithm.

However, the algorithm Zaremski and Wing use is limited for the situation in which components are functions or modules (a collection of functions) and the algorithm Mili et al. use processes the situation in which components are functions. Hence, the algorithms need to be improved for any types of components.

4.4 Component-Based Development

Component-based development (CBD) has roots in modular systems, structured design, and most recently in object-oriented systems. The techniques aim at encouraging large systems to be developed and maintained easily by reducing the complexity of the original

problems. Component-based development actually extends the ideas, and moves the development focus from the programming of code to the assembly of an application from the well-defined pieces of code called components. In this way the software development consists of components selection, evaluation, and assembly processes where the components are acquired from a diverse set of sources, and used together with locally-developed software to construct a complete application [9]. Clearly this view on component-based software development is not complete enough because we cannot separate the paradigm from software architectures. Just as we have justified before, the independence of components and the dominant role of connectors of software architectures characterize the paradigm as a distinguished one from traditional paradigms. Therefore, when we examine a methodology of the paradigm, it will be judged primarily on these two characteristics. This is also why some proposed methodologies focus on the interdependencies among components. In fact, after an architecture is designed, component integration including component selection, evaluation and assembly is the key step for the success of a system.

In addition, component-based software is proposed to solve a software problem, that is, how is a software system designed and implemented from components? The components may be from different vendors, written in different parts of the world, implemented in different languages, run on different platforms, automatically searched and connected into one application. The "automatically" implies semantics-driven component search and integration. A solution for the problem will lead to an automatic production of software systems, even for distributed and heterogeneous applications. This is the ideal target of CBD, and also a difficult task. Currently there is no such

methodology to try to catch the ideal target. The methodology proposed in this dissertation partially meets the requirements. Most existing development methods focus on designing or selecting software architectures and proposing architecture description languages that are surveyed in Chapter 3. The primary motivation of these methods is to standardize architectural styles, the types of components and the types of connectors. In fact, the ADLs represent the main characteristics of the methods. Since the ADLs can be grouped into implicit configuration languages, in-line configuration languages and explicit configuration languages [49], the methodologies can be distinguished as different categories. Some of the methodologies are surveyed here.

Dellarocas proposes an approach for integrating software components in [15][16]. This approach is based on coordination theory that is used to model the interdependencies among components. The activities involved in the approach are almost fully consistent with our observations for integrating components. But the approach is weak on component integration based on arbitrary interconnections and semantics-driven component search.

Formalizing connectors is helpful for automatic semantics-driven component integration. Some related work has been discussed by Shaw et al. [67], and Abowd et al. [1][2]. This work is limited to some existing types of connectors. For example, UniCon only supports a number of built-in connectors. Furthermore, generating code from ASL leads to poor efficiency of software production.

A project conducted at Andersen Consulting, Component-Based Software Engineering (CBSE), focuses on integrating components [57]. The central part of the project is a component-based Architecture Specification Language (ASL). The ASL is

used to describe component interfaces, components, bindings and configurations. Based on descriptions, component-based software development takes a plug-and-play style. However, the methodology does not explicitly describe or abstract the interactions among components as first-class connectors. Such a drawback is caused by the methodology's component model. The component model classifies the interfaces of a component into provided interfaces and required interfaces [57] [8]. The provided interfaces are the services or capabilities that a component supplies to other components. The required interfaces are the services that a component has to receive from other components in order to accomplish its own functionalities. The model still couples the components tightly together, and they lack independence. This property contradicts the general objectives of component-based software.

4.5 Evaluation, Analysis and Management of Component-based Software

Although component-based software ideally and theoretically promises the software developers a bright future for building software systems, there is still skepticism on the development paradigm. As we surveyed before, component-based software development has two centers: component development and architecture design. Up to now there are no complete metrics for developers to evaluate, analyze and manage such a system, though there are some methods, such as scenario-based analysis for software architecture. In this section, a brief survey is given to address these evaluation and management issues.

4.5.1 What Problems Need to Be Solved?

Component-based development has obvious characteristics, such as the separation of component development and the whole system development even though the properties

of components in a system need to be described. Here we list the problems that need to be solved by metrics for components, whole systems and architectures.

Takeshita described some problems in [69]. The problems are around components and a whole system. Here they are simply introduced. The problems refer to metrics and risks of component-based software engineering. Vickers classified the risks of CBSE to five areas and indicated where metrics might be useful [79]. We summarize their conclusions here.

The following problems need to be solved when developing components. What measurements are to be used for the size and complexity of components? How to measure the size and complexity of components? What makes up the quality of a component? What measurements are to be used for quality of a component? How to find out the quality of components? How to estimate the workload to produce components for CBSE? How to price components for CBSE? How will developers and/or distributors charge their customers for the use of their components? What measurements should be used to record and display the total usage of a component?

For a component-based software system, a component-based software engineer needs to answer the following questions. What measurements are to be used for the size and complexity of a completed application? How to measure the size and complexity of completed application programs? How to estimate the workload to put together components to produce completed application programs? What determines the quality of a completed application program? How to find out the quality of a completed application program? How to price completed application products using CBSE? In addition, when a component-based software system is developed, what risks must be taken? What are

potential mistakes a system designer may make? Is there sufficient technical support? What are the difficulties in management in an enterprise? All these problems must be solved in order for component-based software engineering to be a mature discipline. Generally there are two types of risk knowledge, generic risk knowledge which applies to all CBSE projects, and project-specific risk knowledge [79]. They are:

- CBSE risk areas,
- CBSE risk management techniques,
- Risk severity ratings,
- Risk relationships,
- Risk consequences,
- Risk warning signs,
- Component usage profiles,
- Specific project risks,
- Risk monitoring criteria, and
- Risk status.

Moreover, there exist some problems around software architectures. A typical problem is how to obtain the expected system properties. The properties of a software system can be run-time properties and non-run time properties. The run-time properties include performance, behavior, communication patterns, and so on. The non-run time properties include maintainability, portability, reusability, adaptability, extensibility, scalability, ease of use, predictability and learnability that are also called quality

attributes. The run-time properties are addressed by many ADLs [12]. The non-run time quality attributes are analyzed with a proposed method, scenario-based analysis, in [30]. Scenarios are brief narratives of expected or anticipated use of a system from both development and end-user viewpoints.

4.5.2 Some Proposed Solutions

As can be seen in Section 4.5.1, component-based software faces many risks. This is why some developers and users are skeptical towards the component-based software development style. As the supporters of component-based software, we have to find solutions for the problems. It is necessary to propose a comprehensive methodology for component-based software engineering.

One of the solutions is the Risk Analysis and Management (RAM) model in [40]. RAM is a structured process for controlling risks. Risk analysis is supposed to address the identification of risks, the estimation of risks and the evaluation of risks. After the analysis, risk management figures out the selection and implementation of risk management techniques, and the monitoring of their effectiveness during the project. The results or knowledge from risk analysis are stored in a risk repository. The risk management will apply some techniques on the stuff in the repository. The techniques could be pre-risk, which means preventing risks before they become problems, or post-risk, which means figuring out the impact that the risks have.

To solve the problems around software architectures, a solution was proposed in [29]. The technique, deriving software architectures from quality attributes, sets the quality attributes of systems first and then derives a software architecture step by step

such that the architecture has the desired quality attributes. This is really a choice for a component-based software engineer.

4.6 Summary

A survey on component-based software is done in this chapter. The surveyed topics include the definitions for components, requirements of software libraries, component representation and retrieval, component-based software development, and the metrics of component-based software engineering. These topics are related to development methodologies of component-based software systems. The main contents of each topic are summarized in this section.

A number of definitions of components is surveyed in Section 4.1. Generally a component is defined as a unit of computation. Its granularity is varied. It must be reusable, independent, and context-unaware.

For software libraries, some requirements are summarized in Section 4.2. One of these requirements is a consistent systematic taxonomy for all components. The components in software libraries should be as generic and efficient as possible. A software library should be comprehensive.

Component representation and retrieval are surveyed in Section 4.3. It is indicated that a component in a software library must be represented in a formal specification language. For a designed component in an application, it also must be described in a formal specification language so that the behaviors of a component can be matched when it is searched from software libraries. Furthermore, the retrieval of components from

software libraries must be based on formal specifications so as to obtain an expected component. Signature match is inadequate to match the behaviors of two components.

Several existing methods for component-based software system development are explored in Section 4.4. These methods focus on architectural styles and architecture description languages. They are weak on system integration from varied components and connectors, and component search.

In order to propose a comprehensive methodology for component-based software system development, the necessary metrics for component-based software are surveyed in Section 4.5. A lot of problems around component-based software and software architectures are summarized. Besides the problems, some risks component-based software engineers need to evaluate and analyze are introduced. At last, a number of potential solutions is discussed.

CHAPTER 5

CYCLOMATIC COMPLEXITY AND CUBIC FLOWGRAPHS

Cyclomatic complexity has been utilized before in McCabe's work [45] as a tool for the evaluation of computer programs. Such a measure could be used in the estimation of the number of testing paths in a program, if the program should be tested for all the execution paths. Since a program can practically be represented as a directed graph (e. g. program control graph [45]), it is reasonable to apply cyclomatic complexity to programs because the concept, cyclomatic complexity, actually originated from the concept cyclomatic number in graph theory. When we study the cyclomatic complexity of programs, we find that if a program is converted to a cubic flowgraph, the cyclomatic complexity of programs can also be computed. The method implies a solid and theoretic way for the computation of cyclomatic complexity. In this chapter, the essence of the method is discussed.

On the other hand, cubic graphs are structures that can be decomposed into elementary cubic graphs. Utilizing this property, programs can be converted to cubic graphs and their structures can be analyzed with respect to the composition of the corresponding elementary cubic graphs. In addition to this potential use in structural analysis, other benefits of transforming programs to cubic graphs have been exploited. The inherent relation between specific cubic flowgraph nodes and the decision-making points in a program rendered the cubic graphs as a good candidate for calculating cyclomatic complexity.

The research achieved similar results to those of an existing method [45] for calculating the cyclomatic complexity. Also it is easy to observe the preservation of the

calculated complexity in the corresponding collection of elementary cubic graphs. Such a collection constitutes the decomposition of the program.

In this chapter, a brief survey of the existing method for calculating cyclomatic complexity is given. Cubic flowgraphs and computing cyclomatic complexity through cubic flowgraphs are explained. Finally the decomposition of cubic flowgraphs and their relation to cyclomatic complexity are demonstrated.

5.1 Cyclomatic Complexity of Programs

McCabe introduced a way to calculate cyclomatic complexity of a program [45]. This method is especially valuable for structured programs [55]. A structured program is assumed to have one entry point and one exit point. In order to test a program, the number of paths included between the entry and the exit points needs to be known. Every path may need to be tested. McCabe has also indicated in [45] that, it is possible to determine the correctness of a program if a limited number of paths is tested. Each program has a limited number of primary paths and other paths are a linear combination of the primary paths. The details of calculating cyclomatic complexity based on the mentioned method are explained below, considering structured programs.

Every program can be represented by a flowchart. A flowchart can be converted into a directed graph, called a program control graph by McCabe. The corresponding directed graph has two distinguished nodes: an entry node and an exit node. Every node in this directed graph can be reached from the entry node and every node can reach the exit node. Nodes in this graph either represent a decision-making condition or a statement. If an imaginary edge from the exit node to the entry node is added to this graph, a strongly connected directed graph is obtained. Representing programs as graphs

helps in the application of graph theoretical aspects to obtaining software engineering tools. One such tool is the cyclomatic number of a graph. If v(G) is used to represent the cyclomatic number of a graph G, then

$$v(G)=e-n+p,$$

where e edges, n nodes and p connected components are included in the graph G.

Based on McCabe's work [45], the cyclomatic number of a strongly connected graph G is equal to the maximum number of linearly independent circuits. From the linearly independent circuits, the independent primary paths of a strongly connected graph can be produced. The idea can be demonstrated in an example in the form of a program shown in Table 5.1. Figure 5.1 shows a strongly connected directed graph G converted from the program in Table 5.1. The statements in the program are a, b, c, d, e, f, g, h, and x. The decision-making conditions are A, B, C, and D.

Table 5.1 A program example

```
program sample();
begin
    a;
    if A then
          b;
    else
          c;
    d;
    while B do
        begin
        repeat
            e;
            f;
        until C;
        if D then break;
              g;
              h;
    end
    x;
end
```

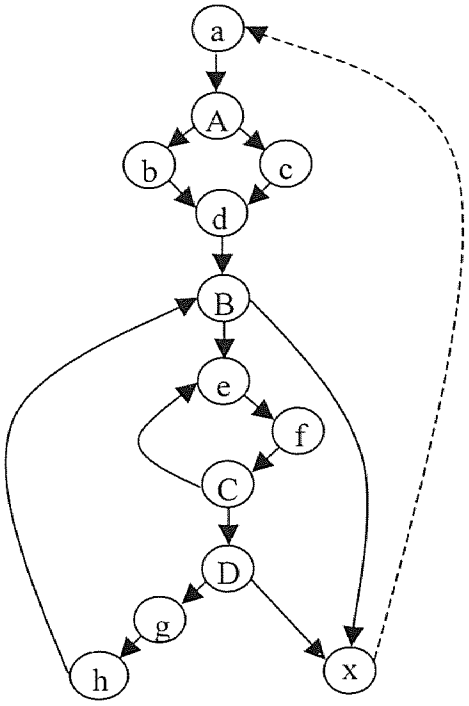

Figure 5.1 A program control graph

The cyclomatic number of G is v(G)=17−13+1=5 because there are 17 edges including the edge from the exit node to the entry node, 13 nodes and only one connected component in G. So the maximum number of linearly independent circuits is 5. There are four linearly independent circuit sets listed in Table 5.2 where the size of each set is 5.

Table 5.2 Independent circuit sets

Name	Circuit
B1	AAbdBxa, aAcdBxa, efCe, BefCDghB, aAbdBefCDxa
B2	AAbdBxa, aAcdBxa, efCe, BefCDghB, aAcdBefCDxa
B3	AAbdBxa, efCe, BefCDghB, aAbdBefCDxa, aAcdBefCDxa
B4	AAcdBxa, efCe, BefCDghB, aAbdBefCDxa, aAcdBefCDxa

Therefore, any one of these sets (B1, B2, B3, or B4) is a primary basis for all circuits in the graph G and paths through G. The circuits and paths in the graph can be

expressed as a linear combination of circuits in any one of these sets. For example, if we take B1 as basis, the circuit (aAcdBefCDxa) can be expressed as (aAbdBefCDxa)-(aAbdBxa)+(aAcdBxa). An algorithm for computing cyclomatic complexity of a program can be found in [73]. Based on the linearly independent circuit sets, there are corresponding linearly independent program path sets as listed in Table 5.3.

Table 5.3 Independent program path sets

Name	Path
PB1	aAbdBx, aAcdBx, aAbdBefCefCDx, aAbdBefCDghBefCDx, aAbdBefCDx
PB2	aAbdBx, aAcdBx, aAcdBefCefCDx, aAcdBefCDghBefCDx, aAcdBefCDx
PB3	aAbdBx, aAbdBefCefCDx, aAbdBefCDghBefCDx, aAbdBefCDxa, aAcdBefCDxa
PB4	aAcdBx, aAbdBefCefCDx, aAbdBefCDghBefCDx, aAbdBefCDxa, aAcdBefCDxa

After the maximum number of linearly independent paths of a program is known, program testing can be done based on the primary paths. The maximum number of linearly independent paths is referred to as the cyclomatic complexity of a program, v(G). Detailed information can be found in [45] for cyclomatic complexity, with the basic set of properties given below:

1) v(G) is equal to or greater than 1.

2) Inserting or deleting functional statements to G does not affect v(G).

3) In fact, v(G) is just determined by decision-making conditions and v(G)=n+1, where n is the number of decision-making conditions in a program.

5.2 Computing Cyclomatic Complexity Based on Cubic Flowgraphs

Every program can be converted to a cubic flowgraph. A cubic flowgraph is a strongly connected directed graph. The cyclomatic number of a cubic flowgraph can be computed similar to that of a normal strongly connected directed graph. But a cubic flowgraph is a

special strongly connected directed graph. Based on the properties of cubic flowgraphs, the following theorem is used for computing the cyclomatic number of a cubic flowgraph.

Theorem 1

If a cubic flowgraph G has 2n(n=0, 1, 2, ...) nodes, then its cyclomatic number is:

$$v(G) = n+1.$$

Proof:

It is assumed that e edges and m nodes are in a cubic flowgraph. A cubic flowgraph is known to be a strongly connected directed graph. Consequently the number of connected components in the graph is 1. Cyclomatic number of G has been defined in [45] as:

$$v(G)=e-m+1$$

On the other hand, the graph is a cubic flowgraph with 2n nodes. So it must have 3n edges meaning that e=3n and m=2n. Then

$$v(G)=3n-2n+1$$

and finally,

$$v(G)=n+1.\blacksquare$$

Remarks:

Generally the n in a cubic flowgraph can be interpreted as the number of decision nodes. Every decision node corresponds to a junction node. For n decision nodes there are n junction nodes, accounting for the total 2n nodes. The application of Theorem 1 to the program in Table 5.1, whose cubic flowgraph is shown in Figure 5.2, yields v(G)=5.

The result verifies Theorem 1. Therefore Theorem 1 has a corollary concluded as below.■

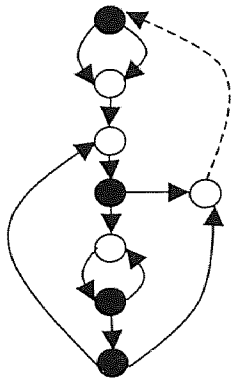

Figure 5.2 A cubic flowgraph of the program in Table 5.1

Corollary 2

The cyclomatic number of a cubic flowgraph is the number of decision nodes plus one. ■

Remarks:

The corollary suggests a simple way to compute the cyclomatic number of a cubic flowgraph. If we have a cubic flowgraph, we do not need to count how many nodes and arcs in it; we just need to count how many decision nodes it has. Then the cyclomatic number of a cubic flowgraph is the number of decision nodes plus one. ■

As discussed before, each program corresponds to a cubic flowgraph and the flowgraph does not change the semantics of the program. Thus the cyclomatic number of a cubic flowgraph is also the cyclomatic complexity of its original program. This is concluded in Corollary 3.

Corollary 3

The cyclomatic complexity of a program is equal to the cyclomatic number of its cubic flowgraph. ■

Remarks:

When the cyclomatic complexity of a program is computed, Corollary 3 supplies us with an easy way to do that. Clearly converting a program to a cubic flowgraph is easier than converting it to a program control graph. Therefore, this method is better than the previous ones. ■

5.3 Computing Cyclomatic Complexity Based on Decomposition of Cubic Flowgraphs

In this section, prime cubic flowgraphs and the relationship between the prime cubic flowgraphs of a program and the cyclomatic complexity of the program are discussed. Prime cubic flowgraphs are the results of decomposition of cubic flowgraphs. We have concluded that there are three prime cubic flowgraphs, P_0, P_1 (two forms) and P_2 as shown in Figures 2.4 and 2.5.

Therefore, a conclusion is reached that every structured program can be decomposed into prime cubic flowgraphs. A relationship between the decomposition of a program and its cyclomatic complexity is concluded in Theorem 4.

Theorem 4

The cyclomatic complexity of a program is

$$v(G)=N_1+2*N_2+1,$$

where N_1 and N_2 are the numbers of P_1 and P_2 in a program respectively.

Proof:

In the calculation of cyclomatic complexity of a program, P_0, P_1 and P_2 in the decomposed collection of prime cubic flowgraphs contribute 0, 1 and 2 respectively to the complexity. Let N_0 be the number of P_0s in the program. Let N_1 be the number of P_1s in the program. Let N_2 be the number of P_2s in the program. If the cubic flowgraph of the program has 2n nodes, then

$$n=N_0*0+ N_1*1+ N_2*2.$$

In fact,

$$n=N_1+ N_2*2.$$

From Theorem 1,

$$v(G)=n+1.$$

Therefore,

$$v(G)=N_1+N_2*2+1. \blacksquare$$

Remarks:

The theorem can be very useful for a large program if the cyclomatic complexity of the program needs to be calculated. The decomposition of the program in Table 5.1 suggests as an example that $N_0=1$, $N_1=2$ and $N_2=1$. Thus $v(G)=5$. The result can be verified by the decomposition of the cubic flowgraph in Figure 5.2. The decomposition results are shown in Figure 5.3. ■

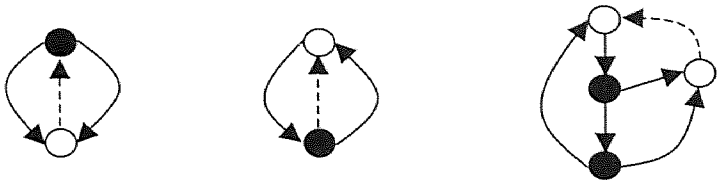

Figure 5.3 The decomposition results of the cubic flowgraph in Figure 5.2

Any program can be represented by a flowchart. A flowchart can be converted to a cubic flowgraph, by following the given rules. When the graph concept "cyclomatic number" is applied to a cubic flowgraph, the same results are achieved as McCabe did in [45]. Besides the conclusion in this theorem, we note that each flowgraph has and only has one prime flowgraph, P_0. As the result of this observation, the 1 in $v(G)=N_1+N_2*2+1$ can be interpreted as the number of P_0. If the mixed control structure corresponding to prime cubic flowgraph P_2 is removed and replaced by standard program control structures, then there are only two prime cubic flowgraphs, P_0 and P_1. Under this situation, $v(G)=N_1+1$ according to Theorem 4. Clearly the v(G) is really the number of prime flowgraphs. Therefore, the result can be concluded as a corollary of Theorem 4.

Corollary 5

The cyclomatic complexity of a standard structured program is equal to the number of prime flowgraphs in the program. ■

Remarks:

The corollary is a very useful result if a real structured program is decomposed into prime flowgraphs. For example, before we are going to decompose a flowgraph of such a program, we can know how many prime flowgraphs are included in it through the computation of its cyclomatic complexity. The corollary is applied in our methodology for system, subsystems and connectors decomposition (see Chapter 7). ■

5.4 Summary

This chapter explored the relationships between cubic flowgraphs and cyclomatic complexity of computer programs. The research results are concluded in Theorem 1, Corollary 2, Corollary 3, Theorem 4 and Corollary 5. These conclusions suggest a new way to compute cyclomatic complexity of computer programs.

Based on cubic flowgraphs, the cyclomatic number of a program is equal to the cyclomatic number of its cubic flowgragh, while the cyclomatic number of a cubic flowgragh is the number of decision nodes plus one. For a standard structured program, its cyclomatic complexity is equal to the number of prime flowgraphs in it. The results can be used to analyze a program and the entities with similar structures as programs, such as software system architectures.

CHAPTER 6

FUNDAMENTALS FOR THE METHODOLOGY

The survey of software architectures and component-based software in Chapters 3 and 4 suggests development steps for component-based software engineering. Roughly speaking, the steps consist of software architecture design, component search, and system integration. In fact, architectural design is to define desired components, connectors for the interactions among the components and configurations. Since the implementation details of components are not considered, we just need to design the details for connectors. Therefore the integration methodology is used immediately after the detailed design of connectors. On this point, what a system needs is the implementation. The methodology focuses on the implementation of a component-based software system. It is based on two fundamentals, an aggregation view on software architectures and a component model. Thus Section 6.1 provides the view, classifies the aggregations into system aggregations, subsystem aggregations and connector aggregations based on the view and demonstrates the view by a number of problem examples. Since the components are the elementary functional units in component-based software system, Section 6.2 introduces a component model for component search from software library. Section 6.3 summarizes our view and component model.

6.1 An Aggregation View on Software Architectures

For a large system, when it has been designed, the system traditionally consists of some subsystems, while a subsystem consists of modules. For example, Figure 6.1 is a typical system level diagram. The lines in the diagram mean containing, such as, a system

contains subsystems, and subsystems contain modules. In a similar way, a component-based software system can be divided into subsystems, but a subsystem actually consists of atomic or composite components. In this section, an aggregation view on software architectures is introduced. Then a number of problem examples are used to verify the view.

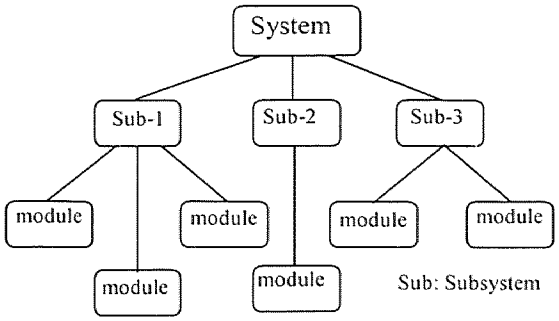

Figure 6.1 A system architecture based on subsystems and modules

6.1.1 Aggregation View

In a component-based software system, the functionalities are distributed among components according to the architectural views. A component can be a function, a module, a package, or an entire system. Generally a component-based system consists of components and the interconnections among components. The interconnections are abstracted as connectors. A component could be atomic or composite. A composite component consists of other components connected by at least one connector. Hence, a traditional subsystem or system can be considered as a composite component in terms of component-based style.

Moreover, architecture-based system design focuses on the interconnections and the functional descriptions of components instead of the details of components. For

example, the diagram in Figure 6.2 shows a typical structure of a component-based system. The structures of a component-based system may not be hierarchical if one functional unit is contained in more than one other functional unit. In this dissertation, it is assumed that when a structure of a component-based system is drawn, a functional unit is allowed to be repetitive. This assumption makes the structures hierarchical.

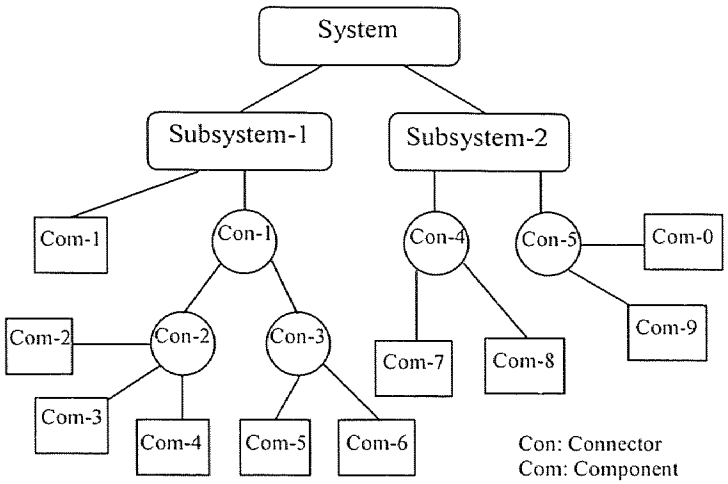

Figure 6.2 A structure of a component-based system

The hierarchical structures of component-based systems illustrate that all the leaves are components. These components can be called atomic ones because they are the results of architecture design and are searched from software libraries. For such a particular system, we do not care that they are originally atomic or composite in software libraries. The other nodes in the hierarchical structures are connectors, subsystems and system. These nodes can also be considered as components according to the general definition of a component. In fact, for a particular system, they are absolutely composite components. For example, a decomposition of the structure in Figure 6.2 produces a number of composite components shown in Figure 6.3.

Clearly a composite component at least includes a connector that connects a number of components. For a system like the example in Figure 6.2, a subsystem is a composite component, while an entire system is also a composite component. The system and subsystem nodes serve as connectors. For the sake of convenience, we call such a composite component an aggregation. The aggregations are a natural decomposition of system's functionalities. Such decomposition suggests a way for the final system integration. In this way, the system integration starts with the implementation of the lowest level, then the higher levels, and finally up to the highest level. As the example in Figure 6.3, we implement the aggregations 5, 6, 7, and 8, then aggregations 4, 3, and 2, finally the aggregation 1.

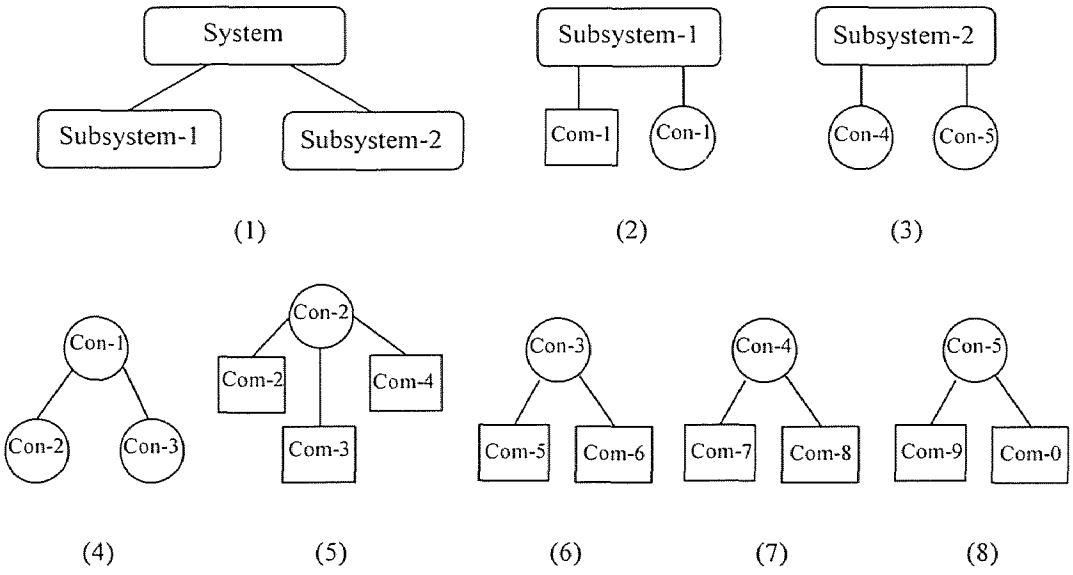

Figure 6.3 A decomposition of the structure in Figure 6.2

Such aggregation relationships are helpful to control the complexity of a system because the system can be decomposed into pieces in terms of the relationships. For example, the system in Figure 6.2 contains eight such relationships and can be

decomposed into eight pieces shown in Figure 6.3. Obviously the aggregations can be on system level, on subsystem level or on connector level. Therefore, the aggregations can be classified into system aggregations, subsystem aggregations and connector aggregations in terms of their root nodes. Every aggregation could include an atomic component. We define this kind of architectural view as aggregation view. Our methodology makes use of the view to decompose a software architecture into various levels aggregations.

We have discussed architectural views in Section 3.3. Aggregation view, in reality, is another expressive way for module view because the module view is the results of refining the functional requirements to functional modules. The architectures under this view are tightly coupled to implementation. Using aggregation view means a natural division of a system architecture. The divided units under aggregation view can be de facto considered as functional modules. The granularity of the modules can be a system, a subsystem, or a connector.

6.1.2 Demonstration by Examples

Now two real problem examples, the Key Word in Context (KWIC) problem and the Embedded Cruise Control (ECC) problem, are examined under the aggregation view. We use the examples to demonstrate the architectural view used in the integration methodology. The two examples are also used through the rest of this dissertation for the purpose of demonstrating the integration methodology.

The KWIC problem was proposed by Parnas [59]. He stated the problem as follows:

> The KWIC [Key Word in Context] index system accepts an ordered set of lines; each line is an ordered set of words, and each word is an ordered set of characters. Any line may be "circularly shifted" by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

Before the problem is examined in terms of aggregation view, the requirements analysis based on component-based software style must be done. The problem statement suggests that the input and output of a desired system are a set of text lines. The set of text lines is the data elements that need to be processed. The main processing elements are to circularly shift a text line word by word and to sort all the lines in alphabetic order. In addition, we should consider how to process the input and output of the system.

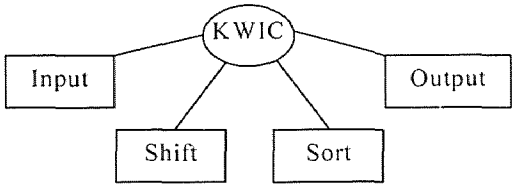

Figure 6.4 A component-based system structure for KWIC

Considering the independence and reusability of components, the input and output should be separated from the main processing elements. Therefore, for this system, it has four components, input, circular shift, sort and output. The next problem is how to coordinate the components in order to complete the tasks. Generally the coordination among components is abstracted as connectors. For this system, an architecture is shown in Figure 6.4 in which the node KWIC serves as the connector. One connector is used to

coordinate the four components. This design is an intuitive and natural one. It just focuses on solving the problem.

The KWIC is a relatively small system. The whole structure can be considered as one aggregation. The aggregation includes four components. The aggregation structure can be implemented in varied ways according to different component types and connector types [64] [15]. On the other hand, considering the enhancement for reusability of potential components, especially some new composite components, the above aggregation structure can be improved to be a two levels structure shown in Figure 6.5.

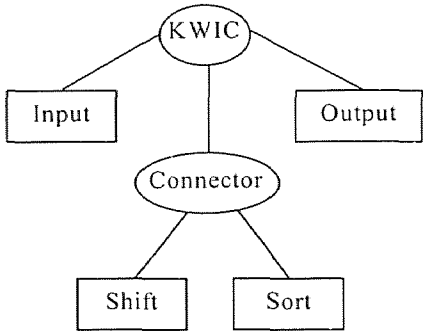

Figure 6.5 Another component-based system structure for KWIC

This new structure contains a composite component in which a connector connects the two components, shift and sort. In this situation, the new composite component may be reused elsewhere. The structure in Figure 6.5 contains two aggregations, one with the root node, KWIC, and the components Input and Output, and a connector, Connector, and another with the root node, Connector and the components Shift and Sort. Obviously the structure in Figure 6.4 and the structure in Figure 6.5 represent different design decisions. In other words, different design decisions will lead to different system structures, and perhaps different sets of aggregations. But one thing is common. It is that all the leaves

in such a structure are components. Those components are the results of primary design and need to be searched.

Table 6.1 The definitions for inputs and output of the ECC system

Input	System on-off	"System on" means that the ECC maintains the speed of the car.
	Engine on-off	"Engine on" means that the engine is running. The ECC system can only be active if the engine is on.
	Wheel pulse	A pulse per revolution of the wheel.
	Accelerator	A value indicating how far the accelerator has been pressed.
	Brake	When brake is pressed, it is on. The ECC system releases speed control when on.
	Increase/decrease	Increase or decrease the maintained speed such that the ECC holds at the new speed.
	Resume	The ECC system resumes the last maintained speed. It is applicable only if the ECC system is on.
	Clock	Timing pulse per millisecond.
Output	Throttle	A value for the engine throttle setting.

As an example, the cruise control system has been used in many software engineering books. Tanik and Chan used it to demonstrate informally the inception and specification of a system that results in a formal representation of the design abstractions [71]. They describe the following problem:

> The ECC system controls the speed of a vehicle equipped with an electronic fuel-injection system; it regulates the speed of the vehicle to a desired value as long as the speed remains uninterrupted by the pressing of the brake pedal; the system is operational only if it is enabled by the on state of the system's on-off switch and only if the engine is running;

when the system is switched on, it immediately begins maintaining the current speed unless it reacts to the pressing of increase or decrease switches or of the accelerator pedal.

The inputs for the ECC system are System on-off, Engine on-off, Wheel pulse, Accelerator, Brake, Increase/decrease, Resume and Clock. The output of the ECC is Throttle. The definitions of the inputs and output are summarized in Table 6.1.

To solve the problem, we need to analyze the requirements, and then refine the requirements to the system's functionalities. If we use a component-based software style, the functionalities are distributed to several components. Tanik and Chan elaborated the problem statement to a data flow diagram in order to analyze the functionalities of the ECC system [71]. It is adapted here as shown in Figure 6.6.

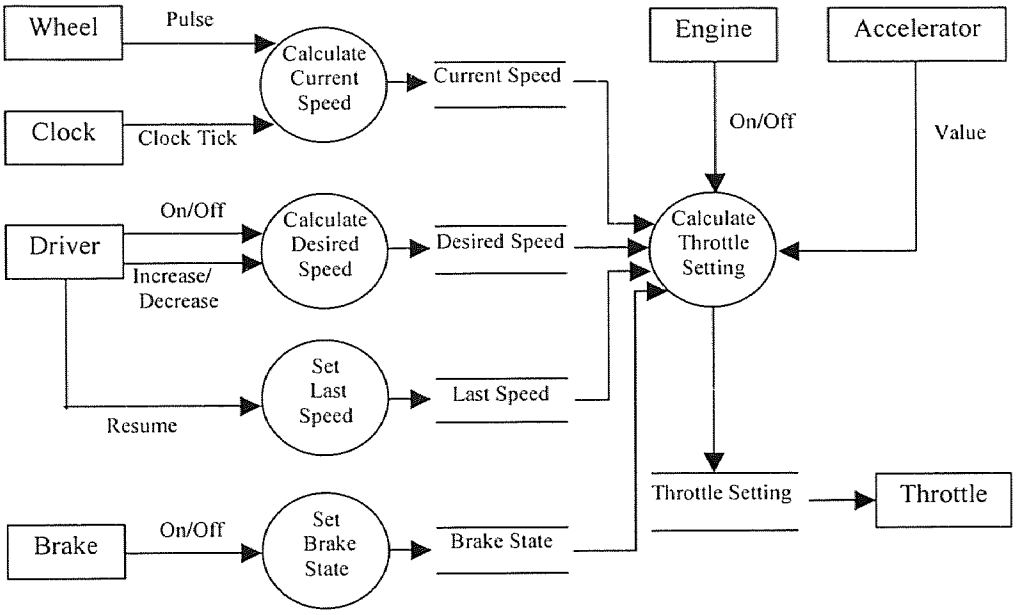

Figure 6.6 ECC data flow diagram (adapted from [71])

The data flow diagram of the ECC system implies five processing elements, that is, Calculate current speed, Calculate desired speed, Set last speed, Set brake state, and Calculate throttle setting. For each processing element, it has its own inputs and output.

The five processing elements can be intuitively considered as components. Since the inputs and output of each component are simple values, we are not going to design them as separate components. Instead, they are supposed to be prepared by connectors. Such a design decision produces a component-based software architecture as shown in Figure 6.7. This is a preliminary design. The design details are discussed in Chapter 7. However, it needs to be indicated that the whole system is executed only when the ECC system is on.

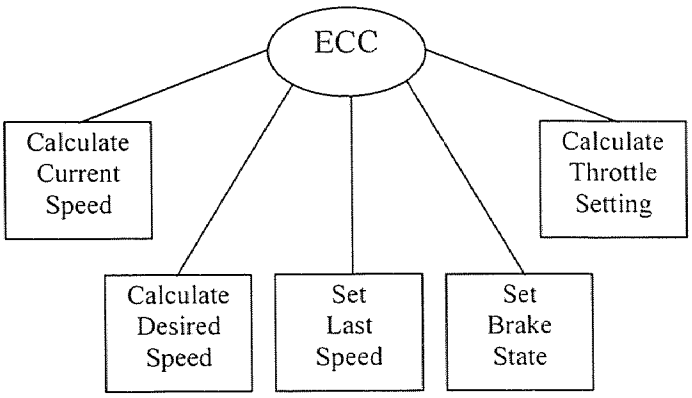

Figure 6.7 A component-based system structure for ECC

The implementation of the connector in the ECC architecture is relatively simple or complex dependent on what coordination protocols are used. The data flow diagram indicates that the four components in this connector aggregation can be executed synchronously or asynchronously. A properly selected protocol for the interactions is a key factor for the system. On the other hand, the interactions can be implemented sequentially. Perhaps this is a way to simplify complex situations.

The solutions here for these two problems are preliminary designs. After the types of all the components are chosen, the detailed designs for the problems need to be

explored. In fact, such detailed designs for component-based systems are the designs of connectors. Generally architectural styles are selected. Then coordination protocols for connectors are determined. The final design results depend on the combinations of architectural styles and the coordination protocols. Those considerations for the two problems are discussed in Chapter 7.

6.2 Component Model

We have to propose a component model in order to integrate a system from components. This component model is used for the component search. It is based on the previous work surveyed in Chapter 4. A component generally consists of interface and implementation while only the interface of a component is exposed to the outside or a client of a component. In order to search for a required component in a software library, the interface needs to be defined as delicate as possible. A component interface consists of type, methods, events, properties, constraints and semantic specifications for methods and events' methods. The type information in this model indicates what kind of a component is. For example, the component is a server or a filter. The methods and events reflect the functional services provided by a component. The specifications embody the behaviors of the methods and events. The properties show the clients of a component what are the runtime contexts of the component. The constraints express some conditions that need to be satisfied when a component is invoked. The component model we propose is shown in Figure 6.8.

In the methodology we propose an implementation description language. Besides the description of aggregations, all the components involved in a system are also

described based on the component model. The descriptions of components are employed for search.

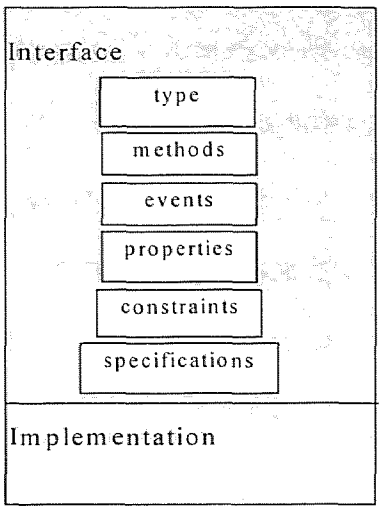

Figure 6.8 A component model

The component model is a comprehensive one because it covers all the required features of an ordinary component. If a software library is to be built, each component in the software library must consist of these features. It is a basic requirement for a software library because users of components always want to know everything about a component. On the other hand, if a component is designed in an application, it is not necessary for the designer to explore all the features in the model. A required component in an application needs to be searched. Therefore, the information collected for component search is the type, methods, events and their semantics that is represented in a formal specification language. When an expected component has been obtained from a software library, all the information for the component included in the model is provided by the vendor of the component.

6.3 Summary

We conclude that a component-based system is a set of aggregations and components. These aggregations belong to three levels: system aggregation, subsystem aggregation and connector aggregation. The implementation of system aggregation is based on subsystem aggregations. The implementation of subsystem aggregations is based on connector aggregations. The implementation of connector aggregations is based on components, while the components need to be searched from software libraries or to be bought from component vendors. The relationship among the aggregations is shown in Figure 6.9 which looks like an aggregation pyramid. Our work is to propose a methodology for constructing such a pyramid.

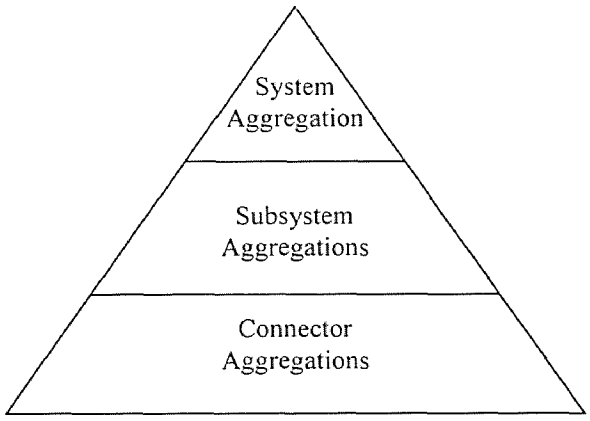

Figure 6.9 Aggregation pyramid of software architectures

Our methodology begins to build the base of the pyramid, then the middle layer, and finally the top. In other words, the implementation of a whole system starts with the component search, the implementation of connector aggregations, then subsystem aggregations, and finally system aggregation. Each aggregation corresponds to a

flowgraph that is used to represent the implementation details for it. The flowgraph representation is discussed in Chapter 7.

It is noteworthy that the decomposition of software architectures to aggregations takes advantages of component-based software because an aggregation is actually a functional unit. From the decomposition, we can really see the shadow of functional decomposition in traditional structured analysis and design. However, component-based software engineering gets rid of considering the algorithms and data structures of components. In this way, we only pay attention to the implementation of connectors.

Based on our approach for system integration, a general component-based software development methodology could consist of requirements analysis, architecture design, architecture description, architecture decomposition to components and aggregations, component search, aggregation construction, system test and system maintenance. The development model is shown in Figure 6.10.

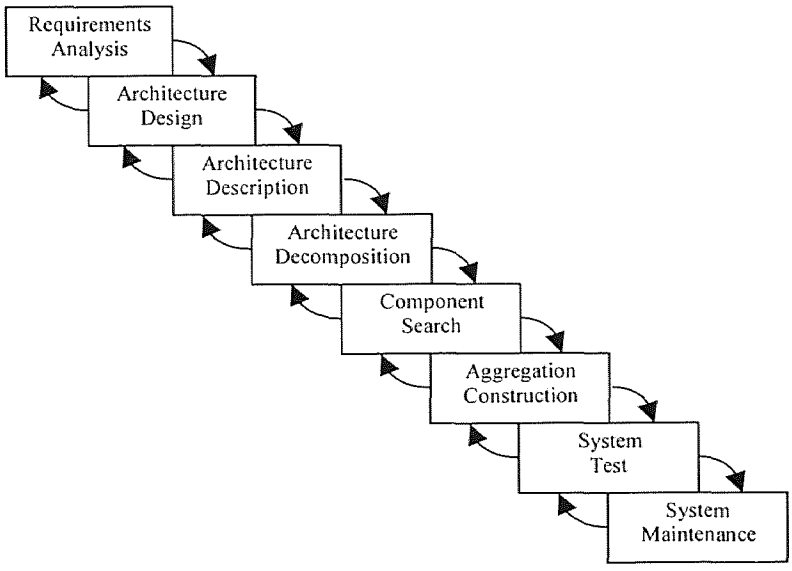

Figure 6.10 A development model for component-based software systems

The development model for component-based software system implies an important advantage, that is, handling requirements changes more easily. The reason is that the elementary functionalities of a system are distributed among the components. If some functionalities have to be changed, only the corresponding components are replaced. If some new functionalities need to be added, we search some new components to meet the requirements and insert them into the system.

CHAPTER 7

SYSTEM REPRESENTATION AND DECOMPOSITION

In the previous chapter, an aggregation view was proposed. We claim that our methodology for system integration would be based on this view and is applied after the detailed system design. In order to generate an executable system automatically or semi-automatically from the detailed designs, each aggregation needs to be represented in a language. As the result of detailed design, every aggregation corresponds to an implementable algorithm, even if the algorithm is simply composed of several assignment statements. The algorithms embody the internal interaction relationships among the parts in aggregations. Moreover the algorithms are actually the implementations of connectors according to our aggregation view.

In addition, our methodology suggests that the algorithms be expressed with structured programming style. In other words, the three program control structures are used to express the detailed design of the algorithms. The graphical language used for the representations of the algorithms is Jackson Diagram [27]. The concrete steps in the methodology are to represent the algorithms by using Jackson Diagrams and then to convert the Jackson Diagrams to structured flowgraphs that were introduced in Chapter 2.

We employ Jackson Diagrams since they are generally built on aggregations of related modules and each aggregation in it is explicit. Jackson Diagrams are also represented based on the three program control structures. We use the structured flowgraphs since each structured flowgraph can be decomposed into some prime

flowgraphs of a sequential, selective or repetitive structure. The structured flowgraph of an aggregation is called an aggregation flowgraph.

In Section 7.1, we discuss how to convert a Jackson Diagram into a structured flowgraph; how to decompose an aggregation flowgraph is discussed in Section 7.2. The Jackson Diagram representation, the conversion of Jackson Diagrams to flowgraphs and the flowgraph decomposition are demonstrated in Section 7.3 with the two problem examples, KWIC and ECC.

7.1 Conversion of Jackson Diagrams to Flowgraphs

Aggregations are classified into three kinds, system aggregations, subsystem aggregations and connector aggregations. After the decomposition of a software system architecture to aggregation sets, the detailed design of each aggregation is represented as a Jackson Diagram. Then Jackson Diagrams are converted into structured flowgraphs.

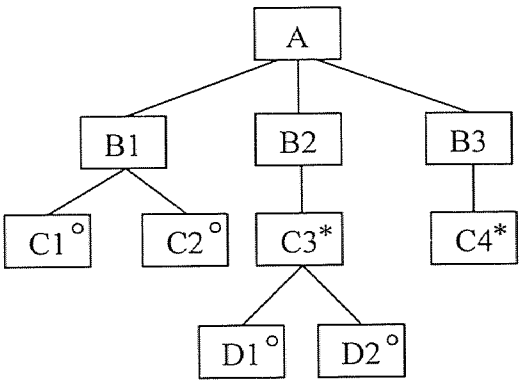

Figure 7.1 A Jackson Diagram

A Jackson Diagram traditionally consists of modules drawn as boxes [27]. For example, a Jackson Diagram is shown in Figure 7.1. The boxes are connected through lines to form a hierarchy. There are three kinds of boxes: plain boxes, circle boxes and asterisk boxes. The root box must be a plain box. A line from a parent to children means

that the parent contains the children while a plain box child means sequential, a circle box child means selective and an asterisk box child means repetitive. The three different kinds of boxes are actually the three program control structures.

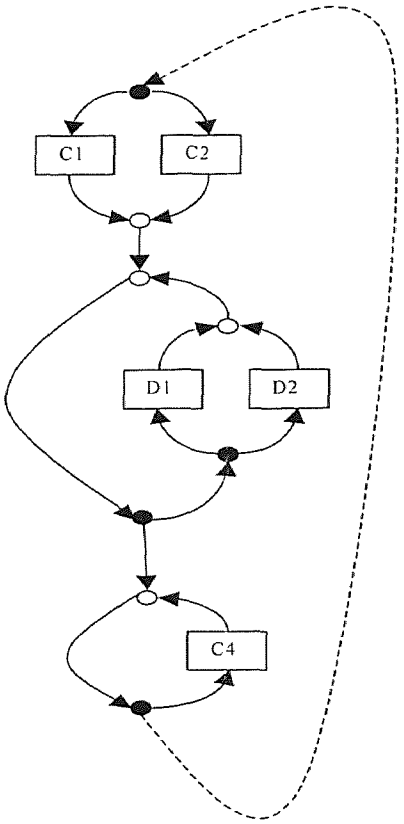

Figure 7.2 The Flowgraph of the Jackson Diagram in Figure 7.1

A common property of Jackson Diagrams and flowgraphs is that they use three program control structures. The only difference is that they use different notations in a diagram. Figure 7.1 is a Jackson Diagram while Figure 7.2 is the corresponding flowgraph. We suppose that the diagram in Figure 7.1 represents an aggregation. Module A consists of modules B1, B2, and B3 that are executed sequentially. Module B1

consists of components C1 and C2, one of them is executed based on a selection condition. Module B2 contains module C3 and uses it repeatedly if a condition is true. Module B3 contains module C4 and uses it repeatedly if a condition is true. Module C3 selectively executes one of components D1 and D2 in terms of a condition.

Thus module A with B1, B2, and B3 is a sequential structure. Module B1 with C1 and C2, and module C3 with D1 and D2 are selection structures. Module B2 with C3 and module B3 with C4 are repetitive structures. For a selection structure or a repetitive structure, the condition is implied in parent node of this structure. For a sequential structure in Jackson Diagrams it is drawn as a sequential prime flowgraph. For a selection structure in Jackson Diagram it is drawn as a selection prime flowgraph. For a repetitive structure in Jackson Diagram it is drawn as a repetitive prime flowgraph. Then we follow the order of the structures in the Jackson Diagram and compose the prime flowgraph to a general flowgraph according to the composition steps for flowgraphs introduced in Chapter 2. The Jackson Diagram example in Figure 7.1 has been converted to a structured flowgraph shown in Figure 7.2.

It is noteworthy that a non-leaves box in a Jackson Diagram implies a prime flowgraph. Furthermore, the condition for a selection or a repetition in a Jackson Diagram is implied in a parent node within a selective or repetitive structure. But the condition for a selection or a repetition is represented explicitly as a decision node in a flowgraph. The leaves in the Jackson Diagram are the explicit process nodes that are actually elementary functional units in an aggregation flowgraph.

According to the properties of the Jackson Diagram and the rules of creating a flowgraph, we got a flowgraph shown in Figure 7.2. It consists of decision nodes,

junction nodes and functional unit nodes. The nodes, A, B1, B2, B3 and C3 are implied in the flowgraph. The implicit correspondences between the Jackson Diagram and the flowgraph are shown in Figure 7.3.

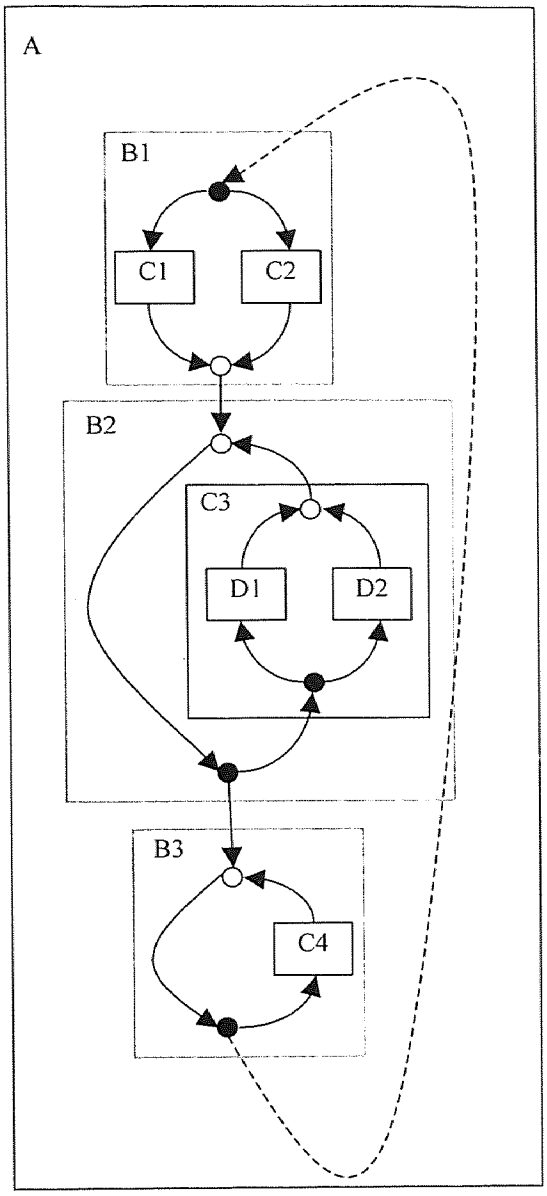

Figure 7.3 Implicit nodes in the Jackson Diagram

7.2 Aggregation Flowgraph Decomposition

As we have discussed in Section 7.1, an aggregation can be represented as an aggregation flowgraph. In order to generate executable code for an aggregation, it is very helpful for us to decompose the flowgraph into prime flowgraphs. Basically it is very useful if we know how many prime flowgraphs are included in a flowgraph before we are going to do the decomposition.

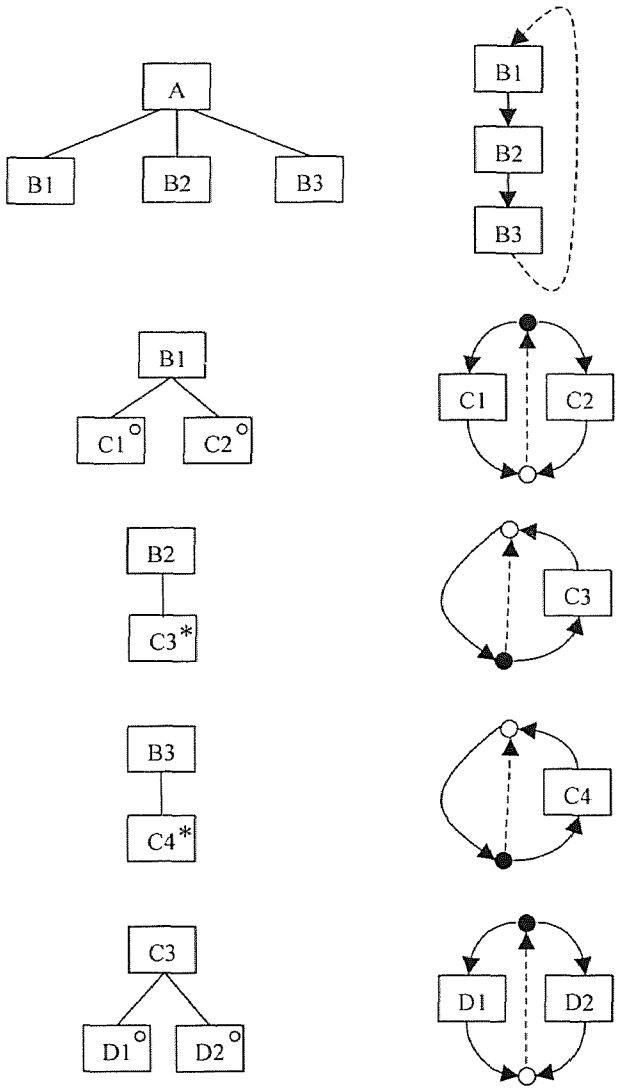

Figure 7.4 Jackson Diagrams and their prime flowgraphs

In Chapter 5, we conclude in Corollary 5 that the cyclomatic complexity of a standard structured flowgraph equals the number of prime flowgraphs in the flowgraph. Since each aggregation flowgraph is a standard structured flowgraph, we can compute its cyclomatic complexity. If we know the number of prime flowgraphs, it is easy for us to perform the decomposition. Now let us check the aggregation flowgraph in Figure 7.2. The cyclomatic complexity of the flowgraph is 5 because it has 4 decision nodes. Therefore, it has 5 prime flowgraphs. The 5 prime flowgraphs and their corresponding Jackson Diagrams are shown in Figure 7.4.

We decompose an aggregation flowgraph into some prime flowgraphs according to the decomposition steps introduced in Chapter 2. However, when we decompose an aggregation flowgraph into prime flowgraphs, some new rules must be followed. Generally a program module has its run-time contexts: some variables need to be initialized, some values need to be prepared, or a return value needs to be saved somewhere. Since each prime flowgraph contains a set of functional units, the units have their own run-time contexts that make the prime flowgraph have its own contexts. Therefore when an aggregation flowgraph is decomposed into prime flowgraphs, the prime flowgraphs and their contexts can not be separated.

This is the integrity of a prime flowgraph. Any decomposition must not violate the integrity rule. In other words, the decomposition must keep a prime flowgraph and its contexts together. In addition, we also need to name each prime flowgraph. For example, each prime flowgraph in Figure 7.2 has its name shown as in Figure 7.3.

7.3 Applications

We have introduced two problem examples, KWIC and ECC, in the previous chapter. The preliminary designs for the problems have been done there. This section focuses on the detailed designs. Through using the two examples, how to represent a system in Jackson Diagrams and flowgraphs, and how to decompose a system into prime flowgraphs are demonstrated. The studies of the problem examples enhance our methodology for component-based system integration.

7.3.1 The KWIC System

Shaw and Garlan provided four solutions for the KWIC problem [64]. They solved the problem based on four architectural styles. The styles are *main program/subroutine with shared data, abstract data types, implicit invocation,* and *pipes/filters.* Dellarocas gave nine solutions for the KWIC problem [15]. The nine solutions used two types of components, *filters* and *servers*. Moreover, three architectural styles were employed. The styles are *pipes/filters, main program/subroutine* and *implicit invocation.* Here two solutions are proposed based on pipes/filters and main program/subroutine styles respectively with filter and server component types.

7.3.1.1 Solution 1 - Pipes and Filters: This solution takes the pipes/filters architectural style and assumes that all the components are filters, typically the UNIX filters. Based on the analysis done in Chapter 6, the components, input, shift, sort and output, are filters. Generally a filter reads data from a pipe, processes them, and then writes them to a pipe. The whole process in a pipe/filter system works similarly to a pipeline. A pipe is usually implemented as sequential byte stream, such as a file, especially a sequential file.

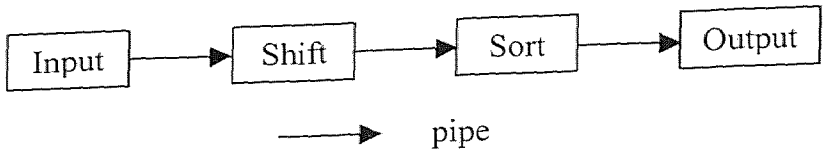

Figure 7.5 A solution with pipes/filters for the KWIC

Since the pipes/filters architectural style is used, we have to assume that all the lines in the KWIC system input are shifted and each line contains the number of words that need to be shifted. Based on this assumption, the input component reads a sequential file and writes it to a pipe. Then the shift component receives the file and reads each line and the number of shifted words, shifts the line and writes the line back to the file. After shifting all the lines, the shift component writes the file to the next pipe. Furthermore, the sort component gets the file, sorts it and writes it to the next pipe. Finally the output component reads the file and outputs it. The pipes/filters architecture for the KWIC system is shown in Figure 7.5. Here the preliminary design shown in Figure 6.4 is used.

Clearly the components in the architecture are executed sequentially. The architecture can be represented as a Jackson Diagram shown in Figure 7.6. It contains only one aggregation. The aggregation flowgraph is shown in Figure 7.7. The aggregation flowgraph is already a prime flowgraph.

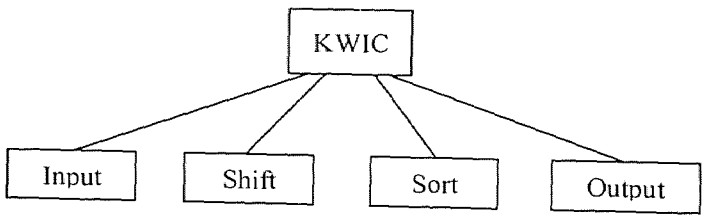

Figure 7.6 A Jackson Diagram for the KWIC system with pipes/filters

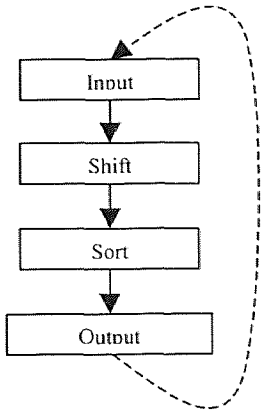

Figure 7.7 A flowgraph for the KWIC with pipes/filters

7.3.1.2 Solution 2 - Main Program/Subroutine: In this section we propose another solution for the KWIC system. The solution takes main program/subroutine as architectural style. In addition, we assume that the input, sort and output components are filters, and the shift component is a server. Considering the reusability of new composite components produced in the system, we use the preliminary design shown in Figure 6.5.

In order to increase the flexibility of the system, it is assumed that the shift component shifts one line of text each time it is called. This assumption makes it possible that selected lines are shifted. Moreover, the shift component is fed one line of text a time. It means that the shift component as a server communicates with others at the level of lines of text. All the components are subroutines and all the connectors serve as main programs because of the main program/subroutine style. Combining the discussions above, the whole system is a main program and has an architecture as shown in Figure 7.8.

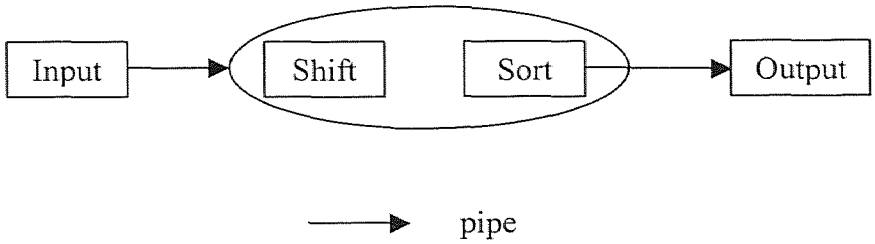

Figure 7.8 A solution for the KWIC with main program/subroutine

Since the input and output components are filters, the main program uses them as in a pipes/filters system. In other words, the main program needs to build such a pipes/filters environments for the filters. The oval shape in the architecture represents the connector in Figure 6.5. The connector contains two components, one is a server and another is a filter. The external circumstances of the connector require it to be a filter. But the internal structure requires it to be implemented slightly different from a real filter. First of all, the connector needs to read from a pipe. It also needs to separate the byte stream into lines of text that are fed to the shift component line by line. Then it has to organize the shifted lines as a byte stream so that the sort component, a filter, can read data from the stream. Finally the sort component writes the sorted result to a pipe. On the other hand, if more than one line needs to be shifted, the shift component would be called repetitively by the connector.

As we discussed before, the implementation details are represented as a Jackson Diagram shown in Figure 7.9. The Jackson Diagram can be converted to a flowgraph according to the conversion rules discussed in Section 7.1. Figure 7.10 shows the flowgraph.

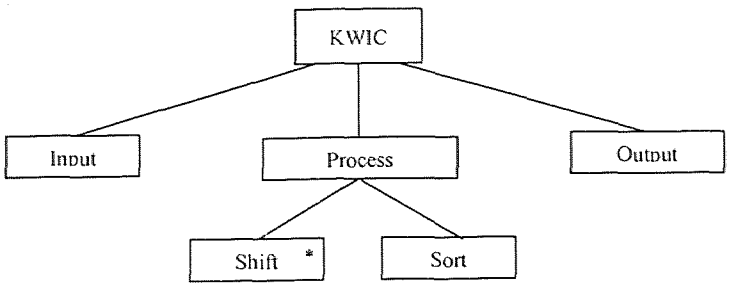

Figure 7.9 Jackson Diagram for the KWIC system with main program/subroutine

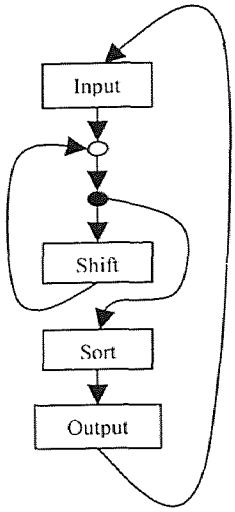

Figure 7.10 Flowgraph for the KWIC system with main program/subroutine

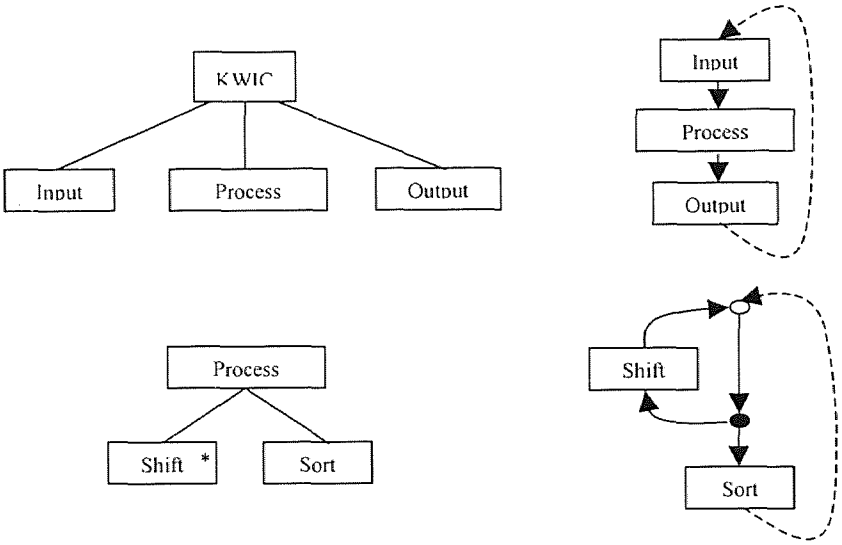

Figure 7.11 Aggregation Jackson Diagrams and flowgraphs

The Jackson Diagram in Figure 7.9 suggests that there be two aggregations. According to the architectural aggregation view, the Jackson Diagram can be

decomposed into two aggregations, a system aggregation and a process aggregation. The two aggregations have their own flowgraphs. One of the aggregation flowgraphs is a prime flowgraph. The other is a non-prime flowgraph. The aggregation Jackson Diagrams and flowgraphs are shown in Figure 7.11. Figure 7.12 shows the decomposition of the non-prime flowgraph.

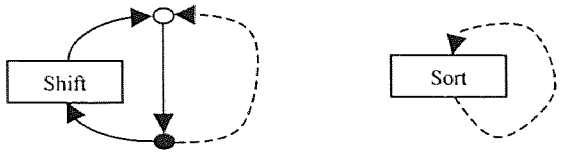

Figure 7.12 A decomposition of the non-prime flowgraph

7.3.2 The ECC System

In Chapter 6, the requirements analysis and a preliminary design are explored for the ECC system. The preliminary design is summarized in Figure 6.7. This section focuses on the detailed design. The details are represented as a Jackson Diagram (Figure 7.13) that will be converted to a flowgraph. The resulted flowgraph will be decomposed into aggregation flowgraphs that are furthermore decomposed into prime flowgraphs, if possible.

The preliminary design in Chapter 6 indicates that there are five components and a connector in the ECC system. A solution proposed here uses the main program/subroutine architectural style and assumes that the type of all the components is server. A component of the server type is fed with inputs and returns the results to requestors. As we discussed in Chapter 6, the ECC system works only if it is on. The Figure 7.13 shows the working details of the ECC system. First of all, the ECC system is set to be on by the vehicle driver. When it is on, the system starts to calculate the values

for the initial throttle setting. Then the system will maintain the on state. If one of the arguments changes, the ECC system adjusts the throttle setting. The ECC system releases the speed control only if the brake pedal is pressed or the brake is in on state.

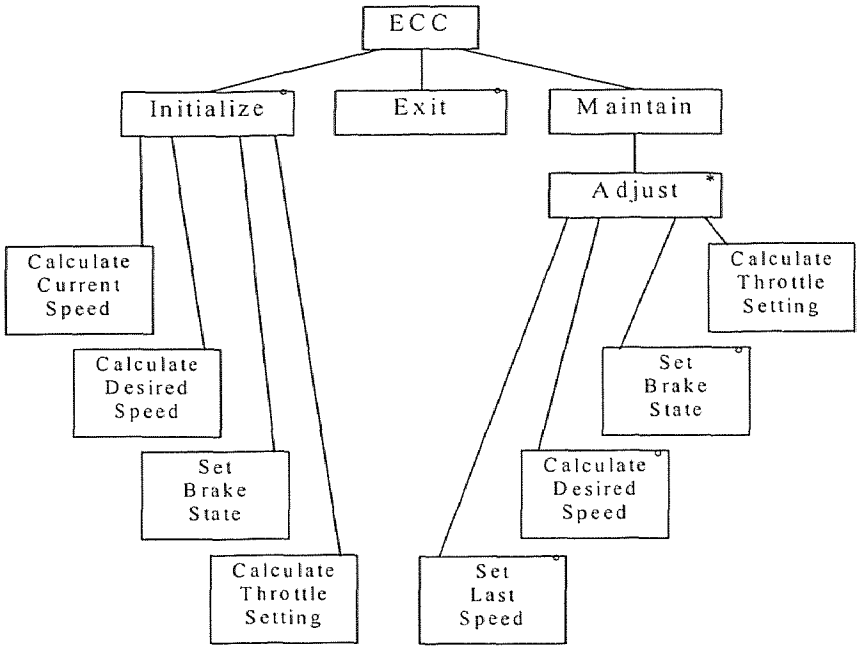

Figure 7.13 A Jackson Diagram for the ECC system

Our methodology requires that a Jackson Diagram be decomposed into aggregations. Each aggregation is represented as a flowgraph. For a small-scale system, it is useful to draw a flowgraph for a system Jackson Diagram because the flowgraph can indicate how many prime flowgraphs are included in the whole system. For a large-scale system, it usually contains lots of components. The flowgraph for such a system is too complex. Therefore, the system is represented as a Jackson Diagram that is furthermore decomposed into aggregations. After decomposition, each aggregation is still represented as a Jackson Diagram. Then the aggregation Jackson Diagrams are converted to flowgraphs. These aggregation flowgraphs are decomposed into prime flowgraphs. These ideas are demonstrated here through exploring the ECC system. The flowgraph of

the Jackson Diagram of the ECC system is shown in Figure 7.14. Both the Jackson Diagram in Figure 7.13 and the flowgraph in Figure 7.14 imply that there are four aggregations. Each of them is already a prime flowgraph or consists of a number of prime flowgraphs.

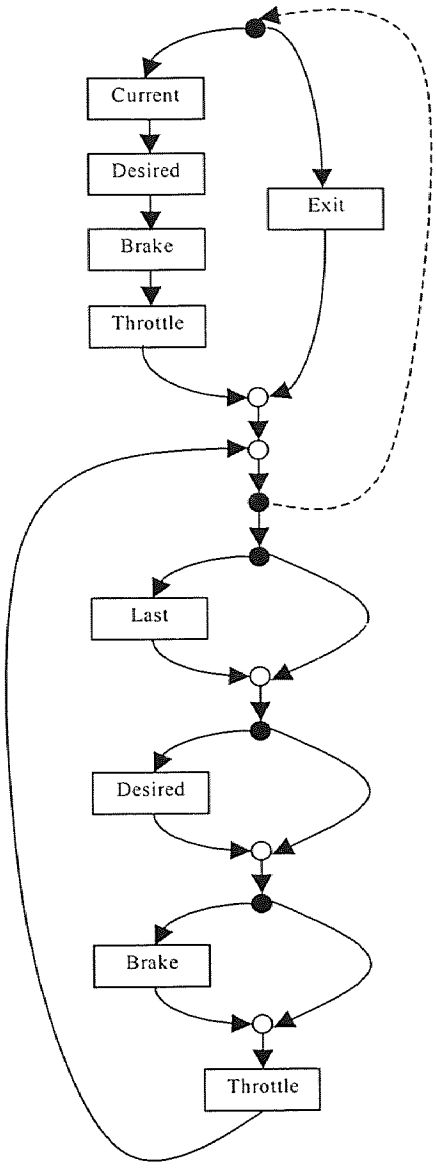

Figure 7.14 The flowgraph for the ECC system

The aggregations in the ECC system are the results of decomposition of the Jackson Diagram in Figure 7.13 or the flowgraph in Figure 7.14. These aggregations correspond to prime flowgraphs or can be decomposed into prime flowgraphs. The Jackson Diagram and flowgraph of each aggregation is shown in Figures 7.15, 7.16, 7.17, and 7.18. In Figure 7.14 the names of components are abbreviated to one word. For example, Current represents Calculate Current Speed; Desired represents Calculate Desired Speed; Last represents Set Last Speed; Brake represents Set Brake State; Throttle represents Calculate Throttle Setting. The short names are also used in Figures 7.17 and 7.18.

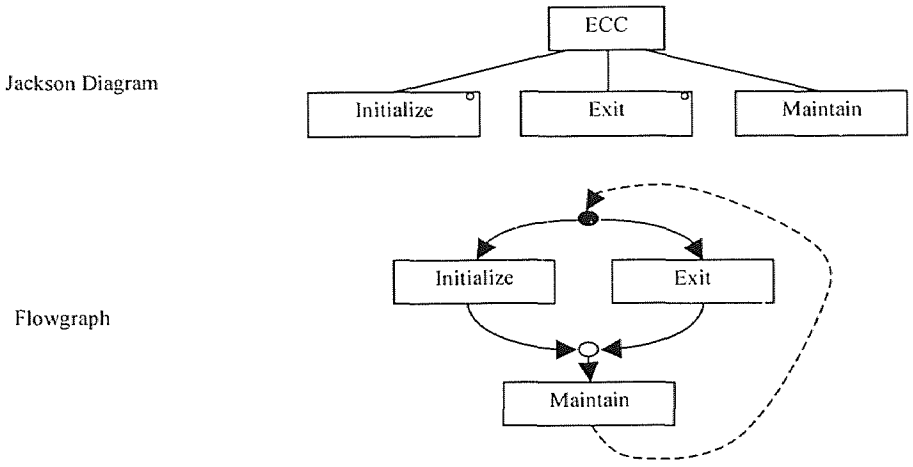

Figure 7.15 Aggregation ECC: Jackson Diagram and flowgraph

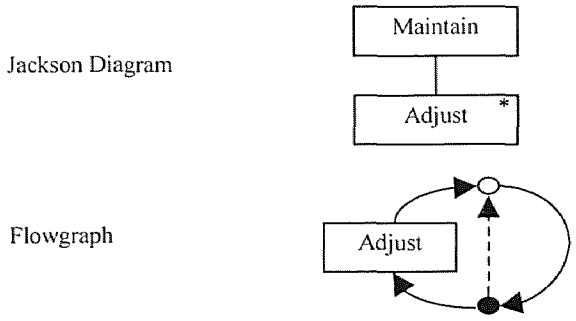

Figure 7.16 Aggregation Maintain: Jackson Diagram and Flowgraph

Jackson Diagram

Flowgraph

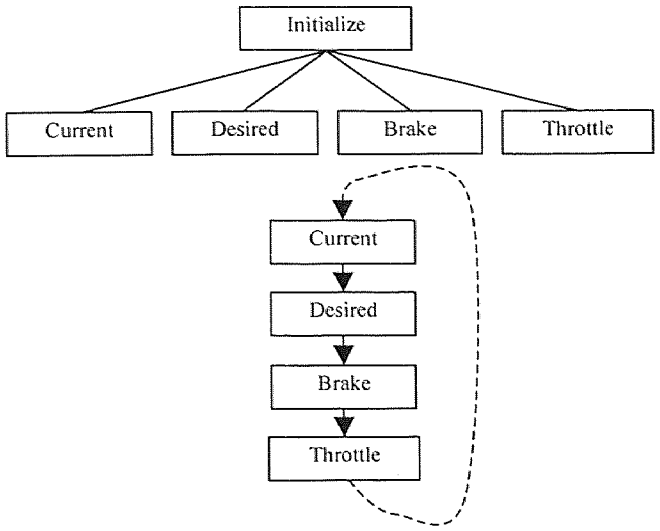

Figure 7.17 Aggregation Initialize: Jackson Diagram and flowgraph

Jackson Diagram

Flowgraph

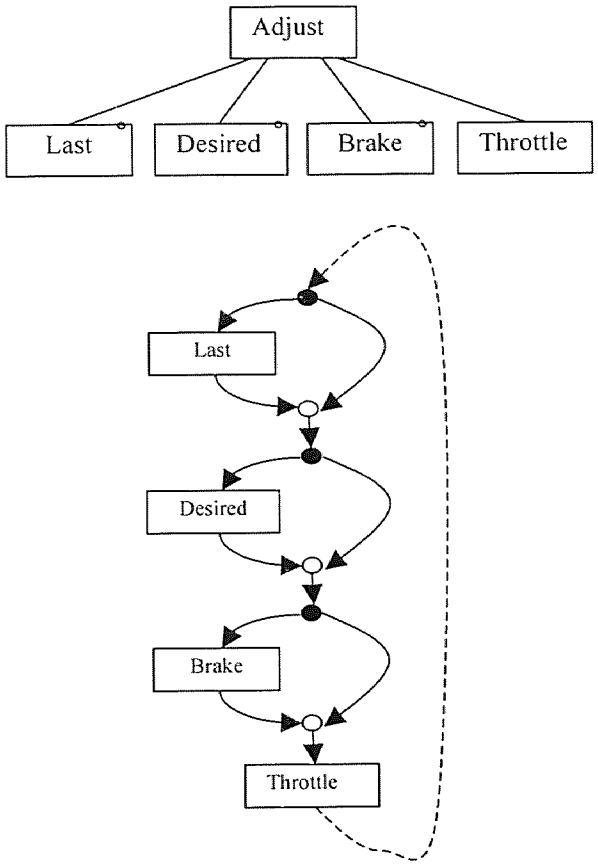

Figure 7.18 Aggregation Adjust: Jackson Diagram and flowgraph

The aggregation flowgraphs in Figures 7.16 and 7.17 are prime flowgraphs. However, the aggregation flowgraphs in Figures 7.15 and 7.18 are non-prime flowgraphs. They can be decomposed to a number of prime flowgraphs. It is really easy to do the decomposition as long as the decomposition rules introduced in Chapter 2 are followed. The decomposition results are shown respectively in Figure 7.19 and Figure 7.20. Figure 7.19 contains the prime flowgraphs from Figure 7.15. Figure 7.20 contains the prime flowgraphs from Figure 7.18.

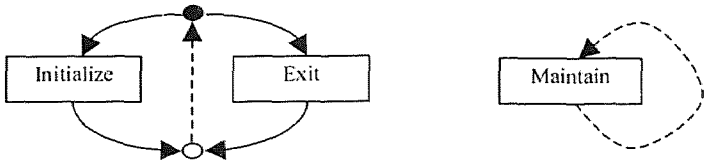

Figure 7.19 Prime flowgraphs contained in aggregation Initialize

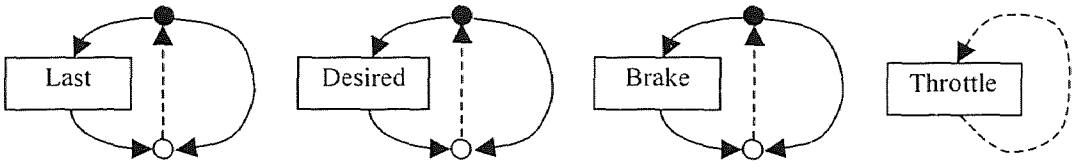

Figure 7.20 Prime flowgraphs contained in aggregation Adjust

7.4 Summary

The decomposition is an efficient way to control the complexity of a large-scale software system. In fact, the component-based software paradigm is proposed to attack the problems a large-scale software system faces. Based on the promise of component-based software system, we proposed an aggregation view on software architecture in Chapter 6. The examples we explored so far are the important proofs for the aggregation architectural view. Therefore, how to make the view feasible to a large-scale software

system is the next task for our proposed methodology. Fortunately we verified the usefulness of Jackson Diagrams and flowgraphs. In this chapter, it is proposed to represent a system architecture as a Jackson Diagram and convert the Jackson Diagram to a flowgraph. The activities involved in the graphical representation are summarized in this section.

As we mentioned before, our methodology starts with the detailed design of a system. Then the detailed design is represented as a Jackson Diagram. The Jackson Diagram explicitly expresses how many aggregations are included in the system. Then the Jackson Diagram is converted into a flowgraph that will be used for the final composition of a system from the aggregations. The Jackson Diagram is further decomposed into aggregations on different levels. The aggregations are represented as aggregation flowgraphs that are decomposed into prime flowgraphs if possible.

CHAPTER 8

SYSTEM IMPLEMENTATION DESCRIPTION

A component-based software system is a set of different-level aggregations and some independently deliverable components in terms of aggregation view. The only difference between different-level aggregations is the granularity of functional units. If the granularity of functional units is ignored, they all can be considered as aggregations of functional units. Since every aggregation has a flowgraph for its detailed design, the flowgraph definitely indicates how the functional units in it work together interactively. We call the interactive relationship of the functional units in an aggregation an interconnection of them. Since a flowgraph can be decomposed into some prime flowgraphs, it means that the interconnection consists of some prime flowgraphs. We call such a prime flowgraph a prime connection. We discuss the prime connection, and what should be included in a prime connection in Section 8.1. A system implementation description framework including a component description framework is proposed in Section 8.2. Section 8.3 introduces an Implementation Description Language (IDL). The work in this chapter is summarized in Section 8.4.

8.1 Prime Connections

An aggregation flowgraph can be decomposed into some prime flowgraphs as mentioned before. The prime flowgraph could be a sequential structure, selection structure or repetitive structure. Such a structured flowgraph can be represented as a regular expression to show the relationship among the involved functional units. Each prime flowgraph consists of a set of subsystems, connectors, components or other prime

flowgraphs that are connected through a sequential relationship, a selective relationship or a repetitive relationship. We define a connection within a prime flowgraph a *prime connection*. Simply speaking, a prime flowgraph is a prime connection. For instance, the prime flowgraphs in Figures 7.7, 7.11, 7.12, 7.15, 7.16, 7.17, 7.19 and 7.20 are prime connections because they are already one of the three program control structures.

In addition, we note that a functional unit in a prime connection could be a component, an aggregation, or other prime connections. We define a prime connection as a *pure prime connection* if it only includes components. Otherwise, it is a *virtual prime connection*. However, the prime connections and the aggregations in a prime connection play the same role as the components in it do. For example, the prime flowgraphs in Figures 7.7, 7.12, 7.16 and 7.20 are pure prime connections because they contain only components. Instead, the prime flowgraphs in Figures 7.15, 7.17, 7.19 and the first prime flowgraph in Figure 7.11 are virtual prime connections. In the rest of this section, we discuss what should be included in a prime connection.

In terms of the definition of a prime connection, a prime connection is a prime flowgraph. A flowgraph could consist of decision node, junction node and process nodes. The decision and junction nodes correspond to a logical condition and process nodes correspond to components, aggregations or other prime connections. Only the selective and repetitive structures have logical conditions with their necessary prerequisites. However, since a prime connection is just a syntactic unit and not a semantic unit, its run-time environment depends on the components and aggregations in it.

Now the internal environment of a prime connection is explored. There are process nodes in a prime connection. The process nodes may be components, aggregations or other prime connections. Together with the components and aggregations are their execution contexts. For example, if a component's method is called, what data need to be prepared, what values need to be passed to the method or what requirements need to be met before a component or a composite component is invoked. For a subsystem or a connector, in fact, they are aggregations. Hence, their execution contexts also mean what need to be prepared in advance before they are invoked. Furthermore, the context of a component or an aggregation can be divided into context before the execution and context after execution. They are called pre-context and post-context respectively.

However, if a prime connection is included in other prime connections, it is not necessary to define its contexts again in them because it is already done elsewhere. Therefore we conclude that a prime connection contains components or aggregations with their pre-contexts and post-contexts, a branching condition with its prerequisites, structural type, structural relationship expressed in a regular expression, and other prime connections. In addition, a prime connection could be pure or virtual. We abstract it as a type for a prime connection to indicate that it is a pure one or a virtual one.

Now we analyze an aggregation example in Figure 7.18. The aggregation flowgraph has been decomposed into four prime flowgraphs shown in Figure 7.20. Actually there are four prime connections. They are shown here in Figure 8.1 with names for each prime connection and their regular expressions. Clearly the prime connections, Adjust_1, Adjust_2, and Adjust_3, are pure prime connections, but the

prime connection Adjust is a virtual one because the first three contain only the components and the last one contains other prime connections. In addition, it is necessary to name each prime connection in an aggregation.

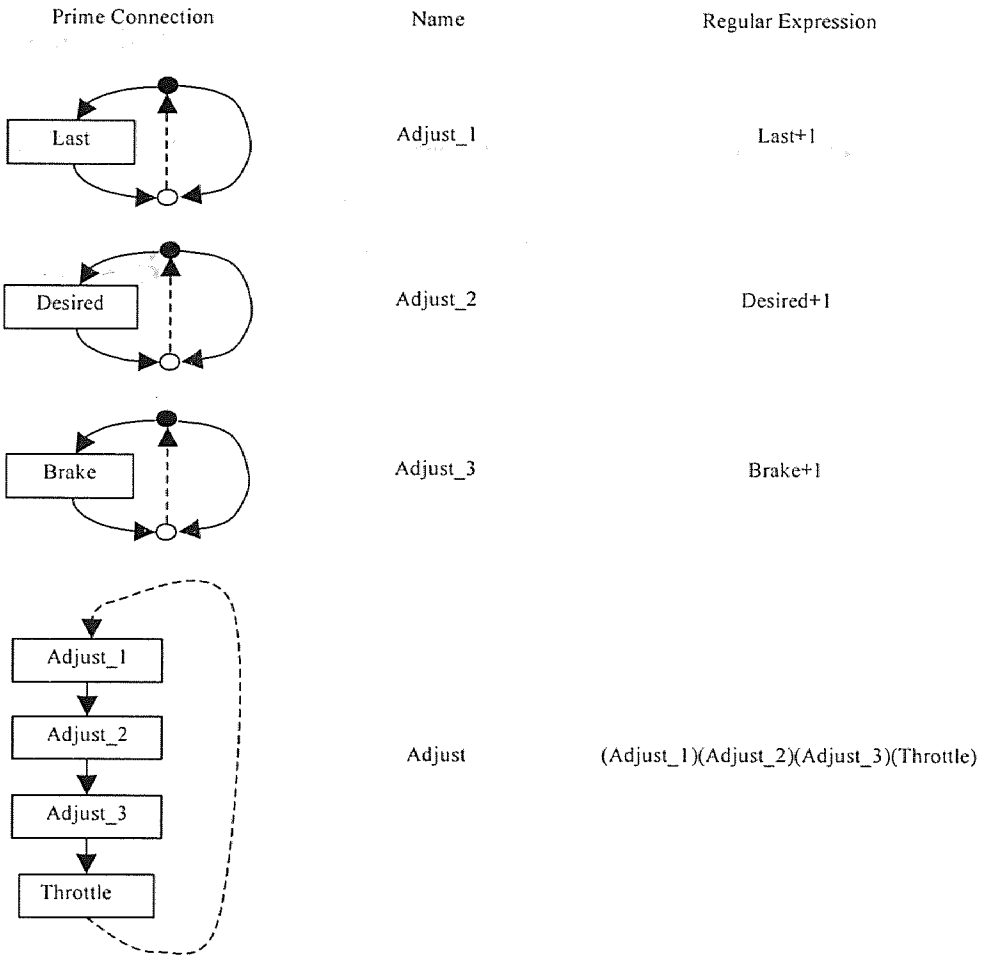

Figure 8.1 Prime connections and their regular expressions

8.2 Implementation Description Framework

The analysis of an aggregation and prime connections in previous section suggests that the aggregation description be the kernel of a component-based software system in terms of our view. If an aggregation can be described completely, there are no problems for us to describe the whole system. Basically an aggregation has a number of prime

connections which include type, structural type, structural relationship, logical condition with its prerequisites, components or aggregations with their pre-contexts and post-contexts, and other prime connections. Therefore, an implementation description framework for an aggregation enhances that an aggregation consists of a set of prime connections. The framework is shown in Table 8.1.

Table 8.1 An aggregation description framework

Aggregation
Prime connection
Type
Structural type
Structural relationship
Logical condition
Prerequisite
Component
Pre-context
Post-context
Aggregation
Pre-context
Post-context
Prime connection

A complete framework should not miss the description of components. Here we discuss what framework could be used for the description of components, especially the interfaces of the components. Although components are the smallest-grain functional units, when we design a component-based system, the required components need to be searched. Thus it is necessary to describe a component interface. We have proposed a component model in Chapter 6. That model is a comprehensive one. The complete information in the model is used for components in a software library. This model implies a component description framework shown in Table 8.2. It is just used for

component search. An IDL must also provide some mechanism to describe components for search. The IDL we propose in the next section covers the component description.

Table 8.2 A component description framework

Component
Type
Method
Type
Signature
Informal specification
Formal specification
Consumed event
Informal specification
Formal specification
Generated event

The interfaces of a component generally consist of type, methods, and events. The type information in the framework indicates what the type of a component is. For example, the component is a server or a filter. The methods and events reflect the functional services provided by a component. Furthermore, for a method, its type, signature, and specifications need to be described. The type of a method means the data type of its return value. The signature of a method includes the name of the method, and arguments together with their names and data types. Since a component may consume a number of events and also generates a number of events, in this framework, we classify the events to be consumed events and generated events. For a consumed event, generally it corresponds to a piece of program. The behavior of the program needs to be specified. The specifications embody the behaviors of the methods and events. We propose to describe specifications for methods and events in two ways, informal specification and

formal specification. The informal specification serves to system designers. The formal specification serves for component search.

8.3 Implementation Description Language

A component-based software system is generally considered as a four-level architecture, system level, subsystem level, connector level and component level. The four levels correspond to system aggregation, subsystem aggregations, connector aggregations and components in terms of our aggregation view. If a system is integrated from components automatically or semi-automatically, it is necessary to describe the detailed design for connectors, subsystems, and system and to specify the details of components. The specification of components is used for searching the required components.

This section introduces an Implementation Description Language that covers the specifications of components and aggregations that could be connector aggregations, subsystem aggregations or system aggregation. In the language, the implementation description starts from the top level, system aggregation, down to subsystem aggregations, then connector aggregation, and finally components. In this way the description covers the whole entities from root level to leaves in a software system architecture based on our aggregation view. But these different-level aggregations are not explicitly called system aggregation, subsystem aggregations or connector aggregations in the language; instead they are simply called aggregations. Since each aggregation has a unique name, we use their names to distinguish them. Therefore, a system consists of components and aggregations, and an aggregation consists of prime connections. The implementation description templates of a system, an aggregation and a prime connection are given in Table 8.3, Table 8.4 and Table 8.5 respectively. The syntax

of the IDL is specified using the Extended Backus-Naur Form (EBNF). The details of the IDL can be seen in Appendix A.

Table 8.3 A system description template

```
System <name> '{'
        Aggregation <aggregation name>{',' < aggregation name>}';'
        Component <component name>{','< component name>}';'
'}'
```

Table 8.4 An aggregation description template

```
Aggregation <aggregation name> '{'
        {<Prime connection>';'}
'}'
```

Table 8.5 A prime connection description template

```
Prime connection <prime connection name> '{'
        Type ('pure' | 'virtual')';'
        Structural type ('sequential'| 'selective' | 'repetitive')';'
        Structural relationship <regular expression>';'
        Logical condition <boolean expression> [with
            Prerequisite <prerequisite specification> ] ';'
        {Component <component name> '{'
            [Pre-context <specification for pre-context>';']
            [Post-context < specification for post-context>';']
        '}'';'}
        {Aggregation <aggregation name>'{'
            [Pre-context <specification for pre-context>';']
            [Post-context < specification for post-context>';']
        '}'';'}
        [Prime connection <prime connection name list>';']
'}'
```

Table 8.3 indicates that a system consists of a number of aggregations and components. Table 8.4 shows that an aggregation contains a number of prime connections. The contents of a prime connection are described in Table 8.5. It illustrates

that a prime connection includes the type of the prime connection, the structural organization of the prime connection, the structural relationship among the functional units and other prime connections, logical condition with its prerequisite for a selective relationship or a repetitive relationship, components, aggregations and other prime connections.

Table 8.6 A component description template

```
Component <component name> '{'
        Type <component type> ';'
        {Method '{'
                Type <data type>';'
                Signature <method name>'('<argument list>')' ';'
                Informal specification <text>';'
                Formal specification <behavior specification>';'
        '}'}
        {Consumed event <name> '{'
                Informal specification <text>';'
                Formal specification <behavior specification>';'
        '}'}
        [Generated event <event name list>';']
'}'
```

When a component-based software system is to be built, the required components need to be searched. Hence, in a comprehensive IDL, a component must be described. A description framework for a component is proposed in Section 8.2. In terms of the framework, an IDL description template for a component is shown in Table 8.6.

The formal specification is used in the IDL to describe the prerequisite of a logical condition, the pre-contexts and the post-contexts of a component and an aggregation in a prime connection, and the behaviors of a method or a consumed event. In this IDL, a formal specification language is not proposed. We leave it open and let a system

architect to choose one that is the best to fit a particular system. A comprehensive survey on formal specification languages was done by Cooke, et. al. [13]. We are not going to analyze and evaluate the formal specification languages again in this dissertation.

```
System KWIC {
        Aggregation KWIC;
        Component Input, Shift, Sort, Output;
}
```

(a)

```
Aggregation KWIC {
        Prime connection KWIC {
                Type pure;
                Structural type sequential;
                Structural relationship (Input)(Shift)(Sort)(Output);
                Logical condition;
                Component Input;
                Component Shift;
                Component Sort;
                Component Output;
        }
}
```

(b)

Figure 8.2 An IDL description for the KWIC with pipes/filters style

The examples we discussed before can be described in the IDL. In Chapter 6, two problem examples are analyzed according to the aggregation view. The examples are the KWIC and ECC systems. Two solutions are provided for KWIC. One solution is provided for ECC. For the solution 1 of KWIC, its IDL descriptions for the system is shown in Figure 8.2(a) and the IDL descriptions for aggregation and prime connection are shown in Figure 8.2(b) in which each of the components has no pre-contexts and

post-contexts. It uses the pipes/filters architectural style and its components are filters. The whole system includes one aggregation.

All the components in this solution can also be described in the IDL. Figure 8.3 shows the description for component Input. Figure 8.4 shows the description for component Shift. Figure 8.5 shows the description for component Sort. Figure 8.6 shows the description for component Output. For those components, they are filters and they do not have any consumed events and generated events.

```
Component Input {
    Type filter;
    Method {
        Type File;
        Signature Input(File Kword);
        Informal specification "Read a file from an input device
                and write it to a pipe.";
        Formal specification;
    }
}
```

Figure 8.3 IDL description for component Input in the KWIC with pipes/filters

```
Component Shift {
    Type filter;
    Method {
        Type File;
        Signature Shift(File Kword);
        Informal specification "Read a file from a pipe, shift each text
                line and write it to a pipe.";
        Formal specification;
    }
}
```

Figure 8.4 IDL description for component Shift in the KWIC with pipes/filters

```
Component Sort {
    Type filter;
    Method {
        Type File;
        Signature Sort(File Kword);
        Informal specification "Read a file from a pipe, sort text lines
                into alphabetic order and write it to a pipe.";
        Formal specification;
    }
}
```

Figure 8.5 IDL description for component Sort in the KWIC with pipes/filters

```
Component Output {
    Type filter;
    Method {
        Type File;
        Signature Output(File Kword);
        Informal specification "Read a file from a pipe
                and write it to another pipe.";
        Formal specification;
    }
}
```

Figure 8.6 IDL description for component Output in the KWIC with pipes/filters

Moreover, solution 2 is provided based on the main program/subroutine style. In this solution, the component shift is a server. The Jackson Diagram for the solution is shown in Figure 7.9. Here we use the IDL to describe the KWIC system, aggregations and prime connections. The system description is shown in Figure 8.7. Figures 8.8 and 8.9 are the IDL descriptions for aggregations, KWIC and Process.

```
System KWIC {
        Aggregation KWIC, Process;
        Component Input, Shift, Sort, Output;
}
```

Figure 8.7 IDL system description for the KWIC with main program/subroutine

```
Aggregation KWIC {
    Prime connection KWIC {
        Type virtual;
        Structural type sequential;
        Structural relationship (Input)(Process)(Output);
        Component Input {
            Pre-context;
            Post-context;
        }
        Component Output {
            Pre-context;
            Post-context;
        }
        Aggregation Process {
            Pre-context;
            Post-context;
        };
    }
}
```

Figure 8.8 IDL description for aggregation KWIC with main program/subroutine

```
Aggregation Process {
        Prime connection Process_1 {
                Type pure;
                Structural type repetitive;
                Structural relationship 1*Shift;
                Logical condition current_shifted_line < expected_shifted_line;
                Component Shift {
                        Pre-context;
                        Post-context;
                };
        };
        Prime connection Process_2 {
                Type virtual;
                Structural type sequential;
                Structural relationship (Process_1)(sort);
                Component Sort {
                        Pre-context;
                        Post-context;
                };
                Prime connection Process_1;
        };
}
```

Figure 8.9 IDL description for aggregation Process with main program/subroutine

We have proposed a solution for the ECC system in Chapter 7. Here the IDL is used to describe the ECC system, the contained aggregations, prime connections and the components. There are four aggregations in the ECC system. Since the ECC system is a relatively large system, the system description and aggregation descriptions are separately written down. Each aggregation description is also written separately.

```
System ECC {
        Aggregation ECC, Initialize, Maintain, Adjust;
        Component Current, Desired, Last, Brake, Throttle;
}
```

Figure 8.10 An IDL system description for the ECC system

```
Aggregation ECC {
        Prime connection ECC_1 {
                Type virtual;
                Structural type selective;
                Structural relationship Initialize+Exit;
                Logical condition system_on;
                Component Exit {
                        Pre-context;
                        Post-context;
                };
                Aggregation Initialize {
                        Pre-context;
                        Post-context;
                };
        };
        Prime connection ECC_2 {
                Type virtual;
                Structural type sequential;
                Structural relationship (ECC_1)(Maintain);
                Aggregation Maintain {
                        Pre-context;
                        Post-context;
                };
                Prime connection ECC_1;
        };
}
```

Figure 8.11 The IDL description for aggregation ECC

The aggregations in the ECC system are ECC, Initialize, Adjust and Maintain. The system description in the IDL is shown in Figure 8.10. Then the aggregations with their prime connections are described. The IDL descriptions for the four aggregations are shown in Figures 8.11, 8.12, 8.13, and 8.14 respectively.

```
Aggregation Initialize {
        Prime connection Initialize {
                Type pure;
                Structural type sequential;
                Structural relationship (Current)(Desired)(Brake)(Throttle);
                Component Current {
                        Pre-context;
                        Post-context;
                };
                Component Desired {
                        Pre-context;
                        Post-context;
                };
                Component Brake {
                        Pre-context;
                        Post-context;
                };
                Component Throttle {
                        Pre-context;
                        Post-context;
                };
        }
}
```

Figure 8.12 The IDL description for aggregation Initialize

```
Aggregation Adjust {
        Prime connection Adjust {
                Type virtual;
                Structural type sequential;
                Structural relationship (Adjust_1)(Adjust_2)(Adjust_3)(Throttle);
                Component Throttle {
                        Pre-context;
                        Post-context;
                };
                Prime connection Adjust_1, Adjust_2, Adjust_3;
        }
        Prime connection Adjust_1 {
                Type pure;
                Structural type selective;
                Structural relationship Last+1;
                Logical condition resume_on;
                Component Last {
                        Pre-context;
                        Post-context;
                };
        }
        Prime connection Adjust_2 {
                Type pure;
                Structural type selective;
                Structural relationship Desired+1;
                Logical condition change_on;
                Component Desired {
                        Pre-context;
                        Post-context;
                };
        }
        Prime connection Adjust_3 {
                Type pure;
                Structural type selective;
                Structural relationship Brake+1;
                Logical condition Brake_on;
                Component Brake {
                        Pre-context;
                        Post-context;
                };
        }
}
```

Figure 8.13 The IDL description for aggregation Adjust

```
Aggregation Maintain {
        Prime connection Maintain {
                Type virtual;
                Structural type repetitive;
                Structural relationship Adjust*1;
                Logical condition system_on;
                Aggregation Adjust {
                        Pre-context;
                        Post-context;
                };
        }
}
```

Figure 8.14 The IDL description for aggregation Maintain

```
Component Current {
        Type server;
        Method {
                Type int;
                Signature calculate_current (int pulse, int tick);
                Informal specification "Calculate current speed of a vehicle.";
                Formal specification;
        }
}
```

Figure 8.15 The IDL description for component Current

```
Component Desired {
        Type server;
        Method {
                Type int;
                Signature Desired_current (int system_on, int change);
                Informal specification "Calculate desired speed of a vehicle.";
                Formal specification;
        }
}
```

Figure 8.16 The IDL description for component Desired

```
Component Last {
        Type server;
        Method {
                Type;
                Signature set_last_speed (int resume);
                Informal specification "Set the previous speed of a vehicle.";
                Formal specification;
        }
}
```

Figure 8.17 The IDL description for component Last

The IDL component descriptions are shown from Figure 8.15 to Figure 8.19. Here we assume that each component has just one method and the assigned arguments and their date types. These assumptions are the intuitive results, instead of designed results. They are used here to serve as a demonstration of the IDL descriptions. Several Boolean

variables, system_on, resume_on, brake_off, and change_on, are introduced in these aggregation descriptions.

```
Component Brake {
        Type server;
        Method {
                Type;
                Signature set_brake_state (int brake_on);
                Informal specification "Set brake state.";
                Formal specification;
        }
}
```

Figure 8.18 The IDL description for component Brake

```
Component Throttle {
        Type server;
        Method {
                Type int;
                Signature calculate_throttle ();
                Informal specification "Calculate throttle setting.";
                Formal specification;
        }
}
```

Figure 8.19 The IDL description for component Throttle

8.4 Summary

In this chapter, firstly we introduce a new concept, prime connections, based on prime flowgraphs. Since our methodology relies on the aggregation architectural view and an aggregation can be represented as a flowgraph, it is reasonable to decompose a flowgraph into prime flowgraphs. The composition theory for structured programs based on cubic graphs suggests that it be possible to compose a software system from components. This is why we use the concept of a prime connection. Then we discuss the description framework for systems, aggregations, prime connections and components. These frameworks can be considered as models for systems, aggregations, prime connections and components. An Implementation Description Language (IDL) is proposed to integrate a software system from components. The IDL is also applied to the KWIC and ECC systems. The systems, aggregations, prime connections and components are specified in the IDL. The results in this chapter are the further improvements on our methodology for component-based system integration. In the next chapter, the final system integration will be discussed. In fact, the system integration is tightly based on the work in this chapter.

The architectural description for a software system is an important research aspect in the software architecture community. In this chapter, we propose an implementation description language based on the concept of a prime connection. If a general architecture description language is used to describe the issues on the system design level, the IDL we proposed is used to describe the system issues on the system implementation level. The IDL is a significant part of the integration methodology because it is a base for automatic or semi-automatic system integration from components.

The purpose of introducing prime connection and the description framework for systems, aggregations, prime connections and components is to propose the language. Based on this point of view, our work in this chapter focuses on the language.

CHAPTER 9

SYSTEM INTEGRATION

System integration is the last step of our methodology. Generally system integration means to integrate a software system from components in terms of a component-based software style. In our methodology, system integration does not only mean that, but also means the translation of the IDL descriptions into source code written in a high level programming language, such as C, C++, or JAVA, and so on. In this chapter, the activities for system integration in our methodology are provided.

These activities may be component search, prime connection translation and composition. This integration step accepts the system implementation description in IDL as input. Then it separates component descriptions from the aggregation descriptions and search for them. After getting all the components, the methodology embeds the components into proper prime connections and translates the prime connections into a program segment coded in a high level programming language. As soon as the pure prime connections are translated, the virtual prime connections are translated. When the translations of prime connections in an aggregation are done, the aggregation is built. In Section 9.1 we introduce an outline of our approach to system integration. Section 9.2 discusses a number of preliminary issues for system synthesis. A synthesis mechanism including translation and composition is introduced in Section 9.3. The work in this chapter is summarized in Section 9.4.

9.1 An Outline of System Integration

The general ideas for system integration in our methodology are explored in this section. Simply speaking, the system integration in the methodology is to produce a software system represented in a high level programming language. In order to achieve this goal, we have proposed the aggregation view, component model, system representation approach, system decomposition approach and an IDL. Now the starting point for system integration are the IDL descriptions of a system. For example, the IDL descriptions from Figures 8.2 to 8.6, the IDL descriptions from Figures 8.7 to 8.9, and the IDL descriptions from Figures 8.10 to 8.19.

As we discussed in Chapter 8, the system descriptions consist of system description, aggregation descriptions and component descriptions. Such a description organization implies the primary activities involved in the system integration. Generally the integration process goes through components, pure prime connections, virtual prime connections, aggregations, and finally system. In order to make the activities clear, it is necessary to analyze description examples, such as the descriptions for KWIC shown in Figures 8.7, 8.8 and 8.9 in which a solution for KWIC is proposed. There are two aggregations, three prime connections, and four components in the solution. The dependencies among these functional units are shown in Figure 9.1.

The dependency relationship diagram in Figure 9.1 clearly indicates that the steps for system integration are to obtain the functional units in the following order: components, pure prime connections, virtual prime connections and aggregations. Thus the activities involved in system integration are to obtain components, to obtain pure

prime connections, to obtain virtual prime connections, and to obtain aggregations. How to perform these activities is the major topic in this chapter.

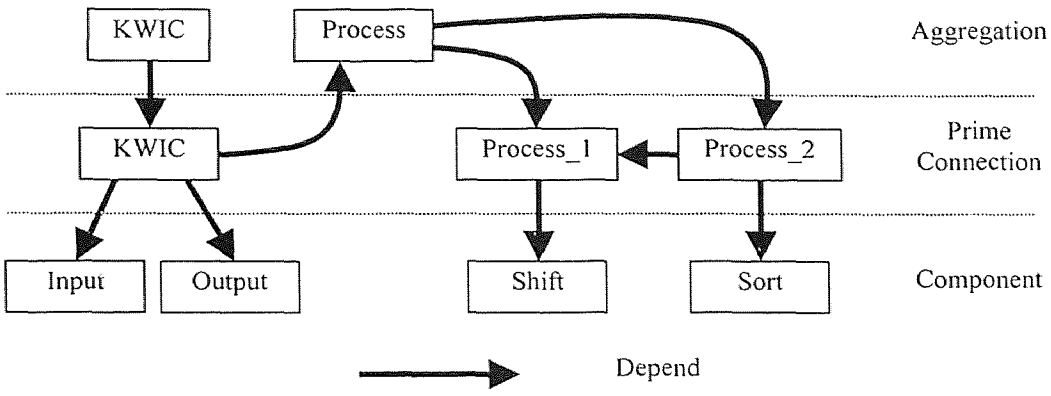

Figure 9.1 The functional units and their dependency relationships

The activity of obtaining components is usually called component search. The proposed component model in Chapter 6 and the component description part of the IDL in Chapter 8 serve the component search. In our methodology, the component search approaches are not proposed as we claimed in Chapter 1. Two approaches are surveyed in Chapter 4. The interested readers may reference the work in [51] and [81]. Actually the component search from a software library is still a critical research topic in the fields of component-based software. Our methodology organizes the component descriptions and sends them to a search engine that is a part of a software library.

The activities for obtaining pure prime connections include embedding the required components into the pure prime connections, and translating the IDL descriptions of the pure prime connections into source code written in a high level programming language. In fact, the embedding is done during the translation in our methodology. The translation will be based on the information provided in the IDL

descriptions. The information contains structural type and relationship and the names and contexts of components. The activities for obtaining virtual prime connections are similar to the activities for obtaining pure prime connections except for the embedding of aggregations and other prime connections.

It is noteworthy that for a system, there is always a corresponding aggregation that has the same name as the system. We used to call the aggregation a system aggregation. Actually when the aggregation is integrated from the functional units contained in it, the whole system is done. Furthermore, the system aggregation always contains a prime connection. This prime connection is always the last prime connection to be built, for example, the prime connection KWIC in Figure 8.8, the prime connection ECC_2 in Figure 8.11. Why does there exist a phenomenon like this? This is de facto the result of our system decomposition approach. Hence, in order to obtain an aggregation, we just need to create all the prime connections included in the aggregation and to know which one is the last prime connection.

Clearly the activities for system integration are to accept the IDL descriptions, to separate the component descriptions for search, to schedule the translation of prime connections, and to compose aggregations from their prime connections. In order to perform the activities, first of all, the descriptions need to be parsed to make these descriptions syntactically correct. Secondly, the component descriptions are separated and sent to search the corresponding components. After all the components are obtained, the pure prime connections are translated into program segments coded in a high level programming language. The components required by a pure prime connection are embedded into the program segment during the translation. Then virtual prime

connections are translated into program segments. At the same time, the required components, prime connections or aggregations are embedded into the program segments.

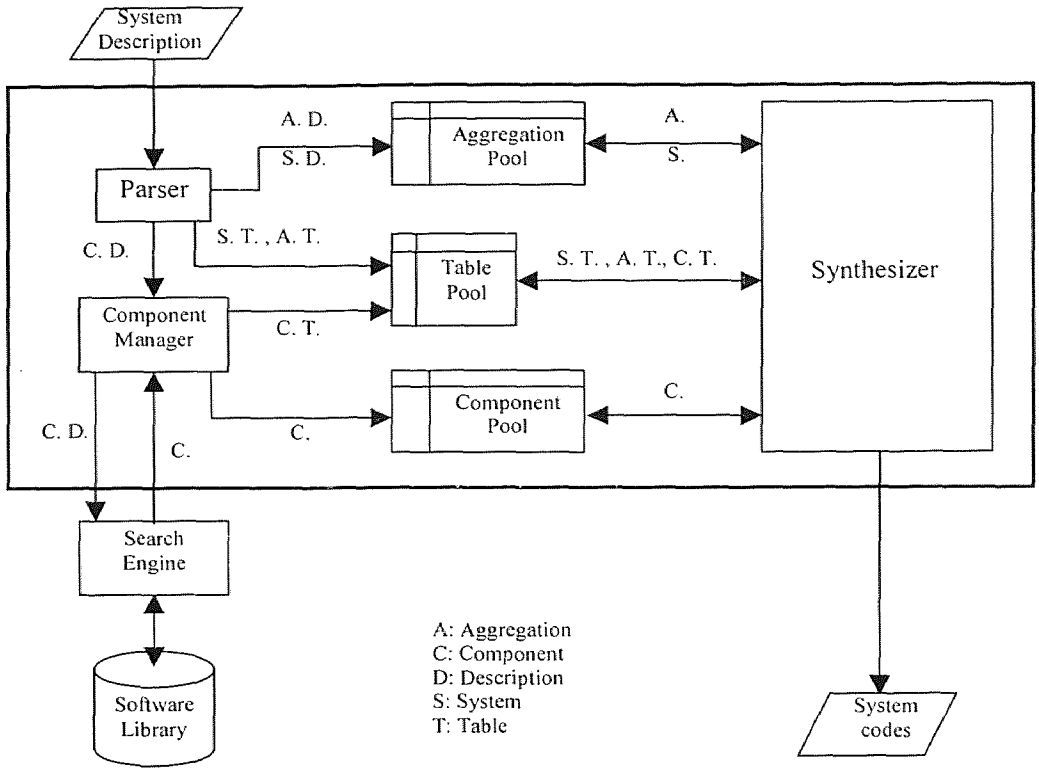

Figure 9.2 System integration

In reality, the embedding of prime connections or aggregations into a prime connection is the composition of flowgraphs. When the last prime connection has been translated, the whole aggregation is ready for being embedded into a subsystem or system. If the aggregation is the system aggregation, then the whole system is done. Therefore, those activities can be classified to be preliminary issues, translation and composition summarized in Figure 9.2. Their details are discussed in the rest of this chapter.

9.2 Preliminary Issues

The preliminary issues included in system integration are the parsing of IDL system descriptions, the separation of the component descriptions, the creation of component tables, and the creation of schedule tables for each aggregation. The tables are stored in a table pool. The translations and compositions of aggregations depend on these issues. We discuss the issues here to indicate how the issues assist the translations and compositions.

9.2.1 Parsing IDL Descriptions

The IDL descriptions are stored as text files. All the descriptions can be saved either in one file or in a number of files. For instance, an aggregation description can be organized in a file; the description of all components can be organized in a file; or the whole descriptions are organized in one file. All these situations are allowed in the methodology. When system integration starts, the IDL descriptions for system, aggregations and components are input. Then the component descriptions are separated from the other descriptions if the descriptions of whole system are organized in one file. Moreover, the descriptions for system and aggregations and the descriptions for components are respectively parsed according to the syntax of the IDL. It suggests that a parser for the IDL be built.

We suppose that we have the parser already. The parser accepts the description files as inputs. If the descriptions of a system are stored in a number of files, the file containing the system description should be parsed first. After the parser is invoked with a description file, it makes the first scan on the file to find any syntax errors. If the parser

finds any syntax errors in a description, it reports them, stops the scanning and exits. Then the syntax errors need to be fixed. When the errors are fixed, the parser is invoked again to repeat the first scan. Such steps may need to be repeated until no more syntax errors are found. Once there are no more syntax errors in the descriptions, the parser makes the second scan to create the system and aggregation tables, and sends the component descriptions for component search and the creation of the component table that is discussed in Section 9.2.4.

9.2.2 Creating the System Table

The information related to translation and composition is saved in system, aggregation and component tables. For a system table, the table has the same name as the system. It records the names and statuses of the aggregations included in the system, and the source code for each aggregation. The names of the aggregations are taken from the IDL description of the system. The status of each aggregation is used to indicate whether an aggregation is already translated or not. This table entry will be used for scheduling of translations of prime connections and other aggregations. After an aggregation has been translated, its source code is written into a file and the file name is entered into source field. A system table template is shown in Table 9.1 in which it is assumed that there is one aggregation. The table entry for the name field is the name of an aggregation. The table entry for the status field is 'yes' or 'no'. The 'yes' means that the aggregation is ready for embedding or composition. The 'no', instead, means that the aggregation is not ready. The table entry for the source field is the file name of source code for the aggregation.

Table 9.1 A system table template

Name	Status	Source

Some system table examples are introduced here. We have described the solutions for the KWIC and ECC systems in IDL in Chapter 8. We know that two solutions are provided for the KWIC system and one solution is provided for the ECC system. In solution 1 for the KWIC system, there is one aggregation. Its system table is shown in Table 9.2. In solution 2 for the KWIC system, there are two aggregations. Its system table is shown in Table 9.3. In the solution for the ECC system, there are four aggregations. Its system table is shown in Table 9.4. All the example tables are initial ones. It means that they are just created by the parser. The value, N/A, means that the source code for an aggregation is not available.

Table 9.2 A system table of solution 1 for the KWIC

Name	Status	Source
KWIC	No	N/A

Table 9.3 A system table of solution 2 for the KWIC

Name	Status	Source
KWIC	No	N/A
Process	No	N/A

Table 9.4 A system table for the ECC

Name	Status	Source
ECC	No	N/A
Initialize	No	N/A
Maintain	No	N/A
Adjust	No	N/A

9.2.3 Creating the Aggregation Table

After the parser creates the system table for a system, it will create the aggregation table as we mentioned before. We introduce the aggregation table now. An aggregation generally consists of prime connections. Thus the table entries in this table are related to prime connections. The information could be the name of a prime connection, its type, included components, aggregations, prime connections and source code for the prime connection. A table template for an aggregation is shown in Table 9.5. We assume that a typical prime connection contains three components, two aggregations and four other prime connections in this template.

Table 9.5 An aggregation table template

Name	Type	Status	Component			Aggregation		Prime Connection				Source

The name field records the name of a prime connection. The type field indicates that the prime connection is a pure or virtual one. The status field is used to show whether the prime connection is already translated or not. The component field records the names of components included in the prime connection. Similarly, the aggregation and prime connection fields are used to show how many aggregations and other prime connections included in the prime connections and the table entries are their names. The table entries for component field, aggregation field and prime connection field could be empty. The source field records the source code for a prime connection. The number of rows in an aggregation table depends on how many prime connections are contained in the aggregation.

Table 9.6 Aggregation table for Aggregation Process in Figure 8.9

Name	Type	Status	Component	Aggregation	Prime Connection	Source
Process_1	pure	No	Shift			N/A
Process_2	virtual	No	Sort		Process_1	N/A

Table 9.7 Aggregation table for Aggregation ECC in Figure 8.11

Name	Type	Status	Component	Aggregation	Prime Connection	Source
ECC_1	virtual	No	Exit	Initialize		N/A
ECC_2	virtual	No		Maintain	ECC_1	N/A

For example, the aggregation in Figure 8.9 has a table shown in Table 9.6 and the aggregation in Figure 8.11 has a table shown in Table 9.7. A typical aggregation table is shown in Table 9.8 in which there are no aggregations and other prime connections, instead of one prime connection. This original aggregation is shown in Figure 8.12. In Table 9.8, we use Agg. to represent Aggregation, Prime to represent Prime connection and Src to represent Source because of space limitations.

Table 9.8 Aggregation table for Aggregation Initialize in Figure 8.12

Name	Typ	Status	Component				Agg.	Prime	Src
Initialize	Pure	No	Current	Desired	Brake	Throttle			N/A

9.2.4 Retrieving Components and Creating Component Tables

After the component descriptions are parsed successfully, they are fed to a component manager. The component manager sends each component description to search. Our methodology requires that the criterion for comparing two components be to match the semantics of them and those needed components must be ready before proceeding to system synthesis. The search results are returned to the component manager and stored

in a component pool. Then the manager creates a component table for each component and stores the table in a table pool.

A component table contains the designated name from system design, the real name from software library, the constraints, the properties, the interface description, and the location information of implementation of a component. When a component is embedded into a prime connection, the system synthesizer uses the information in the table to perform the embedding correctly. A component table template is shown in Table 9.9. Every component has its own table entry.

Table 9.9 A component table template

Designated name	Real name	Constraint	Property	Interface Description	Implementation location

9.3 System Synthesis

When the system integration proceeds to system synthesis, all the required components are already stored in a component pool and their component tables are in a table pool. A system synthesizer is designed to do the system synthesis. Now the synthesizer chooses an unfinished aggregation from the system table and will translate the IDL aggregation into a source aggregation. A source aggregation is an aggregation written in a high level programming language. During the translation, the needed functional units by the aggregation are embedded into it. The functional units can be components, other aggregations or prime connections. In fact, the translation is done just based on each

prime connection in an aggregation. A source aggregation is the result of composition from its prime connections. The synthesis details are discussed below.

The dependency relationships shown in Figure 9.1 imply that, to produce source aggregations, a better way is to arrange all the aggregations in a system into a queue based on the properties of their prime connections. The proposed synthesizer follows this strategy. At first, the synthesizer picks the system table from table pool to examine how many aggregations are included in the system. If the system contains only one aggregation, such as the KWIC system in Figure 8.2(a), the synthesizer goes to translate each prime connection, embed required functional units and compose the aggregation. Otherwise, the synthesizer picks an aggregation table from the table pool to check if those prime connections in the aggregation are pure. If they are, the aggregation is put in an aggregation queue. If they are not, the prime connections are checked one by one to see if a virtual prime connection contains other aggregations. If not, the aggregation is also put in the aggregation queue. Otherwise, the synthesizer will furthermore check if those aggregations are already in the aggregation queue. If they are, the aggregation is put in the aggregation queue, too. Otherwise, the synthesizer postpones inserting the aggregation into the aggregation queue and picks another aggregation table to repeat the above procedure until all the aggregations are inserted into the aggregation queue. The synthesizer basically inserts aggregation names into the queue.

Once the synthesizer finds that there is only one aggregation in the system or all the aggregations have been inserted into the aggregation queue, it starts to translate an unfinished pure prime connection from the unique aggregation or the first aggregation in the queue. The translation will be done based on the structural type, the structural

relationship and the functional units with their contexts. The structural type suggests a program control structure to be selected, such as a sequential, a selective, or a repetitive structure. After choosing a program control structure, the structural relationship is used to organize each functional unit into the program control structure.

Since the functional units in a pure prime connection are components, the pre-context of a component is translated first and embedded into the control structure program segment. Then the synthesizer will read in the information from the component table, such as, its constraints, properties, interface descriptions and implementation location. The constraints are some conditions that need to be satisfied before the invocation of the component. The properties embody the materials related to run-time environments, such as platform, operating system, and so on. The synthesizer will compare the information with the run-time environments of the expected system. If they are not same, it needs to choose an invocation method to the component, for example, using a middleware, such as CORBA, DCOM, etc. Otherwise, the component is embedded directly. The interface descriptions are used to determine how the methods in the component are invoked. The implementation location tells the synthesizer where the component can be accessed. After embedding the component, the post-context is translated.

After a pure prime connection has been translated, the status of the prime connection in the aggregation table is changed to be 'yes' and the source code is written back to the source field in the table. Then the synthesizer checks the aggregation table again to see if there is any other pure prime connection that is still not ready. If it is, it translates the pure prime connection following the steps illustrated above. If not, the

synthesizer goes to choose a virtual prime connection that is not translated, the translation steps for a virtual prime connection are similar to the steps for a pure prime connection except embedding a functional unit. If the functional unit is a component, the synthesizer does the same thing as for a pure prime connection. If the functional unit is an aggregation, the pre-context of the aggregation is translated first. Then the source aggregation is simply embedded. Moreover, the post-context is translated. If the functional unit is a prime connection, the source code of the prime connection is directly embedded. For an aggregation, the synthesizer repeats the steps described above until all prime connections are translated. While the translation of the last prime connection is done, the source aggregation for the whole aggregation is produced. The synthesizer will change the aggregation's status in the system table to be 'yes' which means that the aggregation is ready to be used, and write the source aggregation to the aggregation's source field. Then the synthesizer deletes the aggregation from the aggregation queue and takes the new first aggregation in the queue to repeat the above procedure until the aggregation queue is empty. In fact, the source aggregation for the last aggregation is the source code for the whole system. Finally the synthesizer writes the source code for the system to a text file. For a general system, the system synthesis process expressed above can be represented in a flowchart as shown in Figure 9.3.

The system synthesis can be applied to our problem examples. For solution 1 of the KWIC system, the aggregation queue will not be created because it has only one aggregation. The source aggregation can be produced directly by translating its included prime connection. During the translation, the components Input, Shift, Sort and Output are embedded into the source code of the prime connection.

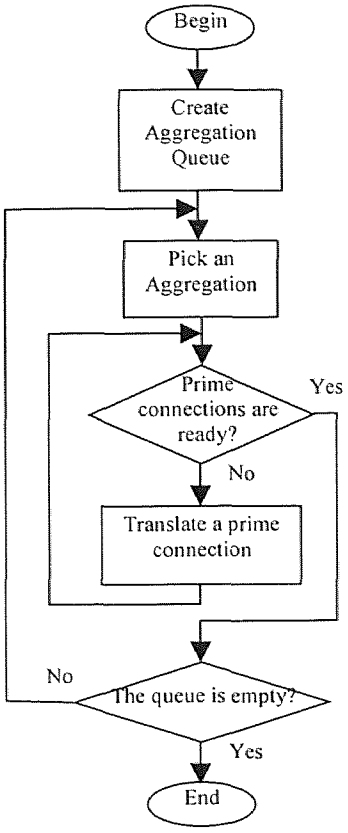

Figure 9.3 System synthesis process flowchart

For solution 2 of the KWIC system, it has two aggregations, KWIC and process. The two aggregations are inserted into the aggregation queue in which Process is the first and KWIC is the second. The synthesizer first picks the aggregation Process, to produce a source aggregation for it. Then the aggregation KWIC is selected. The synthesis details are shown in Figure 9.4. The prime connection Process_1 is translated first. Then it is inserted into prime connection Process_2 while it is translated. After that, the whole aggregation Process is embedded into prime connection KWIC, while it is translated. Finally the whole system is produced at the time the aggregation KWIC is done.

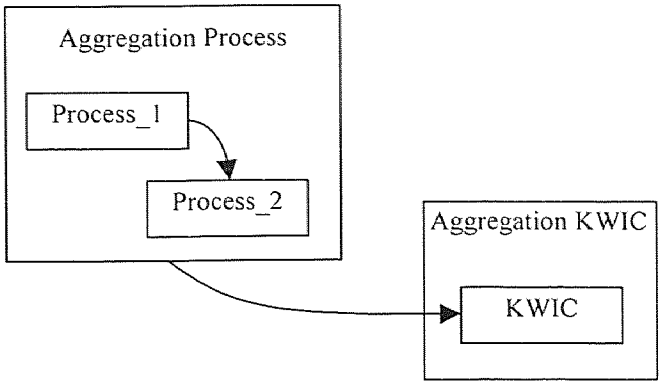

Figure 9.4 The synthesis process for solution 2 of the KWIC system

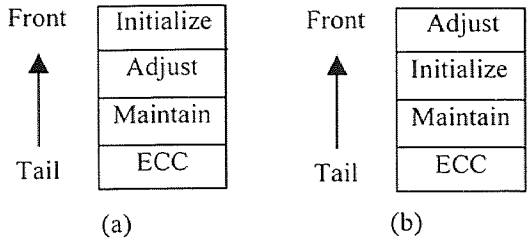

Figure 9.5 Two possible aggregation queues for the ECC system

The ECC system has four aggregations. According to the conditions for queuing the aggregations, the aggregation queue can be Figure 9.5(a) or Figure 9.5(b). Actually the synthesizer creates one of them randomly. For the final system, they are not different. Suppose that the synthesizer creates the aggregation queue in Figure 9.5(a). It translates the prime connections in aggregation Initialize and produces a source aggregation for the aggregation Initialize. Then the source aggregations are produced for aggregations Adjust and Maintain. Finally the aggregation ECC that is the whole ECC system is produced.

9.4 Summary

In this chapter, a system integration mechanism is introduced. Generally the mechanism is realized through parsing IDL descriptions, searching components, creating component tables, aggregation tables and a system table, and synthesizing the whole system by translating prime connections, embedding components and composing aggregations. The whole system integration uses a parser, a component manager, a synthesizer, an aggregation pool, a table pool and a component pool. The functionality of the parser, component manager and synthesizer is summarized below. The pools are storage spaces in computer memory.

A parser is designed to parse the IDL descriptions for the system, aggregations and components. It also creates system table and aggregation tables that are stored in the table pool. The parsed system and aggregation descriptions are stored in the aggregation pool.

The system integration needs a component manager to handle the requirements of searching, storing and managing components. The component manager accepts the parsed component descriptions in IDL. It sends the descriptions out for search one by one. The returned results are stored in the component pool. For each found component, the component manager creates a table for it and stores it in the table pool.

A synthesizer, at last, is designed to accomplish the system synthesis. It creates an aggregation queue, in order to efficiently translate the aggregations into source aggregations that are written in a high level programming language. When the queue is built, the synthesizer takes the first aggregation from the queue and chooses an unfinished prime connection to translate it. The strategy for the translation of prime connections is

to first translate all pure prime connections, and then the virtual prime connections. A required functional unit, such as a component, an aggregation, or another prime connection, is embedded into the source code of a prime connection while it is being translated. Once the source code of a prime connection is produced, the synthesizer changes its status in its aggregation table. The same thing is done for an aggregation while the source aggregation is being produced. When the source aggregation of the last aggregation is produced, the whole system has been synthesized already. Finally the source code for the system is written to a file.

CHAPTER 10

CONCLUSION AND FUTURE WORK

So far a methodology for component-based system integration has been proposed. The primary motivation for the methodology is to automatically produce a component-based software system from independently developed components. If the detailed design of a system is done and all the required components are ready, the integration methodology can be applied to automatically synthesize a software system. The chief activities for the system integration are graphically representing and decomposing a system, describing the system in the IDL, parsing the system descriptions and searching components, and synthesizing the system.

The first activity means representing the detailed design of a system with a Jackson Diagram, decomposing the Jackson Diagram to aggregations that are still represented with Jackson Diagrams, converting the Jackson Diagrams to flowgraphs that are called aggregation flowgraphs, and decomposing the aggregation flowgraphs into prime flowgraphs called prime connections. The second activity represents the results of the first activity. The results include the system, its aggregations and components that are described in the IDL. The third one means parsing the descriptions, and searching the required components. The produced results from the activity are system, aggregation and component tables stored in table pool and the IDL system and aggregations stored in aggregation pool. The last one means synthesizing aggregations from prime connections and functional units that are included in the aggregations. The synthesis is performed through translating each prime connection into source code written in a high level programming language, and embedding the included functional units. The functional

units can be components, other aggregations, or other prime connections. Actually an aggregation is composed from its prime connections based on the composition theory of flowgraphs. This is why an aggregation needs to be represented as a flowgraph in a previous stage. Although this dissertation is designated to propose the integration methodology, the research work furthermore includes the study of cubic graphs, flowgraphs, and computing cyclomatic complexity of a program based on flowgraphs because some knowledge of them is applied in our methodology.

The main research results involved in this dissertation are summarized Section 10.1. The features of the methodology are described in Section 10.2. Then the future research directions are discussed in Section 10.3.

10.1 Summary of Results

This dissertation contains comprehensive surveys of research topics related to the proposed methodology, and a number of research results included in the methodology. The surveyed research topics are cubic graphs, flowgraphs, cubic flowgraphs, software architectures, and component-based software. Theses surveyed topics are not summarized here. We merely summarize the research results in this dissertation. The results reported here are novel and original, and to the best of our knowledge have not published before.

10.1.1 Cyclomatic Complexity

The research work began with the study of the relationships between cyclomatic complexity and flowgraphs. As we surveyed in Chapter 5, cyclomatic complexity is a measure value for computer programs. This value is used to count how many paths

included in a program need to be tested. Furthermore, since the cyclomatic complexity is from a concept, cyclomatic number, of graph theory, it is reasonable to connect the cyclomatic complexity to a graph. In our work, we use flowgraphs because any program can be represented as a flowgraph in which its properties are kept unchanged. Typically we use cubic flowgraphs of programs to compute their cyclomatic complexity because every flowgraph has a unique cubic flowgraph in which the properties of the original flowgraph are kept unchanged except for ignoring the process nodes with degree 2.

Based on these conditions, we have developed new ways to compute the cyclomatic complexity of computer programs. These new ways are expressed in Theorem 1, Corollary 2, Corollary 3, Theorem 4, and Corollary 5 in Chapter 5. The final conclusions are that the cyclomatic complexity of a program is equal to the cyclomatic number of its cubic flowgraph and the cyclomatic complexity of a standard structured program is equal to the number of prime flowgraphs in the program.

In order to compute cyclomatic complexity of a program, the number of decision-making conditions in the program has to be counted because the cyclomatic complexity of the program is the number plus one [27]. In contrast to this method, our new ways are accurate and unambiguous if a cubic flowgraph of a program is created. In addition, it is easier to create a cubic flowgraph of a program than to create a program control graph.

10.1.2 Software Architectures as Aggregations

The aggregation view on software architectures is one of research results in this work. A software system consists of a number of components and the interactions among the components in terms of a component-based software style. The interactions are

abstracted as connectors. Therefore, for a general large system, the system can be decomposed into subsystems, while a subsystem can be decomposed into connectors. Each connector connects a number of components. Such a containing relationship between a system and its subsystems, a subsystem and its connectors, a connector and its components is actually an aggregate relationship because this relationship means more than simply containing. An aggregate relationship is called an aggregation in our work. Such a perspective of software architectures is called an aggregation view.

We have surveyed other architectural views in Chapter 3. These views are conceptual view, module view, process view, physical view and framework view. None of them provides a clear way to decompose a large-scale system into small-scale functional units in order to implement the system. Aggregation view, however, does not only provide a way to decompose a system into aggregations, but also provide a way to recognize composite components in an existing system and to organize composite components in a new system when it is designed. Moreover, the aggregation view can be applied to describe functional units with different granularity.

10.1.3 Component Model

Another research result in this work is a component model. It can be used for dealing with components in software libraries and for component search. The comprehensive component model covers the type of a component, constraints and properties of a component, methods and events that are services provided by a component, and the behavioral specifications for the methods and events.

This component model is based on the previous work surveyed in Chapter 4. One of the advantages of this model is that it describes every aspects of a general component. It can be used to represent a component in a software library and to describe a component that needs to be searched.

10.1.4 System Representation and Decomposition

In this work, Jackson Diagrams are used to provide a graphical representation for a system and its aggregations because an aggregation is explicitly expressed in Jackson Diagrams. When a system is represented with a Jackson Diagram, it is easy to decompose the system into aggregations. Then flowgraphs are applied to represent each aggregation in order to decompose an aggregation into prime flowgraphs. A prime flowgraph in a system is called a prime connection. Thus these representations enhance the aggregation view on software architectures.

A software system is composed of aggregations, while an aggregation is composed of prime connections in terms of the aggregation view. We also propose a decomposition strategy for prime connections, namely the integrity decomposition strategy to avoid the division of contexts of a prime connection. A prime connection is distinguished as a pure one or virtual one in order to translate them into source code conveniently.

The strategies for system representation and decomposition proposed here enhance the aggregation view and the application of decomposition and composition theory of flowgraphs. The strategies hold a potential for the automation of the decomposition. Furthermore, it is easy to learn, understand and apply these strategies.

10.1.5 Implementation Description Language

An implementation description language (IDL) is proposed based on the aggregation view and component model. The IDL can be used to describe a system to indicate what aggregations and components are included in the system. The descriptions of aggregations and components are also covered by the IDL. The purpose of describing an aggregation is to synthesize it from its prime connections. The purpose of describing a component is to search the component from a software library.

The IDL is a prototype of future component-based programming languages. It serves as an intermediary tool between architecture description languages and current programming languages. An obvious feature of the IDL is that it focuses on the interactions or interdependencies of components.

10.1.6 System Integration Mechanism

The last result is a system integration mechanism. It is designed to accomplish the final integration mission for a component-based software system. The mechanism is actually an integrated environment including a parser, a component manager and a synthesizer.

The parser is designed to parse a system description, aggregation descriptions and component descriptions. When no more syntax errors are found, the parser forwards the component descriptions to the component manager for component search, and creates system and aggregation tables. The component manager accepts syntactically correct component descriptions, separates them and sends them to a separable search engine one by one. The returned components are stored in a component pool and a component table is created for each component. The synthesizer sorts the aggregations into an order and

inserts the names of the aggregations into a queue according to the dependency relationships among the aggregations. Then the aggregations in the queue are translated one by one into source aggregations written in a high level programming language until the queue is empty.

An ideal integration mechanism is based on semantics-driven composition. Currently it is hard to develop such a mechanism because there is no a comprehensive way to represent the semantics of a component and a system so that a computer can understand the representations. The proposed integration mechanism does not attack the semantics-driven composition directly. It is based on the composition theory of flowgraphs and successfully accomplishes the integration mission.

Although the work primarily concentrates on component-based software system integration, the theoretic bases are also explored. The theoretic work combined with the integration mechanism constitutes this complete methodology. The contributions made in this work are described below:

- Computing the cyclomatic complexity based on cubic flowgraphs;
- A view on software architectures called aggregation view;
- A comprehensive component model;
- Decomposing a software architecture into aggregations;
- Representing the aggregations as flowgraphs and decomposing the aggregation flowgraphs into prime connections;

- Around the aggregations, prime connections and components, proposing an implementation description language, IDL; and,
- A system synthesis mechanism.

10.2 Features of the Methodology

As we surveyed in Chapter 4, the existing component-based software development methodologies are centered on architecture description languages. These methodologies largely consist of requirements analysis, functionalities refinement, software architecture selection or design, architecture description, and code production from the description. Although they are different in details, these methodologies have as a common property their dependence on the types of components, the types of connectors or interaction protocols, and architectural styles.

Our methodology focuses on system integration from components. In other words, it is developed to aid the implementation of a software system with a component-based style. It accepts the design results produced in any methodology, and then represents the design details in its own way based on an aggregation view. Hence, there is no contradiction or conflict between existing methodologies and our methodology, it is a complementary to others. For our methodology, it does not require that the interactions among components must be implemented with any existing communication protocols bound to specific architectural styles. It accommodates both existing communication protocols and newly developed communication protocols. In addition, the IDL introduced in this dissertation serves as an implementation description so that it is easily

translated into current high level programming languages. Therefore, our methodology offers itself with flexibility, practical utility, effectiveness, and applicability.

Flexibility means that our methodology does not confine itself to any special software architectural styles. There are no prerequisites on the types of components and the types of connectors. It accepts the design results with any architectural styles, any types of components, and any types of connectors.

Practical utility hints that the methods used in the methodology are practical and easy to learn and apply. For example, it is easy to understand and apply the aggregation view, graphical representations, and the IDL. It is not necessary for system designers or software architects to know the details of the integration mechanism.

Effectiveness indicates that our methodology can be used to solve any integration problems for a component-based software system effectively. No matter what software architectural styles and types of components and connectors the detailed design of a system is based on, it can be expressed as a set of aggregations according to aggregation view, and be represented in the IDL. It can also be synthesized from components and translated into a chosen high level programming language.

Applicability means that our methodology is suitable for component-based software system integration because it does not contradict other component-based software development methodologies, but also complements them for the automatic production of a component-based software system. The existing development methodologies are weak in the integration step since the current architecture description languages are too abstract to catch the design details. Our methodology overcomes this

drawback and provides a way to bridge the gap between the current architecture description languages and current high level programming languages.

Since the current architecture description languages do not describe the implementation details of a system, most of them make use of connectors to catch the interactions between or among components. It means that the supported connectors in an ADL must be well-defined to include the interactive information. This is why they only support pre-defined types of connectors. Our IDL focuses on the interactions of components without any limitations on the types of connectors. It accepts any descriptions in ADLs and converts these descriptions into its own descriptions based on the detailed design of a system. In addition, most current programming languages use three program control structures. These structures are supported by our IDL. Therefore, it is easier to translate the IDL descriptions into a high-level programming language programs.

These features make our methodology different from existing methodologies when integrating software systems from components. With these features, our methodology has the potential to overcome one of the bottlenecks, system integration. Although the composition of prime connections is not a semantics-based composition, it keeps the semantics of each aggregation unchanged.

10.3 Future Work

The methodology that is proposed in this dissertation still needs to be improved despite the important features mentioned in Section 10.2. Improvements are possible on supporting tools for the whole methodology, the current IDL as a prototype for

component-based programming languages, and the studies of interactions among the components. These three future research directions are related each other. Actually the last one is more important than the first two because it can serve as a base to automatically compose or integrate a software system from components. It aims to overcome the weaknesses of current high level programming languages because they embed the interactions in each component so that the reuse of components becomes harder and harder. This section focuses on these future research directions.

10.3.1 Supporting Tools

One of the future directions is to develop a set of tools to support the whole process of the methodology. The set of supporting tools consists of a tool for decomposing software architectures into aggregations and for converting Jackson Diagrams to flowgraphs, a tool for decomposing an aggregation into prime connections and for describing aggregations in the IDL, and a tool for describing components in the IDL. The details of the tools are discussed one by one here.

Our methodology suggests that the detailed design of a system be represented as a Jackson Diagram. While a Jackson Diagram expresses an aggregation explicitly, we represent the Jackson Diagram as a tree in which a non-leaves node with its children is an aggregation. Clearly it is possible to design a tool that accepts a tree as its input, and computes each subtree in the tree that consists of a non-leaves node and all its children. The tool is responsible for decomposing a Jackson Diagram into aggregations. The Jackson Diagram is really the system architecture. After the decomposition, each aggregation is still represented as a two-level Jackson Diagram. Then the tool converts each aggregation into an aggregation flowgraph.

At present, each aggregation is represented as a flowgraph. The decomposition of a flowgraph into a number of prime flowgraphs can be done by a tool based on the decomposition theory discussed in Chapter 2. This tool can also be extended to describe each prime connection in the aggregation, and the whole aggregation in the IDL according to the properties of each prime connection and the syntax of the IDL. The tool may work interactively by accepting some input from a user.

The next tool is for describing components included in a system. The tool provides an interactive working environment for a user to input the required information of a component. Then the IDL description for a component is produced. Combined with the parser, component manager and synthesizer of the methodology, these tools can make the whole integration process automatic.

10.3.2 Component-based Programming Languages

An observation on the methodology and the IDL is whether it is possible to have a language in which a system, its aggregations and components are described and that can be compiled into executables directly. Obviously the current IDL cannot be used to serve the purpose because the descriptions of contexts of a functional unit and the behavioral specifications of a component are not covered by the current IDL. Do current architecture description languages serve the purpose? It is partially possible, but not totally because current architecture description languages are used to represent high level abstraction of software architectures. The details of rich interactions among components, however, usually cannot be described in them.

Therefore, an emerging trend in component-based software engineering is to develop component-based programming languages. Such languages focus on the specification of interactions among the components while the components are binary executables. They have explicit mechanisms for describing interactions, instead of implying the interactions in components like the current programming languages. They use components as same as a statement the current programming languages use. Such a language is defined as a component-based programming language. The development of such a language is one of the future research directions of this work.

10.3.3 Studies of Interactions among Components

It is important to know the type and properties of interactions between components in order to automatically produce software systems from components, especially to develop component-based programming languages. If we had known all these interactions, we could classify them into different types according to their properties. For each type, its details may be defined in advance if there were no interactive types we do not know. In fact, we just know a few of them. This is why studying interactions among components is one of the future research directions in the work.

Since the separation of components from their interactions is a fundamental requirement for components to be reused, studying interactions among components is the most important step towards the ideal objective of component-based software, that is, automated composition of components. Therefore, for the future work proposed in Section 10.3, the last future research direction is a base for the first two.

10.4 Epilogue

The research results in this dissertation are summarized in Section 10.1. The features of this proposed methodology are introduced in Section 10.2. The future research directions are discussed in Section 10.3. This section serves as an epilogue of the whole dissertation.

The component-based system integration is a critical field in component-based software engineering. It is drawing more and more attentions from the software engineering community. In this dissertation a methodology for component-based system integration is proposed. This methodology chiefly consists of system representation and decomposition, system description and component search, and system integration. During the development of the methodology, the research results obtained include new ways to compute cyclomatic complexity of a program, an aggregation view of software architectures, a component model, a system representation and decomposition mechanism, an implementation description language and a system integration mechanism. This methodology can be applied to accomplish component-based system integration automatically.

APPENDIX A

THE GRAMMAR OF IDL

The syntax of IDL is specified in an Extended Backus-Naur Form (EBNF). The EBNF specification consists of a collection of rules or productions. Each production consists of a non-terminal symbol and an EBNF expression separated by a special meta-symbol sign (::=) and terminated with a period (.). The non-terminal symbol is a "meta-identifier", and the EBNF expression is its definition. The meta-symbols used in the EBNF expression is summarized the table below. Then the syntax productions are listed.

Meta-symbol	Meaning
::=	is defined to be
|	alternatively
.	end of production
[X]	0 or 1 instance of X
{X}	0 or more instances of X
(X|Y)	a grouping: either X or Y
<X>	The non-terminal symbol X
"XYZ"	the terminal symbol XYZ

System description ::= System <system name> '{'

Aggregation <aggregation name>{',' <aggregation name>}';'

Component <component name>{',' <component name>}';'

'}'.

Aggregation description ::= Aggregation < aggregation name>‘{’

{<Prime connection>‘;’}

‘}’.

Prime connection ::= Prime connection <prime connection name> ‘{’

Type (‘pure’ | ‘virtual’)‘;’

Structural type (‘sequential’| ‘selective’ | ‘repetitive’)‘;’

Structural relationship <regular expression>‘;’

Logical condition <boolean expression> [with

Prerequisite ‘{’<prerequisite specification>‘}’] ‘;’

{Component <component name> ‘{’

[Pre-context ‘{’<specification for pre-context>‘}’ ‘;’]

[Post-context ‘{’< specification for post-context>‘}’ ‘;’]

‘}’‘;’}

{Aggregation <aggregation name>‘{’

[Pre-context ‘{’<specification for pre-context>‘}’ ‘;’]

[Post-context ‘{’< specification for post-context>‘}’ ‘;’]

‘}’‘;’}

[Prime connection <prime connection name list>‘;’]

‘}’.

system name ::= <identifier>.

aggregation name ::= <identifier>.

component name ::= <identifier>.

prime connection name ::= <identifier>.

prime connection name list ::= <prime connection name>{, <prime connection name>}.

Component ::= Component <component name> '{'

Type <component type> ';'

{Method '{'

Type <data type>';'

Signature <method name>'('<argument list>')' ';'

Informal specification '{'<text>'}'';' †

Formal specification '{'<behavior specification>'}' ';'

'}'}

{Consumed event <event name> '{'

Informal specification '{'<text>'}' ';' †

Formal specification '{'< behavior specification>'}' ';'

'}'}

[Generated event <event name list>';']

'}'.

† No definition for text that is default to be English.

component type ::= ‘filter’ | ‘server’ | ‘procedure’ | ‘module’ | ‘executable’

| ‘gui-function’ | ‘user-defined’.

method name ::= <identifier>.

argument list ::= <data type>{‘,’ <data type>}.

event name ::= <identifier>.

prerequisite specification ::= <a formal specification>. ‡

specification for pre-context ::= <a formal specification>. ‡

specification for post-context ::= <a formal specification>. ‡

behavior specification ::= <a formal specification>. ‡

event name list ::= <event name>{‘,’ <event name>}.

data type ::= ‘int’ | ‘float’ | ‘char’ | ‘string’ | ‘void’.

identifier ::= <letter>{<letter> | <digit> | _}.

letter::=‘A’|‘B’|‘C’|‘D’|‘E’|‘F’|‘G’|‘H’|‘I’|‘J’|‘K’|‘L’|‘M’|

‘N’|‘O’|‘P’|‘Q’|‘R’|‘S’|‘T’|‘U’|‘V’|‘W’|‘X’|‘Y’|‘Z’.

digit ::= ‘0’|‘1’|‘2’|‘3’|‘4’|‘5’|‘6’|‘7’|‘8’|‘9’.

integer ::=<digit>{<digit>}.

regular expression ::= <identifier> | ‘(’<regular expression><regular expression> ‘)’ |

‘(’<regular expression> + <regular expression>‘)’|

‘(’<regular expression> * <regular expression>‘)’.

‡ It is left open for users.

boolean expression ::= 'TRUE' | 'FALSE' | <relational expression> |

['('<boolean expression>')' <logic operator>] '('<boolean expression>')'.

logic operator ::= 'AND' | 'OR' | 'NOT'.

relational operator ::= '<' | '<=' | '>' | '>=' | '==' | '!='.

relational expression ::= <simple expression> < relational operator> <simple expression>|

(<relational expression>).

simple expression ::= <constant> | <variable> |

<simple expression> <operator> <simple expression> |

(<simple expression>).

operator ::= '+' | '–' | '*' | '/' | '%'.

constant ::= [<sign>] <digit>{<digit>}['.' <digit>{<digit>}].

sign ::= '+' | '–'.

variable ::= <identifier>.

APPENDIX B

CONVERSION OF CUBIC GRAPHS INTO FLOWGRAPHS

We have discussed how to convert a flowchart into a flowgraph and how to convert a flowgraph into a cubic flowgraph and finally a cubic graph in Chapter 2. Here we discuss how to convert a cubic graph to flowgraphs. This conversion can produce prime program control structures that may not structured.

B.1 Conversion Methods

The generation of flowgraphs from a cubic graph is discussed in this section. Since each vertex in a cubic graph has degree 3, every cubic graph G has an even number of vertices. Let us assume that a cubic graph has 2n vertices where n can be 0, 1, 2, ... Thus the cubic graph has 3n edges.

A cubic flowgraph is a strongly connected digraph with two kinds of vertices: decision vertices and junction vertices. First of all, we select half of the vertices in a given cubic graph as decision vertices colored black. The rest of the vertices are the junction vertices, left white. If a cubic graph has 2n vertices, then there are (2n)!/((n!)*(n!)) new cubic graphs with half black vertices and half white vertices. We call the graphs black-white cubic graphs. For each black-white cubic graph, we need to test whether there is a chromatic circuit in it because if there is such a circuit, the cubic graph can not produce a flowgraph [61]. A chromatic circuit is a cycle on which all the nodes have same color. The black-white cubic graphs including a chromatic circuit are removed.

For each black-white cubic graph, we need to convert it into a direct graph. We start to add orientations to a black-white cubic graph from three edges incident to a black vertex. The three edges are one-in and two-out. Therefore there are three situations. If we number the three edges as 1, 2, 3, we have three situations shown in Figure B.1.

IN	**OUT**	**OUT**
OUT	**IN**	**OUT**
OUT	**OUT**	**IN**

Figure B.1 A black node's in-out combinations

Generally each black vertex has the three situations. But actually if two vertices are adjacent, the situations of second vertex are dependent on the first one's. We take one situation for the black vertex to process. Then we proceed with all other black vertices. Finally we process all the white vertices. For each white vertex, there are three situations shown in Figure B.2 because each white vertex has two-in and one-out.

IN	**IN**	**OUT**
IN	**OUT**	**IN**
OUT	**IN**	**IN**

Figure B.2 A white node's in-out combinations

The above procedure is repeated until all the situations are processed. During the converting procedure, we note that more than one directed black-white cubic graph can be produced from a black-white cubic graph. After all the directed black-white cubic graphs are generated, we check the isomorphism among them. It is not necessary to keep all the direct black-white graphs. We just keep one from each isomorphic group. Now

we can generate flowgraphs from the remaining directed black-white graphs by picking one arc as the special arc from *End* vertex to *Begin* vertex.

B.2 Demonstration by an Example

If a cubic graph has 2n vertices, then (2n)!/((n!)*(n!)) black-white cubic graphs are produced from it. Generally a black-white cubic graph can lead to several flowgraphs. For example, for the cubic graph G_2 with 4 vertices in Figure 2.6, there are 6 black-white cubic graphs (Figure B.3) from the graph, each of which can produce 24 direct black-white cubic graphs.

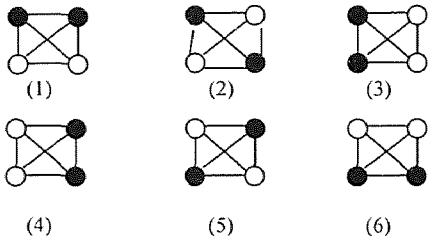

Figure B.3 Black-white cubic graphs

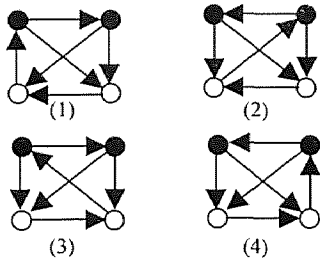

Figure B.4 Black-white cubic digraphs

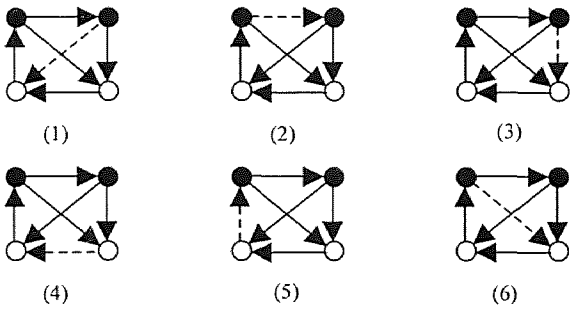

Figure B.5 Produced cubic flowgraphs

Figure B.4 shows 4 direct black-white cubic graphs produced from the first black-white cubic graph in Figure B.3. In this way there are 144 flowgraphs. Figure B.5 shows 6 flowgraphs produced from the direct black-white cubic graph (1) in Figure B.4.

B.3 Black-white Cubic Graphs and Digraphs

When we implemented the procedure above in a Java program, we found that if two black-white cubic graphs are isomorphic, they will produce the same flowgraph. In fact, the 6 black-white cubic graphs from a cubic graph with 4 vertices are isomorphic. Another observation is that a black-white cubic graph can produce isomorphic directed black-white cubic graphs. In the example mentioned above, all the directed black-white cubic graphs are isomorphic. Therefore there is one useful directed black-white cubic graph. At last we get 6 flowgraphs (Figure B.5) from the directed black-white cubic graph. The observations are concluded as two propositions.

Proposition 1

If two black-white cubic graphs are isomorphic, the directed black-white cubic graphs produced from them are isomorphic.

Proposition 2

If two directed black-white cubic graphs are isomorphic, the flowgraphs produced from them are isomorphic.

The two propositions are very important when we produce flowgraphs from a cubic graph because we just need to keep the non-isomorphic black-white cubic graphs and the non-isomorphic directed black-white cubic graphs.

B.4 Conversion Program in Java

Source Codes File 1: cubic_test.java

```
import java.io.*;

public class cubic_test {

  public static void main( String args[] ) throws IOException

  {

    int data[][];

    cubic c;

    data = new int[4][5];

    c = new cubic();

    for (int i=0; i<data.length; i++)

      for (int j=0; j<data[i].length; j++)

        data[i][j]=0;

    for (int i=0; i<data.length; i++)

      for (int j=0; j<data.length; j++)

      {

        System.out.print("Enter A Value for Graph in Position "+i+","+j+":");

        System.out.flush();

        data[i][j]=System.in.read();
```

```
            System.in.skip(1);
            data[i][j]=data[i][j]-48;
          }
      c.SetGraph(data);
      System.out.println(" The Original Graph:");
      c.GetGraph();
      System.out.println("                    ");
      c.BlackWhiteG();
   }
}
```

Source Codes File 2: cubic.java

```
import java.io.*;
public class cubic {
   private int graph[][];
   private int base[][];
   private int first_digraph;
   public  cubic()
   {
      graph= new int[4][5];
```

```
  for (int i=0; i<graph.length; i++)
    for (int j=0;j<graph[i].length; j++)
      base= new int[4][5];
  for (int i=0; i<base.length; i++)
    for (int j=0;j<base[i].length; j++)
      base[i][j]=0;
  first_digraph=0;
}

public void SetGraph(int element[][])
{
   for (int i=0; i<element.length; i++)
    for (int j=0;j<element[i].length; j++)
      graph[i][j]=element[i][j];
}
public void CopyGraph(int element [][])
{
   for (int i=0; i<graph.length; i++)
    for (int j=0;j<graph[i].length; j++)
```

```
        element[i][j]=graph[i][j];
}
public void GetGraph()
{
  for (int i=0; i<graph.length; i++)
  {
    for (int j=0; j<graph[i].length; j++)
    {
      System.out.print(graph[i][j]+" ");
      System.out.flush();
    };
    System.out.println();
  }
}
public void BlackWhiteG()
{
  int NumNodes=graph.length;
  int copy[][], number=0;
  copy = new int[graph.length][graph.length+1];
```

```
  CopyGraph(copy);
  for (int i=0; i<NumNodes-1; i++)
    for (int j=i+1; j<NumNodes; j++)
    {
       if (i==0) number=j;
       if (i==1 && j==2) number=4;
       if (i==1 && j==3) number=5;
       if (i==2 && j==3) number=6;
       SetGraph(copy);
       graph[i][4]=1;
       graph[j][4]=1;
       System.out.println("Number "+number+" Black-White Graph:");
       GetGraph();
       System.out.println("                                    ");
       Digraph(i,j);
    }
    SetGraph(copy);
}
public void Digraph(int black1, int black2)
```

```
{
  int copy[][];
  int mark=1;
  copy = new int[graph.length][graph.length+1];
  CopyGraph( copy );
  for (int situations=1; situations<4; situations++)
  {
    if (situations==1)
    {
      for (int j=0; j<graph.length; j++)
        if (graph[black1][j]==1 && mark==1)
        {
          mark=2;
          graph[black1][j]=0;
          graph[j][black1]=1;
        }
        else if (graph[black1][j]==1 && mark==2)
              graph[j][black1]=0;
      ProduceDigraph(black1,black2);
```

```
  SetGraph(copy);
  mark=1;
};
if (situations==2)
{
  for (int j=0; j<graph.length; j++)
    if (graph[black1][j]==1 && mark==1)
    {
      graph[j][black1]=0;
      mark=2;
    }
    else if (graph[black1][j]==1 && mark==2)
    {
      mark=1;
      graph[black1][j]=0;
      graph[j][black1]=1;
    };
  ProduceDigraph(black1,black2);
  SetGraph(copy);
```

```
        mark=2;
      };
    if (situations==3)
    {
      for (int j=0; j<graph.length; j++)
        if (graph[black1][j]==1 && mark!=0)
        {
          mark--;
          graph[j][black1]=0;
        }
        else if (graph[black1][j]==1 && mark==0)
        {
          graph[black1][j]=0;
          graph[j][black1]=1;
        };
      ProduceDigraph(black1,black2);
    }
  }
}
```

```
public void ProduceDigraph(int b1, int b2)
{
  int i=0, j=0, count=0, first=0, second=0, indicator[];
  int dup[][],w1=0,w2=0, count_w1=0,count_w2=0,index_w1=0,index_w2=0;
  dup = new int [graph.length][graph.length+1];
  indicator = new int [graph.length];
  CopyGraph(dup);
  for (i=0; i<graph.length; i++)
     indicator[i]=0;
  for (j=0; j<graph.length; j++)
     if (graph[b2][j]==1) indicator[j]=1;
  for (i=0; i<graph.length; i++)
     if (indicator[i]!=0) count++;
  if (count==3){
     for (i=0; i<graph.length; i++)
        if (indicator[i]==1 && i==b1) indicator[i]=0;
     for (i=0; i<graph.length; i++)
        if (indicator[i]==1 && first==0) first=i;
        else if (indicator[i]==1) second=i;
```

```
graph[b2][first]=0;

graph[first][b2]=1;

graph[second][b2]=0;

for (j=0; j<graph.length; j++)     // Begin to Process White Nodes

   if (graph[j][4]==0 && w1==0) w1=j;

   else if (graph[j][4]==0) w2=j;

for (j=0; j<graph.length; j++)

   if (graph[w1][j]==1) {

      count_w1++;

      index_w1=j;

   }

for (j=0; j<graph.length; j++)

   if (graph[w2][j]==1){

      count_w2++;

      index_w2=j;

   }

if (count_w1==1) graph[index_w1][w1]=0;

else if (count_w2==1) graph[index_w2][w2]=0;

System.out.println("****** A Produced Digraph ******");
```

```
GetGraph();      // End to Process White Nodes

System.out.println("*********************************");

if (first_digraph==0) {

  first_digraph=1;

  CopyGraph(base);

  rearrange(base);

}

else  isomorphism_check();

w1=0;

w2=0;

count_w1=0;

count_w2=0;

index_w1=0;

index_w2=0;

SetGraph(dup);

graph[b2][second]=0;

graph[second][b2]=1;

graph[first][b2]=0;

for (j=0; j<graph.length; j++)
```

```
        if (graph[j][4]==0 && w1==0) w1=j;
        else if (graph[j][4]==0) w2=j;
    for (j=0; j<graph.length; j++)
        if (graph[w1][j]==1) {
            count_w1++;
            index_w1=j;
        }
    for (j=0; j<graph.length; j++)
        if (graph[w2][j]==1){
            count_w2++;
            index_w2=j;
        }
    if (count_w1==1) graph[index_w1][w1]=0;
    else if (count_w2==1) graph[index_w2][w2]=0;
    System.out.println("****** A Produced Digraph ******");
    GetGraph();      // End to Process White Nodes
    System.out.println("********************************");
    isomorphism_check();
}
```

```
else
{
   for (i=0; i<graph.length; i++)
      if (indicator[i]==1 && first==0) first=i;
      else if (indicator[i]==1) second=i;
   graph[first][b2]=0;
   graph[second][b2]=0;
   for (j=0; j<graph.length; j++)
      if (graph[j][4]==0 && w1==0) w1=j;
      else if (graph[j][4]==0) w2=j;
   for (j=0; j<graph.length; j++)
      if (graph[w1][j]==1) {
        count_w1++;
        index_w1=j;
      }
   for (j=0; j<graph.length; j++)
      if (graph[w2][j]==1){
        count_w2++;
        index_w2=j;
```

```
        }
      if (count_w1==1) graph[index_w1][w1]=0;
      else if (count_w2==1) graph[index_w2][w2]=0;
      System.out.println("****** A Produced Digraph ******");
      GetGraph();       // End to Process White Nodes
      System.out.println("********************************");
      if (first_digraph==0) {
        first_digraph=1;
        CopyGraph(base);
        rearrange(base);
      }
      else isomorphism_check();
    }
}
public void rearrange ( int digraph [][] )
{
   int degree[], hold=0, k2=0, k3=0, p1=999, p2=0;
   degree = new int [graph.length];
   for (int d=0; d<graph.length; d++)
```

```
  degree[d]=0;
for (int i=0; i<graph.length; i++)
  for (int j=0; j<graph.length; j++)
    if ( digraph [i][j]==1 ) degree[i]++;
for (k2=0; k2<graph.length; k2++)
 if (degree[k2]==1 && p1==999) p1=k2;
 else if (degree[k2]==1) p2=k2;
if (p1!=0) {
  for (k3=0; k3<graph.length+1; k3++)
  {
    hold=digraph[p1][k3];
    digraph[p1][k3]=digraph[0][k3];
    digraph[0][k3]=hold;
  }
  for (k3=0; k3<graph.length; k3++)
  {
    hold=digraph[k3][p1];
    digraph[k3][p1]=digraph[k3][0];
    digraph[k3][0]=hold;
```

```
    }
  }
  if (p2!=0) {
    for (k3=0; k3<graph.length+1; k3++)
    {
      hold=digraph[p2][k3];
      digraph[p2][k3]=digraph[1][k3];
      digraph[1][k3]=hold;
    }
    for (k3=0; k3<graph.length; k3++)
    {
      hold=digraph[k3][p2];
      digraph[k3][p2]=digraph[k3][1];
      digraph[k3][1]=hold;
    }
  }
}
public void isomorphism_check()
{
```

```
int cpy[][], mark=0, k1=0, k2=0, k=0, hold=0;

int original[][];

original = new int [graph.length][graph.length+1];

cpy = new int [graph.length][graph.length+1];

CopyGraph(original);

rearrange(original);

CopyGraph(cpy);

rearrange(cpy);

for (k1=0; k1<graph.length; k1++) {

  for (k2=0; k2<graph.length; k2++)

    if (base[k1][k2]!=cpy[k1][k2])

    {

      mark=1;

      break;

    }

  if (mark==1) break;

};

if (mark!=0) {

  mark=0;
```

```
for (k=0; k<graph.length+1; k++) {
   hold=cpy[2][k];
   cpy[2][k]=cpy[3][k];
   cpy[3][k]=hold;
};
for (k=0; k<graph.length; k++) {
   hold=cpy[k][2];
   cpy[k][2]=cpy[k][3];
   cpy[k][3]=hold;
};
for (k1=0; k1<graph.length; k1++) {
  for (k2=0; k2<graph.length; k2++)
    if (base[k1][k2]!=cpy[k1][k2])
    {
      mark=1;
      break;
    }
  if (mark==1) break;
}
```

```
};
if (mark!=0) {
  mark=0;
  for (k1=0; k1<graph.length; k1++)
    for (k2=0; k2<graph.length+1; k2++)
      cpy[k1][k2]=original[k1][k2];
  for (k=0; k<graph.length+1; k++) {
    hold=cpy[0][k];
    cpy[0][k]=cpy[1][k];
    cpy[1][k]=hold;
  };
  for (k=0; k<graph.length; k++) {
    hold=cpy[k][0];
    cpy[k][0]=cpy[k][1];
    cpy[k][1]=hold;
  };
  for (k1=0; k1<graph.length; k1++) {
    for (k2=0; k2<graph.length; k2++)
      if (base[k1][k2]!=cpy[k1][k2])
```

```
        {
          mark=1;
          break;
        }
      if (mark==1) break;
    }
};
if (mark!=0) {
  mark=0;
  for (k=0; k<graph.length+1; k++) {
     hold=cpy[2][k];
     cpy[2][k]=cpy[3][k];
     cpy[3][k]=hold;
  };
  for (k=0; k<graph.length; k++) {
     hold=cpy[k][2];
     cpy[k][2]=cpy[k][3];
     cpy[k][3]=hold;
  };
```

```
      for (k1=0; k1<graph.length; k1++) {
        for (k2=0; k2<graph.length; k2++)
          if (base[k1][k2]!=cpy[k1][k2])
          {
            mark=1;
            break;
          }
        if (mark==1) break;
      }
    };
    if (mark==0) System.out.println("The Digraphs Are Isomorphic.");
    else
      System.out.println("The Digraphs Are Not Isomorphic.");
  }
}
```

REFERENCES

1. G. Abowd, R. Allen and D. Garlan, "Formalizing Style to Understand Descriptions of Software Architecture," *ACM Trans. Software Engineering and Methodology*, Vol. 4, No. 4, pp. 319-364, 1995.

2. R. Allen and D. Garlan, "Formalizing Architectural Connection," in *Proceedings of the Sixteenth International Conference on Software Engineering*, Sorrento, Italy, pp. 71-80, 1994.

3. R. Allen and D. Garlan, "A Formal Basis for Architectural Connection," *ACM Transactions on Software Engineering and Methodology*, Vol. 6, No. 3, pp. 213-249, 1997.

4. L. Bass, P. Clements and R. Kazman, *Software Architecture in Practice*, Addison-Wesley, Reading, MA, 1997.

5. P. Bassett, "The Theory and Practice of Adaptive Reuse," *ACM Software Engineering Notes*, Vol. 22, No. 3, pp. 2-9, 1997.

6. U. Bellur, "The Role of Components & Standards in Software Reuse," *Workshop on Compositional Software Architectures*, Monterey, California, January 6-8, 1998.

7. B. Boehm, "Engineering Context (for Software Architecture)," Invited talk, *First International Workshop on Architecture for Software Systems*, Seattle, Washington, April 1995.

8. F. Bronsard, D. Bryan, W. Kozaczynski, E. Liongosari, J. Ning, A. Olafsson, and J. Wetterstrand, "Toward Software Plug-and-Play," *ACM Software Engineering Notes*, Vol. 22, No. 3, pp. 19-29, 1997.

9. A. Brown and K. Short, "On Components and Objects: The Foundations of Component-Based Development," in *Proceedings of the Fifth International Symposium on Assessment of Software Tools*, Pittsburgh, PA, pp. 112-121, June 2-5, 1997.

10. P. C. Clements, "A survey of Architecture Description Languages," in *Proceedings of the Eighth International Workshop on Software Specification and Design*, Paderborn, Germany, March 1996.

11. P. C. Clements, "Coming Attractions in Software Architecture," *Technical Report*, CMU/SEI-96-TR-008, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, January 1996.

12. P. C. Clements and L. M. Nothrop, "Software Architecture: An Executive Overview," *Technical Report*, CMU/SEI-96-TR-003, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1996.

13. D. Cooke, A. Gates, E. Demirors, O. Demirors, and M. M. Tanik, "Languages for the Specification of Software," *J. Systems Software*, Elsevier Science Inc., New York, New York, Vol. 32, pp. 269-308, 1996.

14. R. Crispen and L. Stuckey, "Structural Model: Architecture for Software Designers," in *Proceedings of TRI-Ada'94*, Baltimore Convention Center, Maryland, November 1994.

15. C. N. Dellarocas, "A Coordination Perspective on Software Architecture: Towards a Design Handbook for Integrating Software Components," *Ph. D. Dissertation*, Massachusetts Institute of Technology, February 1996.

16. C. N. Dellarocas, "Toward a Design Handbook for Integrating Software Components," in *Proceedings of the Fifth International Symposium on Assessment of Software Tools*, Pittsburgh, PA, pp. 3-13, June 2-5, 1997.

17. A. H. Dogru, S. N. Delcambre, C. Bayrak, Y. T. Chen, E. S. Chan, W. Yin, M. G. Christiansen, and M. M. Tanik, "An Integrated System Design Environment: Concepts and a Status Report," *Journal of Systems Integration,* Volume 2, Number 4, pp. 317-347, October 1992.

18. A. H. Dogru, L. K. Jololian, and M. M. Tanik, "Component-based Technology for the Engineering of Virtual Enterprises and Software," *Technical Report,* TR-98-7, Computer Engineering Department, Middle East Technical University, Turkey, 1998.

19. D. Garlan and M. Shaw, "An Introduction to Software Architecture," *Advances in Software Engineering and Knowledge Engineering*, Vol. 2, World Scientific Publishing, Inc., River Edge, NJ, 1993.

20. D. Garlan, R. Allen, and J. Ockerbloom, "Exploiting Style in Architectural Design Environments," in *Proceedings of SIGSOFT'94*, ACM Press, New York, pp. 179-185, 1994.

21. D. Garlan and D. Perry, "Introduction to the Special Issue on Software Architecture," *IEEE Trans. Software Engineering*, Vol. 21, No. 4, April 1995.

22. D. Garlan, A. Robert, and J. Ockerbloom, "Architectural Mismatch: Why Reuse Is So Hard," *IEEE Software*, Vol. 12, No. 6, pp. 17-26, 1995.

23. A. Gibbons, *Algorithmic Graph Theory*, Cambridge University Press, Cambridge, England, 1985.

24. R. Greenlaw and R. Petreschi, "Cubic Graphs," *ACM Computing Surveys*, Vol. 27, No. 4, pp. 471-495, 1995.

25. J. V. Guttag and J. J. Horning (Eds.), "Larch: Languages and Tools for Formal Specification," *Texts and Monographs in Computer Science*, Springer-Verlag, New York, New York, 1993.

26. F. Hayes-Roth, "Architecture-Based Acquisition and Development of Software: Guidelines and Recommendation from the ARPA Domain Specific Software Architecture (DSSA) Program Version 2," *http://www-internal.sei.cmu.edu/arpa/dssa/dssa-adage*, 1994.

27. M. Jackson, *System Development*, Prentice Hall, Englewood Cliffs, New Jersey, 1983.

28. M. Jazayeri, "Component Programming – a fresh look at software components," in *Proceedings of the 5th European Software Engineering Conference*, Barcelona, Spain, Sept. 25-28, 1995.

29. R. Kazman and L. Bass, "Toward Deriving Software Architectures From Quality Attributes," *Technical Report*, CMU/SEI-94-TR-10, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1994.

30. R. Kazman, G. Abowd, L. Bass, and P. Clements, "Scenario-Based Analysis of Software Architecture," *IEEE Software*, Vol. 13, No. 6, 1996.

31. V. Kirova, L. Jololian, H. Lawson, and T. Zemel, "A Generic Model for Software Architectures," *IEEE Software*, Vol. 14, No. 4, IEEE Computer Society Press, Los Alamitos CA, pp. 84-92, July/August 1997.

32. V. Kirova and W. Rossak, "ASPECT: The Generic Architecture Description Language and its Customization Facilities," in *Proc. of the Workshop on Systems Engineering of Computer Based Systems*, IEEE ECBS TSC, IEEE Computer Society Press, Los Alamitos, CA, pp. 192-199, April 1998.

33. V. Kirova and W. Rossak, "ASPECT - An Architecture SPECification Technique: A Report on Work in Progress," in *Proc. of Symposium and Workshop on Systems Engineering of Computer Based Systems*, IEEE ECBS TSC, IEEE Computer Society Press, Los Alamitos, CA, March 1996.

34. V. Kirova and W. Rossak, "Representing Architectural Designs: A Central Issue in the Development of Complex Systems," in *Proc. of the IEEE First International*

Conference on Engineering of Complex Computer Systems (ICECCS), Ft. Lauderdale, FL, USA, pp. 80-87, November 1995.

35. V. Kirova, W. Rossak, and H. Lawson, "Software Architectures: An Analysis of Characteristics, Structure and Application," *IEEE International Conference on Software Engineering, ICSE-17, Workshop on Architectures for Software Systems*, Seattle, WA, USA, pp. 166-174, April 1995.

36. P. Kogut and P. Clements, "Features of Architecture Description Languages," presented at *Software Technology Conference*, Salt Lake City, Utah, April 1995.

37. D. Krieger and R. M. Adler, "The Emergence of Distributed Component Platforms," *IEEE Computer,* pp. 43-53, March 1998.

38. P. B. Kruchten, "The 4+1 View Model of Architecture," *IEEE Software*, Vol. 12, No. 6, pp. 42-50, 1995.

39. P. Kruchten and C. Thompson, "An Object-Oriented, Distributed Architecture for Large Scale Ada Systems," in *Proceedings of TRI-Ada'94*, Baltimore, Maryland, November 1994.

40. W. Lam and A. Vickers, "Managing the Risks of Component-Based Software Engineering," in *Proceedings of the Fifth International Symposium on Assessment of Software Tools*, Pittsburgh, PA, pp. 123-132, June 2-5, 1997.

41. D. C. Luckham and J. Vera, "An Event-Based Architecture Definition Language," *IEEE Transactions on Software Engineering*, pp. 717-734, September 1995.

42. D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and Analysis of System Architecture Using Rapide," *IEEE Transactions on Software Engineering*, pp. 336-355, April 1995.

43. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying Distributed Software Architectures," in *Proceedings of the fifth European Software Engineering Conference (ESEC'95)*, Barcelona, Spain, September 1995.

44. J. Magee and J. Kramer, "Dynamic Structure in Software Architectures," in *Proceedings of ACM SIGSOFT'96: 4th Symposium on the Foundations of Software Engineering (FSE4)*, San Francisco, CA, pp. 3-14, October 1996.

45. T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, pp. 308-320, 1976.

46. D. McIlroy, "Mass-produced software components, Software Engineering Concepts and Techniques," reprinted *Proceedings of the 1968 and 1969 NATO*

Conferences (Edited by P. Naur, B. Randell, and J. N. Buxton), Petrocelli/Charter, New York, pp.88-98, 1969.

47. N. Medvidovic, "ADLs and Dynamic Architecture Changes," in *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, San Francisco, CA, pp. 24-27, October 1996.

48. N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor, "Using Object-oriented Typing to Support Architectural Design in the C2 style," in *Proceedings of ACM SIGSOFTT'96: 4th Symposium on the Foundations of Software Engineering (FSE4)*, San Francisco, CA, pp. 24-32, October 1996.

49. N. Medvidovic and R. N. Taylor, "A Framework for Classifying and Comparing Architecture Description Languages," *ACM Software Engineering Notes*, Vol. 22, No. 6, pp. 60-76, 1997.

50. N. Medvidovic, R. N. Taylor, and E. J. Whitehead, Jr., "Formal Modeling of Software Architectures at Multiple Levels of Abstraction," in *Proceedings of the California Software Symposium 1996*, Los Angeles, CA, pp. 28-40, April 1996.

51. R. Mili, A. Mili, and R. T. Mittermeir, "Storing and Retrieving Software Components: A Refinement Based System," *IEEE Trans. SE*, Vol. 23, No. 7, pp.445-460, 1997.

52. R. T. Monroe, A. Kompanek, R. Melton, and D. Garlan, "Architectural Styles, Design Patterns, and Objects," *IEEE Software*, pp.43-52, January 1997.

53. M. Moriconi and X. Qian, "Correctness and Composition of Software Architectures," in *Proceedings of the second ACM SIGSOFT Symposium on Foundations of Software Engineering*, New Orleans, Louisiana, December 1994.

54. M. Moriconi, X. Qian and R. A. Riemenschneider, "Correct Architecture Refinement," *IEEE Trans. on Software Engineering*, pp. 356-372, April 1995.

55. S. S. Muchnik and N. D. Jones, *Program Flow Analysis,* Prentice Hall Inc., Englewood Cliffs, New Jersey, 1981.

56. E. Nahouraii(Ed.), *Proceedings of the Fifth International Symposium on Assessment of Software Tools*, IEEE Computer Society, Pittsburgh, PA, June 2-5, 1997.

57. J. Q. Ning, "Component-Based Software Engineering (CBSE)," in *Proceedings of the Fifth International Symposium on Assessment of Software Tools*, Pittsburgh, PA, pp. 34-43, June 2-5, 1997.

58. NIST, *Component-based Software*, ATP Focused Program Competition 97-06, National Institute of Standards and Technology, United States Department of Commerce, February 1997.

59. D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, Vol. 15, No. 12, pp. 1053-1058, 1972.

60. D. E. Perry and A. L. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Engineering Notes*, Vol. 17, No. 4, 1992.

61. R. E. Prather, "Design and Analysis of Hierarchical Software Metrics," *ACM Computing Surveys*, Vol. 27, No. 4, pp. 497-518, 1995.

62. R. E. Prather, "Regular Expressions for Program Computations," *The American Mathematical Monthly*, Vol. 104, No. 2, pp. 120-130, Feb. 1997.

63. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

64. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, Upper Saddle River, New Jersey, 1996.

65. M. Shaw, M. et al., "Abstractions for Software Architecture and Tools to Support Them," *IEEE Transactions on Software Engineering*, pp. 314-335, April 1995.

66. M. Shaw, "Comparing Architectural Design Styles," *IEEE Software*, Vol. 12, No. 6, pp. 27-41, 1995.

67. M. Shaw, R. DeLine, and G. Zelesnik, "Abstractions and Implementations for Architectural Connections," in *Proceedings of the Third International Conference on Configurable Distributed Systems (ICCDS96)*, 1996.

68. D. Soni, R. Nord, and C. Hofmeister, "Software Architecture in Industrial Applications," in *Proceedings of 17th International Conference on Software Engineering*, Seattle, WA, pp. 196-210, April 23-30, 1995.

69. T. Takeshita, "Metrics and Risks of CBSE," in *Proceedings of the Fifth International Symposium on Assessment of Software Tools*, Pittsburgh, PA, pp. 91-93, June 2-5, 1997.

70. Y. Tang, A. H. Dogru, and M. M. Tanik, "Cyclomatic Complexity Based on Cubic Flowgraphs," in *Proceedings of the Third Conference on Integrated Design and Process Technology*, Berlin, Germany, IDPT Vol. 4, pp. 82-85, July 6-9, 1998.

71. M. M. Tanik and E. S. Chan, *The Fundamentals of Computing for Software Engineers*, Van Nostrand Reinhold, New York, New York, 1991.

72. M. M. Tanik and A. Ertas, "Interdisciplinary Design and Process Science: A Discourse on Scientific Method for the Integration Age," *Journal of Integrated Design and Process Science*, Vol. 1, No. 1, pp. 76-94, September 1997.

73. M. M. Tanik, U. Pooch, and S. Yurttas, *Advanced Programming Techniques in TURBO PASCAL*, Wordware Publishing, Inc., Plano, Texas, 1988.

74. A. Terry, R. London, G. Papanagopoulos, and M. Devito, "The ARDEC/Teknowledge Architecture Description Language (ArTek), Version 4.0," *Technical Report*, Teknowledge Federal Systems, Inc. and U. S. Army Armament Research, Development, and Engineering Center, July 1995.

75. W. Tracz, "LILEANNA: A Parameterized Programming Language," in *Proceedings of the Second International Workshop on Software Reuse*, Lucca, Italy, pp. 66-78, March 1993.

76. W. Tracz, "DSSA Frequently Asked Questions," *Software Engineering Notes*, pp. 52-56, April 1994.

77. S. Vestal, "A Cursory Overview and Comparison of Four Architecture Description Languages," *Technical Report*, Honeywell Technology Center, Minneapolis, MN, 1993.

78. S. Vestal, "MetaH Programmer's Manual, Version 1.09," *Technical Report*, Honeywell Technology Center, Minneapolis, MN, April 1996.

79. A. Vickers, "CBSE: Can We Count the Cost?" in *Proceedings of the Fifth International Symposium on Assessment of Software Tools*, Pittsburgh, PA, pp. 95-97, June 2-5, 1997.

80. J. M. Wing, E. Rollins, and A. M. Zaremski, "Thoughts on Larch/ML and a new application for LP," *In the First International Workshop on Larch*, U. Martin and J. M. Wing, Eds., Spring-Verlag, New York, New York, 1993.

81. A. M. Zaremski and J. M. Wing, "Specification Matching of Software Components," *ACM Trans. on Software Engineering and Methodology*, Vol. 6, No. 4, pp.333-369, 1997.