An interface-based testing technique for component-based software systems

Brian Bui
San Jose State University

AN INTERFACE-BASED TESTING TECHNIQUE FOR COMPONENT-BASED SOFTWARE SYSTEMS

A Thesis

Presented to

The Faculty of the Department of Computer Engineering

San Jose State University

In Partial Fulfillment

of the Requirement for the Degree

Master of Science in Computer Engineering

By

Brian Bui

December 2005

APPROVED FOR THE DEPARTMENT OF
COMPUTER ENGINEERING

[illegible] 12/12/05

Dr. Jerry Gao, Thesis Advisor

Simon Shim 12/12/05

Dr. Simon Shim, Thesis Committee Member

Wm A Barrett 12/7/05

Dr. William Barrett, Thesis Committee Member

APPROVED FOR THE UNIVERSITY

[illegible] 12/21/05

ABSTRACT

AN INTERFACE-BASED TESTING TECHNIQUE FOR COMPONENT-BASED SOFTWARE SYSTEMS

By Brian Bui

Since 1990, Component-Based Software Engineering (CBSE) has rapidly been emerging as a trend in industrial software engineering. Although the trend towards CBSE continues to grow, Component-Based Software System (CBSS) testing has remained a neglected research area. This thesis describes a systematic approach to test and maintain a CBSS based on its interface and specifications. Our approach is to identify each component and analyze the dependency and interaction between each component pair. The dependency information obtained is then used to construct a component dependency graph. The benefit of the constructed graph is twofold. It fastens the process of designing the functional and system test suites. Also, this graph facilitates the change impact analysis, which is crucial to maintain a CBSS if changes to any of its components have been made. A case study is then used to demonstrate the efficiency and effectiveness of the test model.

ACKNOWLEDGMENTS

I am greatly indebted to my thesis advisor, Dr. Jerry Gao, for his advice and support throughout my work on this dissertation. His insight about the subject has helped develop my initial interest in the subject and inspired me to turn the subject of component-based software system testing into my dissertation.

I am especially grateful to my committee members, Dr. William Barrett and Dr. Simon Shim. Their suggestions and comments greatly improve my work.

I am grateful to Ms. Parrish for her assistance in the proofreading of this manuscript.

My sincere thanks also go to my parents for their support and encouragement throughout my study at SJSU, and especially during my time of involvement in working on this thesis.

I owe a great deal of thanks to my friends, Chau Le and Thuan Ly, whose suggestions and feedback have greatly benefited my work.

Above all, I would like to thank my wife, Cuc Huynh, who was always there to support and encourage me, especially during times of stress and uncertainties. Without her emotional support and patience during my research and writing, the completion of this dissertation would become impossible.

Table of Contents

List of Figures viii

List of Tables xiii

1. Introduction 1

  1.2 Why Does CBSS Testing Need Attention? 2

  1.3 Scope and Research Objectives 3

  1.4 Related Work 3

  1.5 Thesis Organization 4

2. Overview of Component-Based Software System 6

  2.1. What Is a Component? 6

  2.2. Component-Based Software Engineering 7

  2.3 Major Issues and Challenges of CBSS Testing 9

3. Component-Based Software System Test Model 11

  3.1 Characteristics of the Test Model 11

  3.2 Assumptions 11

  3.3 Basic Definitions of the Test Model 12

  3.4 The Test Model 14

    3.4.1 An Overview of the Test Model 14

      3.4.1.1 Component Interaction Representation 15

      3.4.1.2 Dependency Types 18

    3.4.2 Component Identification 18

    3.4.3 Data-Function Dependency Analysis 19

      3.4.3.1 Data-Function Dependency Analysis 20

      3.4.3.2 Data-Function Dependency Graph 20

    3.4.4 Function-Function Dependency Analysis 22

    3.4.5 Component Dependency Representation 23

    3.4.6 Transitive Closure Algorithm 24

    3.4.7 Change Impact Analysis 29

      3.4.7.1 Component's API Firewall and Change Impact Algorithms 31

      3.4.7.2 Component's CIDL Firewall and Change Impact Algorithms 36

      3.4.7.3 Test Change Firewall 40

4. Case Study—the Simulated Elevator System 48

4.1 Identifying the Simulated Elevator's Components 51
4.2 Dependency Analysis 51
4.2.1 Dependency Associated with CurrentLevel 51
4.2.2 Dependency Associated with DoorStatus 54
4.2.3 Dependency Associated with ElevatorStatus 57
4.2.4 Complete Elevator System Dependency 59
5 Conclusion and Future Directions 65
5.1 Conclusion 65
5.2 Future Directions 66
References 67
Appendix A. Component Interface Description Language 69
A.1 Motive 69
A.2 Component Interface Description Language 70
Appendix B. Supporting Data for the Elevator System 74
Appendix C. The SPARC Emulator Case Study 80
C.1 Overview of the SPARC Emulator 80
C.2 CIDL and CDG of the SPARC Emulator, Version 1 81
C.3 CIDL Descriptions of the SPARC Emulator, Version 2 104
C.3.1 Added and Modified Components 104
C.4 Change Impact Analysis 108

List of Figures

Figure 2.2.1. Component Diagram. 8
Figure 2.2.2. Component Assembly Process. 9
Figure 3.4.1.1. A Service-Provider Component 16
Figure 3.4.1.2. A Service-Requested Component 16
Figure 3.4.1.3. Component Interaction 17
Figure 3.4.2.1. Component Identification and Isolation 19
Figure 3.4.3.2.1. Data-Function Dependency Graph. 22
Figure 3.4.4.1. Direct Function-Function Dependency 23
Figure 3.4.5.1. Function-Function Dependency 24
Figure 3.4.6.1. Transitive Closure Sub-Problem 25
Figure 3.4.6.2. Transitive Closure Matrix 29
Figure 4.0.1. The Simulated Elevator CBSS. 49
Figure 4.2.1.1. Data-Function Dependency Associated with CurrentLevel 52
Figure 4.2.1.2. Function-Function Dependency Associated with CurrentLevel 53
Figure 4.2.1.3. Adjacency Matrix for CurrentLevel 54
Figure 4.2.2.1. Data-Function Dependency Associated With the DoorStatus Variable 55
Figure 4.2.2.2. Data-Function Dependency Associated with The DoorStatus Variable 56
Figure 4.2.2.3. Adjacency Matrix for DoorStatus 57
Figure 4.2.3.1. Data-Function Dependency Associated with ElevatorStatus 57
Figure 4.2.3.2. Function-function Dependency Associated with ElevatorStatus 58
Figure 4.2.3.3. Adjacency Matrix for ElevatorStatus 59
Figure 4.2.4.1. Complete Elevator System Dependency 60
Figure 4.2.4.2. Adjacency Matrix for the Simulated Elevator System 61
Figure 4.2.4.3. Transitive Closure Matrix for the Simulated Elevator System 62
Figure 4.3.1. Changes Made to Version 2. 63
Figure 4.3.2. Change Impact Analysis for the Removal of the MoveTo Node 64
Figure A.2.1. CBSS Described by CIDL 71
Figure A.2.2. A Two-Component CBSS Example. 72
Figure A.2.3. Comp-1's API 72
Figure A.2.4. CII of Comp-1 and Comp-2. 73
Figure B.1. API of the OperationPanel. 74
Figure B.3. CII between the Button2 Function of the OperationPanel Component and the MoveTo Function of the Elevator Component. 75
Figure B.4. CII between the Button3 Function of the OperationPanel Component and the moveTo Function of the Elevator Component. 75
Figure B.5. CII between the OpenDoorButton Function of the OperationPanel Component and the OpenDoor Function of the Elevator Component. 75
Figure B.6. CII between the CloseDoorButton Function of the OperationPanel Component and the CloseDoor Function of the Elevator Component. 76
Figure B.7. CII between the StopButton Function of the OperationPanel Component and the StopElevator Function of the Elevator Component. 76

Figure B.8. CII between the DialButton Function of The OperationPanel Component and the Dial Function of the Elevator Component. 76
Figure B.9. CII between the OperationPanel Function of the OperationPanel Component and the ShowStatus Function of the Elevator Component.......................77
Figure B.10. CII between the moveUp Function of the Elevator Component and the ControlMoveUp Function of the Controller Component. 77
Figure B.11. CII between the MoveDown Function of the Elevator Component and the ControlMoveDown Function of the Controller Component. 77
Figure B.14. CII between the UpButtonL2 Function of the FloorPanel Component and the ControlMoveUp Function of the Controller Component. 78
Figure B.15. CII between the DownButtonL2 Function of the FloorPanel Component and the ControlMoveDown Function of the Controller Component. 79
Figure B.16. CII between the DownButtonL3 Function of the FloorPanel Component and the ControlMoveDown Function of the Controller Component. 79
Figure C.2.1. API of the SparcEmulator (the CUT). 81
Figure C.2.2. CII between the OpenFileButton of the SparcEmulator Component and the ReadSource Function of the LoadFile Component. 81
Figure C.2.3. CII between the ResetButton of the SparcEmulator Component and the SetCurrentStatement Function of the StatementHolder Component. 82
Figure C.2.4. CII between the ResetButton of the SparcEmulator Component and the Reset Function of the SparcData Component. 82
Figure C.2.5. CII between the StepButton Function of the SparcEmulator Component and the AddNorm Function of the Addxx Component. 82
Figure C.2.6. CII between the StepButton Function of the SparcEmulator Component and the Addx Function of the Addxx Component...83
Figure C.2.7. CII between the StepButton Function of the SparcEmulator Component and the Addxcc Function of the Addxx Component. 83
Figure C.2.8. CII between the StepButton Function of the SparcEmulator Component and the AddNormcc Function of the Addxx Component. 83
Figure C.2.9. CII between the StepButton Function of the SparcEmulator Component and the Sub Function of the SubtractX Component. 84
Figure C.2.10. CII between the StepButton Function of the SparcEmulator Component and the Subx Function of the SubtractX Component. 84
Figure C.2.11. CII between the StepButton Function of the SparcEmulator Component and the Subxcc Function of the SubtractX Component. 84
Figure C.2.12. CII between the StepButton Function of the SparcEmulator Component and the Subcc Function of the SubtractX Component. 85
Figure C.2.13. CII between the StepButton Function of the SparcEmulator Component and the Umul Function of the MultX Component...85
Figure C.2.14. CII between the StepButton Function of the SparcEmulator Component and the Umulcc Function of the MultX Component. 85
Figure C.2.15. CII between the StepButton Function of the SparcEmulator Component and the Smul Function of the MultX Component. 86

Figure C.2.16. CII between the StepButton Function of the SparcEmulator Component and the Smulcc Function of the MultX Component. 86
Figure C.2.17. CII between the StepButton Function of the SparcEmulator Component and the DivSign Function of the DivX Component. 86
Figure C.2.18. CII between the StepButton Function of the SparcEmulator Component and the DivUsign Function of the DivX Component. 87
Figure C.2.19. CII between the StepButton Function of the SparcEmulator Component and the DivSigncc Function of the DivX Component. 87
Figure C.2.20. CII between the StepButton Function of the SparcEmulator Component and the DivUsigncc Function of the DivX Component. 87
Figure C.2.21. CII between the StepButton Function of the SparcEmulator Component and the Andnorm Function of the AndX Component. 88
Figure C.2.22. CII between the StepButton Function of the SparcEmulator Component and the AndNormcc Function of the AndX Component. 88
Figure C.2.23. CII between the StepButton Function of the SparcEmulator Component and the AndN Function of the AndX Component. 88
Figure C.2.24. CII between the StepButton Function of the SparcEmulator Component and the ORnorm Function of the AndX Component. 89
Figure C.2.25. CII between the StepButton Function of the SparcEmulator Component and the ORnormcc Function of the ORX Component. 89
Figure C.2.26. CII between the StepButton Function of the SparcEmulator Component and the ORcc Function of the ORX Component. 89
Figure C.2.27. CII between the StepButton Function of the SparcEmulator Component and the ORN Function of the AndX Component. 90
Figure C.2.28. CII between the StepButton Function of the SparcEmulator Component and the ORNcc Function of the AndX Component. 90
Figure C.2.29. CII between the StepButton Function of the SparcEmulator Component and the XOR Function of the XORX Component. 90
Figure C.2.30. CII between the StepButton Function of the SparcEmulator Component and the XORcc Function of the XORX Component. 91
Figure C.2.31. CII between the StepButton Function of the SparcEmulator Component and the XNOR Function of the XNORX Component. 91
Figure C.2.32. CII between the StepButton Function of the SparcEmulator Component and the XNORcc Function of the XNORX Component. 91
Figure C.2.33. CII between the StepButton Function of the SparcEmulator Component and the SLL Function of the ShiftX Component. 92
Figure C.2.34. CII between the StepButton Function of the SparcEmulator Component and the SRL Function of the ShiftX Component. 92
Figure C.2.35. CII between the StepButton Function of the SparcEmulator Component and the SRA Function of the ShiftX Component. 92
Figure C.2.36. CII between the LoadFile Function of the LoadFile Component and the StatementHolder Function of the StatementHolder Component. 94
Figure C.2.37. CII between the LoadFile Function of the LoadFile Component and the StatementHolder Function of the StatementHolder Component. 95

Figure C.2.38. CII between the ReadSource Function of the LoadFile Component and the StatementHolder Function of the StatementHolder Component. 95
Figure C.2.39. CII between the ReadSource Function of the LoadFile Component and the StatementHolder Function of the StatementHolder Component. 95
Figure C.2.40. CII between the ReadSource Function of the LoadFile Component and the StatementHolder Function of the StatementHolder Component. 96
Figure C.2.41. CII between the LoadFile Function of the LoadFile Component and the SetIsLabel Function of the StatementHolder Component....................96
Figure C.2.41. CII between the ReadSource Function of the LoadFile Component and the SetIsComment Function of the StatementHolder Component....................96
Figure C.2.42. CII between the ReadSource Function of the LoadFile Component and the SetIsInvalid Function of the StatementHolder Component. 97
Figure C.2.44. CII between the SetBreakpointMouseClick of the SparcEmulator Component and the SetBreakPoint Function of the Comp-StatementHolder Component....................97
Figure C.2.45. CII between the Addcc Function of the Addxx Component and the SetRegisterValue Function of the Comp-SparcData Component....................98
Figure C.2.46. CII between the Addcc Function of the Addxx Component and the SetRegisterValue Function of the Comp-SparcData Component....................98
Figure C.2.47. CII between the StepButton of the SparcEmulator Component and the BA Function of the Comp-BranchX Component....................100
Figure C.2.48. CII between the RunButton of the SparcEmulator Component and the BA Function of the Comp-BranchX Component....................100
Figure C.2.49. CII between the RunButton of the SparcEmulator Component and the BN Function of the Comp-BranchX Component....................100
Figure C.2.49. CII between the StepButton of the SparcEmulator Component and the BN Function of the Comp-BranchX Component....................101
Figure C.2.50. CII between the SetRegisterValue Function of the SparcData Component and the SetValue Function of the Comp-RegisterHolder Component. ..102
Figure C.2.51. CII between the GetRegisterValue Function of the SparcData Component and the GetValue Function of the Comp-RegisterHolder Component...102
Figure C.2.52. Partial CDG of the Sparc Emulator Showing the Execution Path When the StepButton Is Pressed. 103
Figure C.2.53. Partial CDG of the CBSS Showing the Execution Path of the openFilebutton 103
Figure C.3.1. CII between the StepButton Function of the SparcEmulator Component and the TST Function of the Comp-TestX Component. 104
Figure C.3.2. CII between the RunButton Function of the SparcEmulator Component and the TST Function of the Comp-TestX Component. 105
Figure C.3.3. CII between the TestReg Function of the TestX Component and the GetRegisterValue Function of the SparcData component. 106
Figure C.3.4. CII between the TestReg Function of the TestX Component and the SetRegisterValue Function of the SparcData Component. 106

Figure C.3.5. CII between the ReadSource Function of the LoadFile Component and the SetIsInvalid Function of the StatementHolder Component. 107
Figure C.3.5. CII between the ReadSource Function of the LoadFile Component and the SetIsInvalid Function of the StatementHolder Component. 107
Figure C.4.1. Impact on Part of the System CDG due to the Added IsInstruction Function 108
Figure C.4.2. Impact on Part of the System CDG due to the Modified Addcc Function... 109

List of Tables

Table 3.4.1.1. Test Model Algorithm.. 15
Table 3.4.3.1. Data-Function Dependency Types... 20
Table 3.4.6.1. Floyd-Warshall Algorithm.. 27
Table 3.4.6.2. Transitive Closure Algorithm.. 28
Table 3.4.7.0.1. Possible Changes Made to a Component...................................... 30
Table 3.4.7.0.2. Test Model Change.. 31
Table 3.4.7.1.1. Change Impact Algorithm for Modified API Functions..................... 33
Table 3.4.7.1.2. Change Impact Algorithm for Deleted API Functions...................... 35
Table 3.4.7.1.3. Change Impact Algorithm for Added API Functions........................ 36
Table 3.4.7.2.1. Change Impact Algorithm for Any Function Modified from Any Component's CII of a CIDL-Based CBSS... 38
Table 3.4.7.2.2. Change Impact Algorithm for Any Function Deleted from Any component's CII of a CIDL-Based CBSS... 39
Table 3.4.7.2.3. Change Impact Algorithm for Any Function Deleted from Any component's CII of a CIDL-Based CBSS... 40
Table 3.6.2.1. Algorithm to Find a Path from One Node to Another......................... 46
Table 3.6.2.2. Path between All-Node-Pair Algorithm....................................... 47
Table C.2.1. Direct Dependency between the SparEmulator.runButton and Other Components in the SPARC Emulator System, Version 1............................. 93
Table C.2.2. Dependency of the StepButton or RunButton on the Functions of the StatementHolder Component... 94
Table C.2.3. Dependency between a Mnemonic Function and the Functions of the SparcData Component.. 99
Table C.2.4. The Branch Component..101
Table C.3.1. Components Added to Version 2...104
Table C.3.2. Dependency between the StepButton/RunButton of the SparcEmulator Component and the Functions of the New Components..............................105
Table C.3.3. Dependency between the Added Components and the Old Components....106

1. Introduction

In this chapter, we describe the basic concepts of Component-Based Software Engineering (CBSE), point out the importance of Component-Based Software System (CBSS) testing, and discuss the reasons why a testing model is needed. The chapter is concluded with a brief discussion of the organization of this paper.

1.1 Background

Early in the 1990's, under the pressure to meet the marketplace need, software researchers and practitioners began the search for an alternative way to build efficient and cost-effective software systems. Their work and research gave birth to a very attractive, fast, efficient, and cost-effective technique, known as CBSE, to build new software systems. The idea is to use high-quality, reusable modules called components as the building blocks for constructing large and complex software systems.

As software systems become larger and more complex, the abandon of the traditional software development approach makes sense since it often requires building a software system from scratch. No doubt, that approach is too costly and far too often fails to meet the demand of today's fast-changed market and technology.

Saving time and money are two critical aspects of developing software systems. According to Voas (1998), even the best programmer can only produce 10 lines of "validated, documented code per day." Also, building software systems using the traditional approach proves to be "often too late—too late to be productive before becoming obsolete" (Szyperski et al., 2002). Given these facts, the cost of developing new software systems, containing hundreds of thousands of lines of code, will be far too expensive to be marketable in the near future. Furthermore, the time it takes to drive a product from inception to delivery will soon mean a battle of life and death.

The widespread and rapid acceptance of CBSE has indicated the importance of an emerging technology that can be used to rapidly build CBSS's to meet today's fast-changed market. The trend towards CBSE was once predicted that "at least 70% of all new software applications to be developed in 2003 will be assembled primarily from components" (Goulao & Abreu, 2002). Although the tendency to use components to build software systems has been increasingly growing, most of the work in CBSE has focused on the "technical and technological issues" (Cechich et al., 2003). CBSS testing, however, has received very little attention (Gao et al., 2003).

1.2 Why Does CBSS Testing Need Attention?

Components' reliability plays an extremely important role in the development of any CBSS. This implies that reliability should take into account not only the reliability of any individual component, but all the components in the system as a whole. If any component in a CBSS fails to function, it can certainly lead to disaster. One example is the case of the Ariane 5 vehicle that exploded only one minute after takeoff (Weyuker, 1998; Xie, 2004). The problem was the inadequate testing of the components reused from its predecessor—the Ariane 4.

Today's trend towards CBSD indicates the need to develop an efficient and effective model to test CBSS's to avoid the disaster similar to the Ariane 5 problem. And this is the very objective of this paper, which will be discussed in the next section.

1.3 Scope and Research Objectives

The goal of this research is to develop a systematic approach to test a CBSS based on its interface and specifications. The test model focuses on the component's interface, also known as Application Programming Interface (API), and its specifications to deduce the dependency and interaction among the components of a CBSS. The process of constructing the test model begins with the static analysis of individual component's interface. The interaction between every component pair, through the publicly accessible methods from the interfaces, is then analyzed. Complete coverage is achieved after all the dependency among the components in that CBSS have been examined. The direct dependency information between one component and another is then stored in a table, called the adjacency matrix. Once this table is constructed, we apply an algorithm to find all the indirect dependency between one component and the others. Finally, a Component Dependency Graph (CDG) is constructed to help analyze the change impacts resulting from possible future modifications of any component. Another algorithm is then applied on the CDG to examine the ripple effects that might occur if changes to any component in that CBSS would be made.

1.4 Related Work

Orso et al. (1999) propose incorporating metadata into a component that provides both dynamic and static information about that component. Component users can then use the information provided by the metadata to test and analyze a CBSS. Harrold et al. (2001) then extends the idea so that the metadata provided in a component can be used to support regression testing based on the component's source code or specifications.

Gao et al. (2003) employ the collaboration diagram, state-chart diagram, and component diagram provided by the Unified Modeling Language (UML) as a tool for CBSS integration testing. Similarly, Strembeck & Zdun (2005) suggest a scenario-based approach (similar to use cases in UML) to develop test cases against a component's functionality.

Beydeda & Gruhn (2001) propose constructing a component-based software flow graph derived from the component's specification (black-box) and source code (white-box) for creating test cases. The study of both the data and function dependency between the internal and external functions are examined to generate test cases.

Haddox et al. (2002) suggests the use of a wrapper that sits between a component and the system on which that component runs. This extra layer helps intercept all the input and output from that component (or a CBSS). Information obtained from the interception makes it possible to apply fault injection, internal data state gathering, and assertion checking as the appropriate techniques to test a CBSS.

1.5 Thesis Organization

The remainder is organized into several chapters and appendices. Chapter 2 gives an overview of CBSE and the issues and challenges of CBSS testing. Chapter 3 covers the component test model, which is the research objective of this paper. Chapter 4 presents a case study to show how the test model can be applied on a real CBSS for testing and maintaining purposes. Chapter 5 concludes the paper and gives recommendations for future direction. Appendix A focuses on the description of the Component Interface Description Language (CIDL), which can be used to facilitate the task of CBSS testing.

Appendix B includes the CIDL representation for the case study presented in Chapter 4.

Appendix C goes over a relatively large, real-world project as another case study.

2. Overview of Component-Based Software System

In this chapter we discuss a CBSS, the characteristics of each component, and the interaction of these components in a CBSS. Also, we go over the issues and challenges in CBSS testing. We conclude the chapter by presenting the techniques that component software researchers, over a decade, have proposed to test a CBSS.

2.1. What Is a Component?

In the software component literature, a "component" is broadly and loosely defined. Szyperski's definition would be one of the most widely accepted definitions of a component. According to Szyperski (1996), "a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

Szyperski's definition tends to emphasize 3 important characteristics of a component: an independent unit, a unit of composition, and a well-defined interface. This means a component, as an independent unit, can have its own interface. And the composition or integration of several components to form a CBSS may have another interface—the interface of the composition as a whole.

Similarly, Crnkovic and Larsson (2002) define a component as "a reusable unit of deployment and composition that is accessed through an interface." This definition stresses three important characteristics of a component: reusable, composition, and interface. A component as a reusable unit may mean that it can be used "as is" at the time needed, and it can later be modified for any particular purposes. The other two characteristics—interface and composition—are defined in a similar fashion to those of Szyperski's definition.

Likewise, D'Souza & Wills (1998) defines a component as "... a coherent package of software that can be independently developed and delivered as a unit, and that offers interfaces by which it can be connected, unchanged, with other components to compose a larger system."

As we have seen, researchers have attempted to define a component in many different ways. Despite those differences, all definitions clearly share one common point—the emphasis of the importance of well-defined interfaces. In fact, a component's interface is very important since it is "a specification of its access point" (Crnkovic & Larsson, 2002).

2.2. Component-Based Software Engineering

Since 1990, CBSE has rapidly been emerging as a trend in industrial software engineering. CBSE can be described as a paradigm shift from the traditional approach to build software systems that can meet the demand of a fast-change market and technology in a timely manner. Loosely speaking, CBSE is the process of assembling preexisting software components, often Commercial-Off-The-Shelf (COTS) components, to build new systems rather than implementing them from scratch.

Components are available in two forms: in-house and COTS (Goulad & Brito, 2002). They can be written in a wide variety of programming languages and run on different platforms. Today, system designers are increasingly becoming interested in using COTS components to build new software systems. This approach has been widely adopted, and Mehta et al. (2002) even suggest rebuilding legacy systems using CBSE. Also, Voas

(1998) observes that components have been deployed in many areas such as mobile phones, radios, bridges, etc.

Figure 2.2.1 is a pictorial representation of a component. A component may have a number of interfaces, and each interface represents a service access point. A service access point can either be a provided service or a requested service. The function signatures (i.e., return type and formal arguments) associated with each service specifies how its functionality can be accessed. Well-written specifications usually point out the pre-conditions and post-conditions required before and after an API function is accessed and returned, respectively.

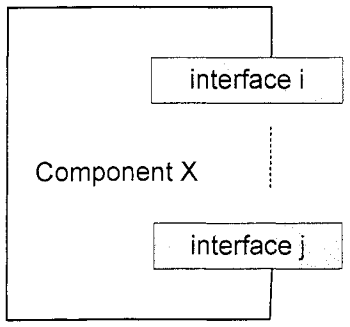

Figure 2.2.1. Component Diagram.

Figure 2.2.2 illustrates the process of assembling a CBSS. Software system designers begin the process by selecting the desired COTS or in-house components. Each component is then tested to make sure that it behaves according to its specification. Once unit testing is done, integration testing usually follows to make sure that the selected components interact with one another well. Finally, all the components are assembled to form a functional CBSS.

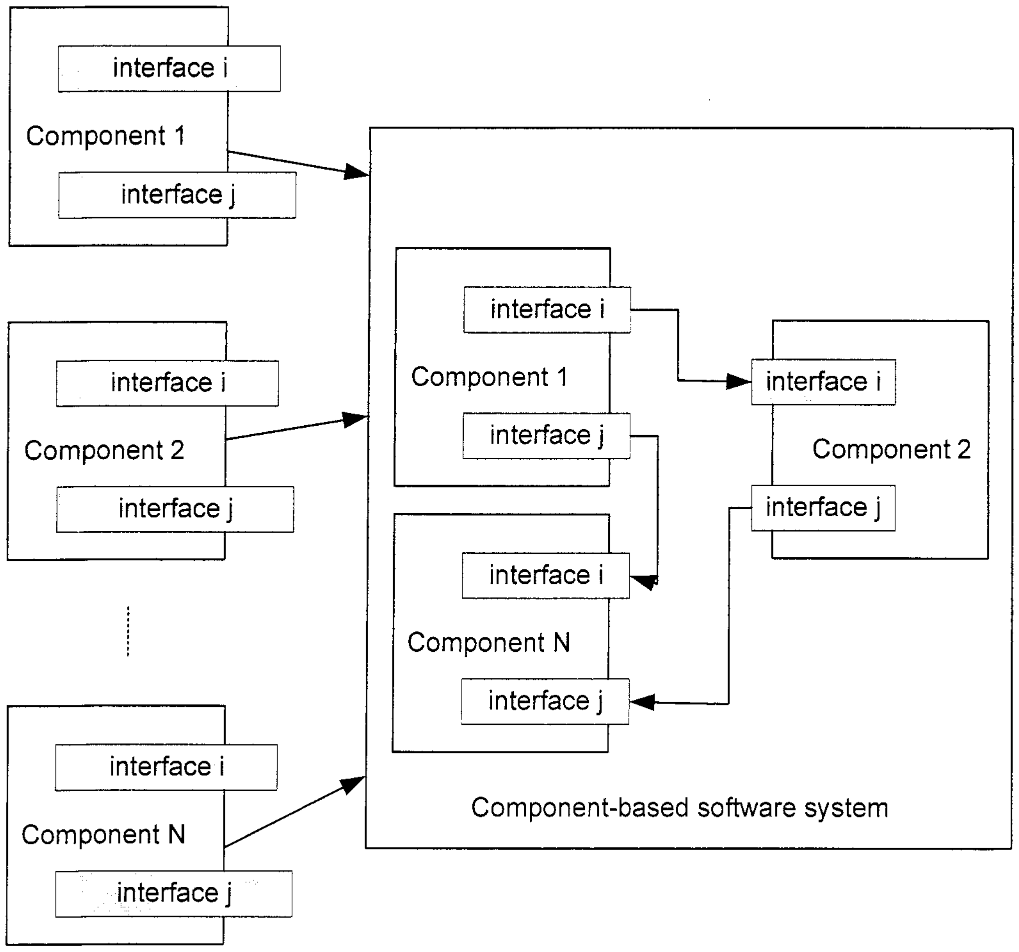

Figure 2.2.2. Component Assembly Process.

2.3 Major Issues and Challenges of CBSS Testing

Software system maintenance is a very expensive process. Testing can account for more than half the total cost of a software system (Harrold, 2000). The cost for maintaining a CBSS is even higher (Ye & Offutt, 2001). Unfortunately, CBSS testing has remained a neglected research area.

One of the most obvious challenges of CBSS testing, as cited by Rosenblum (1997), is the "technological heterogeneity" nature of the prefabricated components. Technological heterogeneity means that different components used to assemble a CBSS may be written in different programming languages and targeted different platforms. Thus, the testing methodology used must be both platform independent and language independent.

Other major issues and challenges of CBSS testing, according to Gao et al. (2003), include the following

- COTS components are shipped as binaries. This makes it very difficult to analyze and test components because of the lack of access to the source code.
- Reusable components must be retested in the new contexts to avoid the Ariane 5 problem mentioned earlier.
- It's costly to construct test beds and test drivers.
- COTS components provided by component vendors are not built-in testability components.

3. Component-Based Software System Test Model

In this chapter, we detail the steps necessary for constructing the CBSS test model. Assumptions about a CBSS are made, and definitions of each part of the test model are given. Then, we explain how the test model can be used to design the necessary test cases for the functional and system testing purposes. We conclude the chapter with the explanation of how the test model can be used to maintain a CBSS.

3.1 Characteristics of the Test Model

The component test model represents a high-level abstraction of a CBSS, and it has the following characteristics

- The test model is both platform- and language-independent, so it can be used to represent a wide variety of CBSS's.
- It is scalable.
- It is constructed from the component's API and functional specifications. Ideally, component dependency is described using the CIDL, which is described in Appendix A.
- Dependency among components is presented using a CDG that helps ease the maintenance task.

3.2 Assumptions

We make the following assumptions about all the components of a CBSS under study

- Each component should have well defined interfaces and functional specifications.

- If any dependency exists between a service of one component and another, this information should be clearly and explicitly stated in either the component's API or its functional specifications.
- Ideally, a CBSS is specified by a simple scripting language called the Component Interface Description Language (CIDL), which has been proposed and developed by Dr. Jerry Gao at SJSU. We postpone the discussion of CIDL and revisit this topic in Appendix A. For now, it suffices to say that CIDL offers many advantages over the current component technology since it helps increase component testability and controllability.

3.3 Basic Definitions of the Test Model

This section is intended to go over the basic definitions of the test model only. Definitions specific to any part of the test model will be discussed in a separate and relevant section for ease of reference. Let's begin with the definition of a component.

Definition 3.3.1. A component, $C = API_c \cup FS_c \cup B_C$, is a set of API functions and functional specifications where

- $API_c = \{API_{F1}, API_{F2},\ldots, API_{Fn}\}$ is the functional API of the component C. Each API_{Fi} should, at least, clearly define every externally accessible function Fi.
- FS_c is a well-defined functional specification of the component C, and
- B_C is the component binary.

Several components may be required to form a functional CBSS. In such a system, the interaction among the components is an important concept since the realization of component dependency would both enhance the understanding of a CBSS and facilitate the process of change impact analysis. And change impact analysis is extremely important to evaluate the effects of changes made to a CBSS. The discussion thus far leads to Definition 3.3.2.

Definition 3.3.2. An event, E = $\{E_1, E_2\ E_i, E_{i+1}, \ldots, En\}$, is a finite set of events that may happen during the execution of a CBSS where

- Each event E_i denotes a distinct link that represents the interaction between 2 functions of a component or a function of one component to that of the others.

Each API function of a component in a CBSS should be thoroughly tested. A test suite is specifically designed for each API function of a component to ensure its functionality. The component's unit-testing test suite is defined in Definition 3.3.3.

Definition 3.3.3. A CBSS test suite, $T = \{t_1, t_2, \ldots, t_n\}$, is a test suite that consists of all possible test cases intended to execute against all the API and Component Interaction Interface (CII) functions , $F_C = \{F_1, F_2, \ldots., F_n\}$, of that CBSS. Each t_i is a test set that is used to execute against a particular API or CII function F_i, each representing a node on the CDG.

The test model is graph-based, and this graph is used to represent the direct dependency among the functions of every component pair. The data structure used to store the dependency information is a matrix (i.e., a 2 x 2 array). It is called the adjacency matrix and is defined in Definition 3.3.4.

Definition 3.3.4. Adjacency Matrix. An adjacency matrix, $A = a_{ij}$, is a Boolean matrix whose i^{th} row and j^{th} column is 1 if and only if there occurs a directed edge from i to j.

3.4 The Test Model

This section gives detailed descriptions of the test model. Section 3.4.1 gives an overview of the test model. Section 3.4.2 shows how the interaction among the functions of the components would be represented. Section 3.4.3 covers the component identification process. Section 3.4.4 includes the steps necessary to identify data-function dependency. The process of identifying the dependency between one function of a component and the function(s) of another component is discussed in Section 3.4.5. Section 3.4.6 covers the representation of data-function and function-function dependency. Section 3.4.7 concludes the chapter with the explanation of how the test model can be used to maintain a CBSS if changes to any component in a CBSS would be made.

3.4.1 An Overview of the Test Model

The process of constructing the test model consists of 7 steps. We begin the process by identifying every component of a CBSS. The dependency between one component and another is then examined and presented using a dependency matrix, called the adjacency matrix. The adjacency matrix only records all the direct dependency among the functions of the components. Another algorithm is used to find all the direct and indirect dependency whose information is later stored in another matrix, called the transitive-closure matrix. Information obtained from the transitive-closure matrix can later be used to construct the CDG. Test coverage algorithms as well as change impact algorithms are then developed to traverse the CDG for the purpose of testing and maintaining a CBSS. Table 3.4.1.1 summarizes the algorithm used to construct the test model.

Test Model Algorithm

1. Identify each component of a CBSS.
2. Analyze the dependency among the functions of one component and those of the others.
3. Construct an adjacency matrix to record all the direct dependency among the functions of the components.
4. Construct the transitive-closure matrix to record all direct an indirect dependency among the components.
5. Construct the component dependency graph for maintenance purposes.
6. Develop the algorithms required for test coverage criteria.
7. Develop the algorithms necessary to analyze change impacts.

Table 3.4.1.1. Test Model Algorithm

3.4.1.1 Component Interaction Representation

Components can be divided into two categories: service provider and service requestor. A service-provided component is designed to serve a request of other components. A service-requested component sends a request to a service provider for a particular service. There are also cases in which a component can serve as both a service-provided and service-requested component. In any case, a component's interface is the access point for inter-component interactions. Figure 3.4.1.1 is a pictorial representation of a service-provided component. For simplicity, this example shows a component with only 2 services, both of which are provided services. In reality, however, keep in mind that a component may offer many different services, and it may have both provided and requested services.

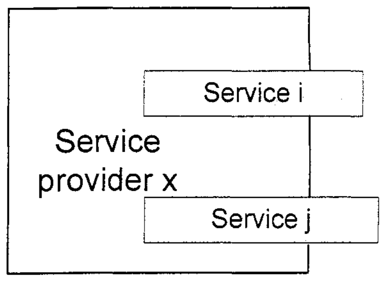

Figure 3.4.1.1. A Service-Provider Component

Similarly, Figure 3.4.1.2 represents a service-requested component. Again, the purpose here is, for simplicity, to show a component with only two requested services: Request i and Request j.

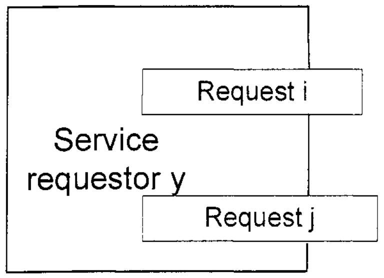

Figure 3.4.1.2. A Service-Requested Component

Components in a CBSS interact with one another via their interfaces. Depending on the complexity of a CBSS, the number of components can be very large, and the dependency among these components can be very complex. But, no matter how complex a CBSS might have been imagined, the interaction among the components of a CBSS follows the same principle. That is, these components provide or request a service through their interfaces. As long as the requested function of one component passes the correct arguments to the provided service of another, the requested service is guaranteed to be fulfilled (of course with the assumption that no system fault or error has been occurred during this time period). For simplicity, the interaction between 2 simple components in a typical CBSS is shown in Figure 3.4.1.3.

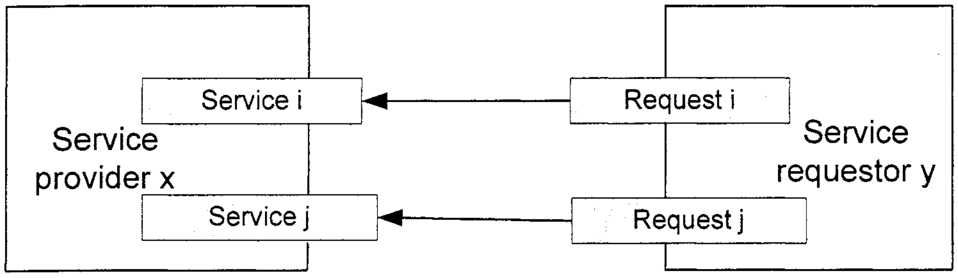

Figure 3.4.1.3. Component Interaction

In our test model, the interaction among all the components in a CBSS is represented as a CDG. From that graph, system testers can easily visually view the dependency among the components. The concept of the CDG is formally defined in Definition 3.4.1.1.

Definition 3.4.1.1. Component Dependency Graph. A component dependency graph, denoted $G_C = (V, E, D)$, is a directed graph where

- $V = \{v_1, v_2, \ldots, v_n\}$ is a finite set of nodes, and each node $v_i \in V$ represents a function of a component.
- $E = V \times V$ is a finite set of edges. Each edge $e(u,v) \in E$ is a link from the source node u to the destination node v.
- D is a finite set of edged label, which either represents data-function dependency or function-function dependency.

3.4.1.2 Dependency Types

Component dependency can be classified into 3 categories: data-function dependency, function-function dependency, and control sequence dependency. Each dependency type will be discussed in a separate section shortly.

3.4.2 Component Identification

Given the facts that a CBSS can be very large and complex, the task of validating and revalidating such a system would be very complicated (in fact, it might be very sophisticated and tedious) if one has no knowledge of the dependency among its constituent components. The component identification process is very important for the following reasons

- One component may have a number of features. In this case, unit testing is mandatory to ensure the quality and reliability of each component before conducting integration and system testing.
- It helps component users and testers gain a better understanding of the CBSS under study.
- It greatly benefits the process of change impact analysis and regression testing when changes are made to any component of a CBSS.

From now on, the CBSS example shown in Figure 3.4.2.1 will be used to discuss the process of constructing the test model. As illustrated in that Figure, the CBSS example is composed of 3 components: component X, component Y, and component Z. Each is identified and isolated for the purpose of data-function dependency analysis describing in the next section.

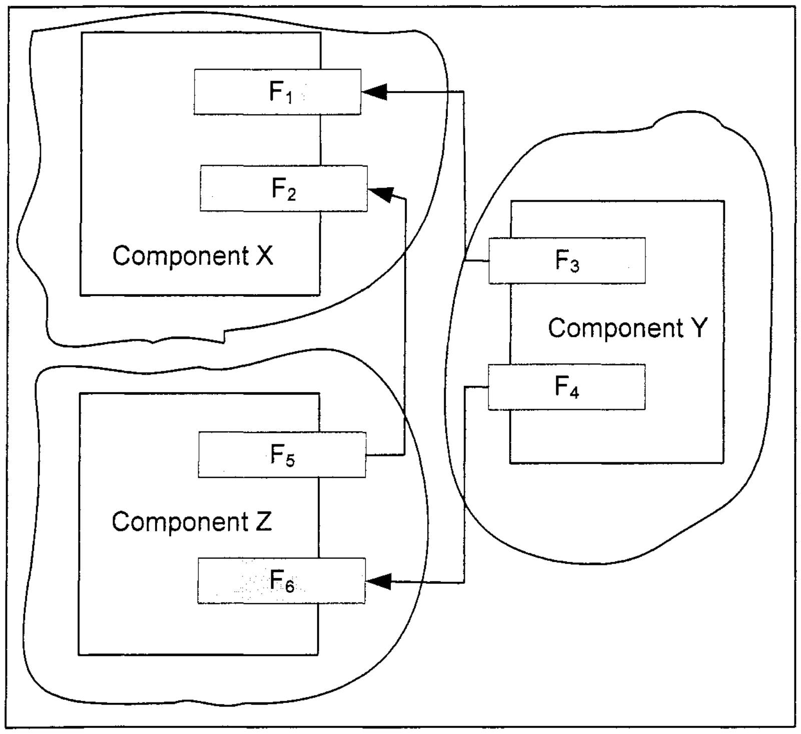

Figure 3.4.2.1. Component Identification and Isolation

3.4.3 Data-Function Dependency Analysis

Data-function dependency can be classified into 5 categories as follows

- Defined→Used: one function (may be more than one) Fi defines the datum Di, and another function Fj (may be more than one) uses Di.
- Defined→defined and used: one function (may be more than one) Fi defines the datum Di, and another function Fj (may be more than one) uses Di and redefines it.
- Defined and used→Defined and used: one function (may be more than one) Fi defines and uses the datum Di, and another function Fj (may be more than one) also uses Di and redefines it.

Table 3.4.3.1 is designed to review the key points of the data-function dependency types.

Data Dependency Types	
Function Fi	Function Fj
Fi defines Di only	Fj uses Di only
Fi defines Di only	Fj both defines and uses Di
Fi both defines and uses Di	Fj uses Di only
Fi both defines and uses Di	Fj both defines and uses Di

Table 3.4.3.1. Data-Function Dependency Types

3.4.3.1 Data-Function Dependency Analysis

Information about the Data-function dependency is extracted from the component's API and functional specifications. The process begins with the construction of a data-function dependency graph. This graph shows all the functions of a component that either define or use a particular datum D_i directly. Information from this graph can be used to build an adjacency matrix that conveys the same information as the data-function dependency graph; however, we will later discuss how the adjacency matrix can be used to construct another matrix that shows all the indirect dependency related to the datum D_i.

3.4.3.2 Data-Function Dependency Graph

Before discussing how we represent the data-function dependency, let's first go through a definition that serves as one of the fundamentals of the test model.

Definition 3.4.3.2.1 Data-Function Dependency Graph. A data-function dependency graph G, denoted $G = (V_F, V_D, E, D)$, is a graph in which

- $V_F = \{v_1, v_2, \ldots, v_n\}$ is a finite set of nodes. Each node $v \in V$ represents an API or CII function of a component that either defines or uses the datum D_i. In this context, circles are used to represent a function as a node in any graph.
- V_D is a node that represents a global datum D_i. In this context, square boxes are used to represent the datum D_i in any graph.
- Each edge $e \in E$ represents the relationship between an API or CII function F_i and the datum D_i.
- D specifies the data-function dependency type that indicates whether F_i uses or defines the datum D_i.

A data-function dependency graph is constructed by gathering information from a component's API and CII, and the corresponding functional specifications. Given well-written API and specifications, we can check the entire CBSS to see what functions would define or use the datum D_i. The data-function dependency extracted is then used to construct a data-function dependency for each global datum (if provided in the API or specifications). Figure 3.4.3.2.1 shows a typical data-function dependency graph. This graph indicates the following:

- F_1 (CompX) defines D_i only.
- F_2 (CompX) both defines and uses D_i
- F_3 and F_4 (CompY) use D_i only.
- F_5 and F_6 (CompZ) use D_i only

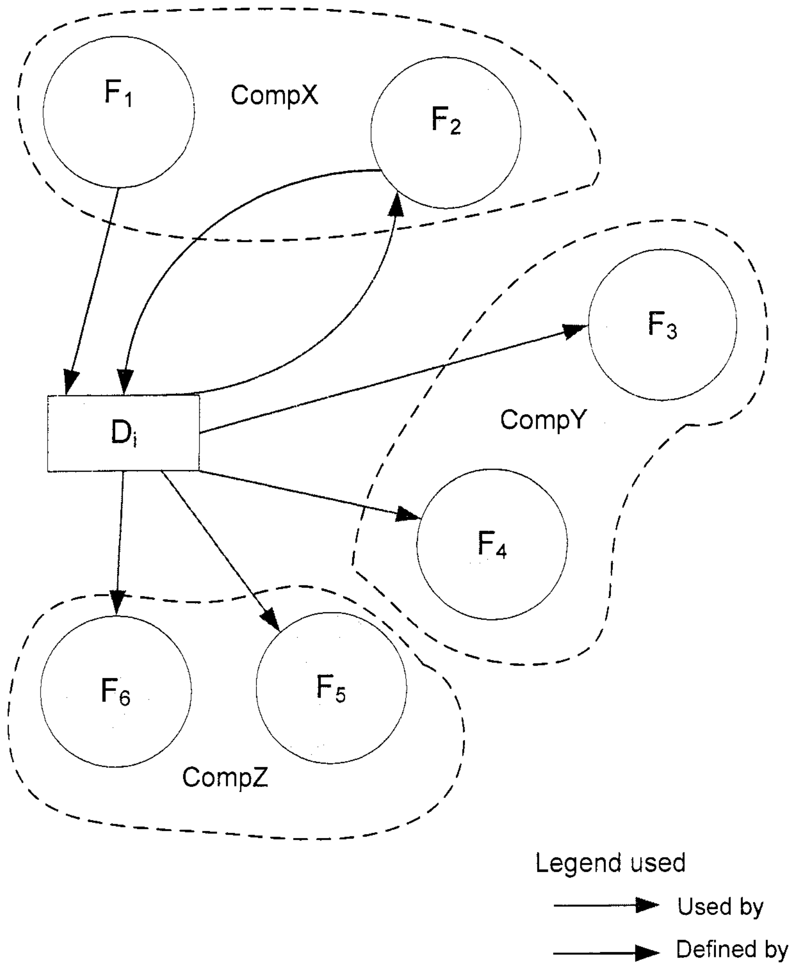

Figure 3.4.3.2.1. Data-Function Dependency Graph.

3.4.4 Function-Function Dependency Analysis

We begin this section with the definition of a function-function dependency graph.

Definition 3.4.3.2.1 Function-Function Dependency Graph. A function-function dependency graph G, denoted G = (V, E, D), is a graph in which

- Each node $v \in V$ represents an API or CII function of a component.
- Each edge $e \in E$ represents the relationship between an API or CII function F_i and another API or CII function Fj.

- D indicates that the dependency type is function-function rather than data-function.

From Figure 3.4.2.1, we can identify the following function-function dependency among the components

1) Dependency between component 1 and component 2: $F_1 \rightarrow F_3$
2) Dependency between component 1 and component 3: $F_2 \rightarrow F_5$
3) Dependency between component 2 and component 3: $F_4 \rightarrow F_6$

The information obtained so far can now be used to construct an adjacency matrix that records all the direct function-function dependency as shown in Figure 3.4.4.1.

	F1	F2	F3	F4	F5	F6
F1	0	1	1	0	0	0
F2	0	0	0	0	1	0
F3	0	0	0	1	0	0
F4	0	0	0	0	0	1
F5	0	0	0	0	0	0
F6	0	0	0	0	0	0

Figure 3.4.4.1. Direct Function-Function Dependency

3.4.5 Component Dependency Representation

Suppose the function-function dependency between the 3 components shown in Figure 3.4.2.1 is further complicated as illustrated in Figure 3.4.5.1. The indirect dependency between any function pair can not be identified by examining the adjacency matrix shown in Figure 3.4.4.1. We need to find a way to represent both the direct and indirect dependency among the components. This is the topic of the next section in which we will present a simple but very effective algorithm that helps reveal all the dependency among the components of a CBSS.

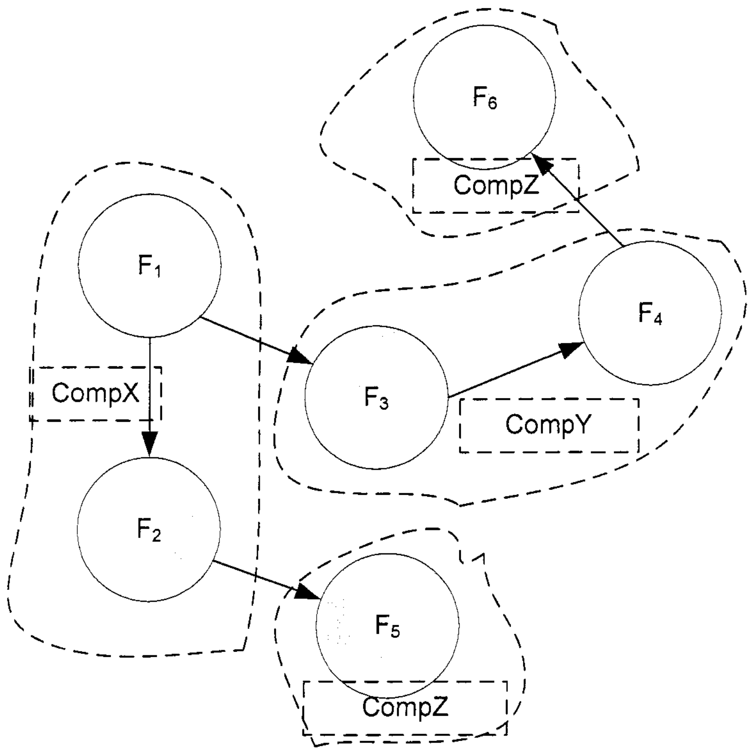

Figure 3.4.5.1. Function-Function Dependency

3.4.6 Transitive Closure Algorithm

An obvious disadvantage of an adjacency matrix is that it only shows the direct dependency between two functions F_i and F_j. Any indirect dependency between F_i and F_j (if occurs) would remain unknown. Therefore, we should somehow come up with an algorithm to detect any indirect dependency between the two functions F_i and F_j. Given a graph G = (V, E) represented by a Boolean N x N matrix A (i.e., an adjacency matrix), the problem now becomes the question of "how would we determine whether there is a path from i to j for all vertexes i, j ∈ V?"

The problem we are trying to solve is to find all paths from node i to node j that may or may not pass through any intermediate nodes. Careful thought reveals the fact that there are only two cases that make it possible to travel from i to j without having to visit any intermediate nodes. The first case is the path that directly connects i and j. The second path is the self loop path. In other words, this is a path from i to j in which i equals to j.

All paths from i to j other than the 2 cases mentioned earlier must pass through some intermediate node. Now, let's discuss this sub-problem based on the simple graph shown in Figure 3.4.6.1. Suppose we have a Boolean N x N matrix A. Suppose further that we are interested in knowing the results of the AND operation being applied on A[i][k] and A[k][j]. The result of that computation is true (i.e., equals to 1) if and only if both the values of A[i][k] and A[k][j] are true. This observation implies two facts. First, there must exist some link E_1 that connects i and k. Likewise, k and j must be connected by some link E_2. The second fact can be deduced from the first by observing that the length of the path from i to j is 2 (i.e., the two links E_1 and E_2).

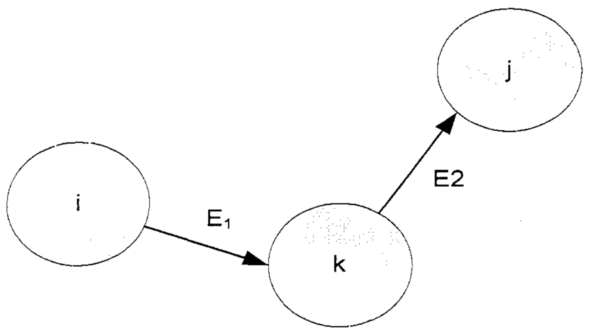

Figure 3.4.6.1. Transitive Closure Sub-Problem

The observation discussed in the previous paragraph can be further supported by the calculation of the Boolean product of an adjacency matrix. Let's take the graph shown in Figure 3.4.5.1 as an example to illustrate this idea, and let's call it graph G. All paths of length 1 have already been presented in the adjacency matrix shown in Figure 3.4.4.1, and let's name this matrix as A_1. Now, suppose we want to find all paths of length 2 by examining A_1 without referring to graph G at all. How can we do that?

The answer to the question turns out to be very simple. Let's try to compute the entry for row F_1 and column F_5 by finding their Boolean product as follows

$$(0\ 1\ 1\ 0\ 0\ 0) \text{ X } \boxed{\begin{matrix}0\\1\\0\\0\\0\\0\end{matrix}} = 0*0 + 1*1 + 1*0 + 0*0 + 0*0 + 0*0 = 1$$

Note that the Boolean product just obtained clearly indicates that a path of length 2 exists between the 2 nodes F_1 and F_5. If we continue to calculate the Boolean product for all other entries in matrix A, the end result is another matrix whose truth value of the entry on row i and column j, if equals to 1, would indicate that there is a path of length 2 from i to j. In other words, the resulting matrix is one that contains all paths of length 2 from node i to node j. Let's denote this matrix as A_2. Similarly, a matrix consisting of all paths of length 3 can be obtained by forming the Boolean product of the entries in the adjacency matrices A_1 and A_2.

Now we can formally define the transitive closure that shows whether there is a path between any pair of nodes F_i and F_j.

Definition 3.4.6.1. Transitive Closure Matrix. Given a directed graph, G = (V, E), where V is a finite set of vertices, and E $\subseteq$ V x V is a finite set of edges. A directed graph, G' = (V, E'), is the transitive closure of G if and only if for any given pair of edges (a, b) and (b, c), there exists a path (a, c).

Now that the transitive closure of graph G has been defined, we are interested in seeking the answer to the question "how can we compute the transitive closure of G?" One way to compute it is to set 1 as the weight for all edges $e_i \in$ E, and then run the well-known Floyd-Warshall Algorithm, which is derived from the following observation

- Let $d_{i,j}$ be the weight of the shortest path from node i to node j whose intermediate nodes belong to the set {1, 2, 3, …, k}.
- When k equals to 0, $d_{i,j}$ must be equal to $a_{i,j}$ since there are no intermediate nodes from i to j; otherwise, $c_{i,j}{}^{(k)} = \min\{c_{i,j}(k-1), c_{i,k}(k-1) + c_{k,j}(k-1)\}$.

Table 3.4.6.1 shows the Floyd-Warshall Algorithm that can be used to compute the all-pairs shortest path from a weighted and directed graph.

```
Floyd-Warshall_Algo(A)
C = A;
n = A.length();
for k = 1 to n do
  for i = 1 to n do
    for j = 1 to n do
      c[i][j] = min { c[i][j], c[i][k] + c[k][j]}
return C.
```

Table 3.4.6.1. Floyd-Warshall Algorithm

Now, with some modification to the Floyd-Warshall Algorithm, we can easily compute the transitive closure of graph G. Let's consider the following relation

- $T_{i,j} = 1$ if there is a path from i to j
- Otherwise, $T_{i,j} = 0$.

Since the weight of all the edges in this graph is 1, we can replace the operators "min and +" in the Floyd-Warshall Algorithm with the operators "($\vee$ and $\wedge$)" to achieve our goal (note that this algorithm is also called the original Warshall Algorithm). The Transitive Closure Algorithm (TCA) is shown in Table 3.4.6.2.

```
TransitiveClosure_Algo(A)

C = A;
n = A.length();
for k = 1 to n do
   for i = 1 to n do
      for j = 1 to n do
         t[i][j] = t[i][j] ∨ (t[i][k] ∧t[k][j]).
return C.
```

Table 3.4.6.2. Transitive Closure Algorithm

We are now ready to return to the topic of figuring out all the function-function dependency from an adjacency matrix. Figure 3.4.6.2 shows the transitive closure matrix obtained after running the TCA on the adjacency matrix shown in Figure 3.4.4.1. Note that not only does this matrix retain all the direct dependency among the nodes in that graph, but also it records all the indirect dependency. For example, cell (6, 6) indicates the indirect dependency between F_1 and F_6. The indirect dependency of every other pair of nodes can be explained in a similar fashion.

	F1	F2	F3	F4	F5	F6
F1	0	1	1	1	1	1
F2	0	0	0	0	1	0
F3	0	0	0	1	0	1
F4	0	0	0	0	0	1
F5	0	0	0	0	0	0
F6	0	0	0	0	0	0

Figure 3.4.6.2. Transitive Closure Matrix

3.4.7 Change Impact Analysis

As a CBSS evolves, changes made to one or more components happen frequently. Whenever any changes are introduced into a CBSS, it is crucial to know what changes have been made and what might be affected. The process of identifying the changes and the inadvertent effects caused by the changes is called change impact analysis. And it benefits us in the following ways

- Enhance our understanding of the system.
- Identify the potential effects caused by the changes.
- Help save time and effort in regression testing.
- Help identify and select test cases that need be executed against the changes to ensure that the system under test is still dependable and reliable.

CBSS change impact analysis is both quantitative and qualitative. Quantitative analysis refers to the measurement of the number of changed components and the amount of time spending on identifying the parts or items that would be affected by the software changes. Qualitative analysis addresses the issue of how effective the end results of the change impact analysis would help system testers determine and select the right test cases to execute against the changes. Failure to understand software changes and their potential effects on a CBSS means that we allow inadvertently ripple effects to go unnoticed until disaster occurs as in the case of the Ariane 5.

Typical changes in a CBSS include data, function, and control sequence changes. These changes are summarized in Table 3.4.7.0.1

Category	Possible Changes
Function change	1) Change function name 2) Change function return type 3) Change function scope 4) Add, delete, or change formal parameters. 5) Add or delete a function 8) Change internal logics, algorithms, or data structures 9) Change an internal-function interaction sequence
Data change	1) Change declaration or definition 2) Change data scope 3) Add or delete data.
Control sequence change	1) Delete node 2) Add node 3) Delete Link 4) Add Link

Table 3.4.7.0.1. Possible Changes Made to a Component

When any changes are made to a component, the changes will also affect the test model. Even a minor change may have ripple effect on many parts of a CBSS. In this case, the entire CBSS should be re-examined to identify the affected parts and determine what test cases should be rerun to ensure the reliability and stability of the system. Since the test model is represented using a CDG, the only two changes that can happen in the test model are node and link changes. In other words, the removal/addition of any functions from/to any component in a CBSS would cause some changes to occur in the test model. Possible changes of the test model are summarized in Table 3.4.7.0.2.

Change category	Possible changes
Test model	1) Delete node 2) Add node 3) Delete Link 4) Add Link

Table 3.4.7.0.2. Test Model Change

3.4.7.1 Component's API Firewall and Change Impact Algorithms

In order to identify the changes and their potential effects on a CBSS, we need to find a systematic way to analyze the changes. To solve this problem, we adopt the firewall concept and notation proposed by Kung et al. (1995). The technique presented in this paper, however, differs to Kung et al.'s in three fundamental ways. First, our firewall approach is constructed based on the API functions and/or data provided in the component's CIDLs (i.e., API and CII) instead of object-oriented class firewall. Secondly, our testing strategy is interface-based instead of white-box testing. Finally, the change impact algorithms we develop to compute the firewall of the CDG and the CIDL are unique, and they haven't been addressed in the literature.

Let's begin the discussion of the component firewall and change impact analysis by considering two versions of a CBSS. Let G = (V, E, D) be the current CDG where V = $\{v_1, v_2, \ldots, v_n\}$ is a finite set of vertices, each representing an API or CII function of a component. E = $\{e_1., e_2, \ldots, e_n\}$ is a finite set of edges between 2 nodes v_i and v_j. D is a set of labeled edge that either represents the data-function dependency or function-function dependency of one version of a CBSS. Likewise, let G' = (V', E', D') be the CDG resulting from the changes introduced into a CBSS, where $V' = \{v_1', v_2', \ldots, v_n'\}$ is a finite set of vertices, each representing an API or CII function of a component. $E' = \{e_1'., e_2', \ldots, e_n'\}$ is a set of edges between 2 nodes v_i' and v_j', and D' is a set of labeled edge that either represents data-function or function-function dependency of the changed CDG.

Let's review the key point of the transitive closure of a graph G defined in Definition 3.4.5.1. This definition states that G' is the transitive closure graph of G "if and only if for any given pairs of edges (a, b) and (b, c), there exists a path (a, c)." This information can now be used to define a binary relation R_F for a component's API functions as follows

$$R_F = \{<F_i, F_j> \mid <F_i, F_j> \in V \wedge <F_i, F_j, d> \in E\} \qquad (1).$$

When changes are made to any function of a component, the changed node of the CDG must also be identified. We next define the component function firewall relation, denoted as R_{FW}, which represents the changes made to a CBSS.

$$R_{FW} = R_F \cap (V'_{API} \times V'_{API}) \cap (V_{API} \times V_{API}) \qquad (2).$$

The component's API function firewall for any modified API function F_{mod}, denoted as C_{APIF} FW, can be defined as shown below

$$C_{APIF}\ FW\ (F_{mod}) = \{F_k \mid <F_{mod}, F_k> \in\ R'_{FW}\} \qquad (3).$$

Similarly, the component's API function firewall for any deleted API function F_{del}, denoted as C_{APIF} FW, can be defined as follows

$$C_{APIF}\ FW\ (F_{del}) = \{F_k \mid (\exists\ F_{del}, \exists\ F_k)\ [<F_{del}, F_k, d>] \in (E - E') \wedge <F_{del}, F_k> \in\ R'_{FW}\} \qquad (4).$$

Also, the component's API function firewall for an added API function F_{add}, denoted as C_{APIF} FW, can be defined as follows

C_{APIF} FW $(F_{add}) = \{F_k \mid (\exists F_{add}, \exists F_k) [<F_{add}, F_k, d>] \in (E' - E) \wedge <F_{add}, F_k> \in R'_{FW}\}$ (5).

With the firewall defined thus far, we can design an algorithm to identify the changes introduced into a CBSS, reconstruct a new CDG, and analyze the change impacts. Table 3.4.7.1.1 shows the algorithm for identifying the functions that would be affected by a modified function F_i of any component in a CBSS.

```
ChangeImpactAlgoForModifiedFunct(Cc.Fc, G)

1. Input:

2. Fc = array containing M modified API functions from later version of the CBSS
3. G = transitive closure graph of the previous version of the CBSS with N nodes;
4. Output:
   $F_{affected}$[], which is an array of vector of the affected functions/nodes

5. Begin:
6. String found = null; //hold a modified function
7. Vector rowElements = null; //hold elements of a row retrieved from the TC matrix
8. Vector [] colElements = null; //hold elements of a column retrieved from the TC matrix
9. For i = 1 to number of elements in Cc.Fc do
10.   found = Cc.Fc[i];
11.   rowElements = retrieveRowFromTCMatrix(found);
12.   $F_{affected}$ [i] = $F_{affected}$ [i] $\cup$ rowElements;
13.   if (rowElements != null) // sanity check
14.      for j = 0 to retrieveRowFromTCMatrix.length do
15.         colElements[j] = retrieveColumnFromTCMatrix(j);
16.         $F_{affected}$ [i] = $F_{affected}$ [i] $\cup$ colElements[j];
17.      end for
18.   end if
19. end for
20. Return $F_{affected}$;
21. end
```

Table 3.4.7.1.1. Change Impact Algorithm for Modified API Functions

From Table 3.4.7.1.1, one would realize that the key to the algorithm is the transitive closure matrix of the CDG. Given the identity of a modified API function, we can directly go to the row corresponding to that node and add all the entries with a 1 to the affected function set. Then, for each entry with a 1 in that row, we jump to the column corresponding to that entry and record all the entries with a 1 in that column. The final affected function set is the union of the set of that row (i.e., all elements in that row) and the sets of all those columns.

The same algorithm as shown in Table 3.4.7.1.1 can be used to find the affected functions as a result of any changes made to the datum D_i. The only difference between the algorithms for function change analysis and data change analysis is that the former takes the array of the modified API functions as one of its formal parameters to analyze the affected functions caused by the changes. The later, however, takes as input an array of the globally modified data as one of its formal parameters to analyze the affected functions. This means the algorithm to find the affected functions resulting from the globally modified data is exactly the same as the algorithm presented in Table 3.4.7.1.1, except that every instance of the array Fc (i.e., lines 2 and 10) would be replaced by Dc. And Dc is an array of the globally modified data from a later or modified version of the CBSS under study.

The algorithm to compute the change impact resulting from any deleted API function is similar to that of the function-modified algorithm. Table 3.4.7.1.2 shows the algorithm necessary to compute the change impact of any deleted API function.

```
ChangeImpactAlgoForDeletedFunct(Cc.Fc, G)

1. Input:
2. Fc = array containing M deleted API functions from later version of the CBSS
3. G = transitive closure graph of the previous version of the CBSS with N nodes;
4. Output: $F_{affected}$[], which is an array of vector of the affected functions/nodes

5. Begin:
6. String found = null; //hold a deleted function
7. Vector rowElements = null; //hold elements of a row retrieved from the TC matrix
8. Vector [] colElements = null; //hold elements of a column retrieved from the TC matrix
9. For i = 1 to number of elements in Cc.Fc do
10.    found = Cc.Fc[i];
11.    rowElements = retrieveRowFromTCMatrix(found);
12.    $F_{affected}$ [i] = $F_{affected}$ [i] ∪ rowElements;
13.   if (rowElements != null) // sanity check
14.       for j = 0 to retrieveRowFromTCMatrix.length do
15.          colElements[j] = retrieveColumnFromTCMatrix(j);
16.         $F_{affected}$ [i] = $F_{affected}$ [i] ∪ colElements[j];
17.       end for
18.    end if
19. end for
20. Return $F_{affected}$;
21. end
```

Table 3.4.7.1.2. Change Impact Algorithm for Deleted API Functions

For data and functions added to a newer version of a CBSS, we also need to slightly modify the input to the algorithm shown in Table 3.4.7.1.1 in order to find the impacted functions presented in the CDG. Recall that the input to the function-modified algorithm (line 3 of Table 3.4.7.1.1) is the transitive closure matrix of an older version of a CBSS. Data and functions added to a newer version of a CBSS, however, only appear in the transitive closure matrix of the newer version. As a result, the input to the algorithm that would be used to find the affected nodes of the CDG should be the transitive closure matrix of the later version. Table 3.4.7.1.3 shows the algorithm to find the affected nodes resulting from any number of functions added to any component in a CBSS. Again, if the array Fc is replaced by Dc, which is an array of M added data from a later version of the CBSS, we would obtain an algorithm to compute the affected function set resulting from the added data.

```
ChangeImpactAlgoForAddedFunct(Cc.Fc, G)

1. input:
2. Cc.Fc = array containing M added API functions from a later version of the CBSS
3. Tc = transitive closure graph of the newer version of the CBSS with N nodes.
4. Output: $F_{affected}$[], which is an array of vector of the affected functions

5. Begin:
6. String found = null; //hold an added function
7. Vector rowElements = null; //hold elements of a row retrieved from the TC matrix
8. Vector [] colElements = null; //hold elements of a column retrieved from the TC matrix
9. For i = 1 to M do
10.   found = Cc.Fc[i];
11.   rowElements = retrieveRowFromTCMatrix(found);
12.   $F_{affected}$ [i] = $F_{affected}$ [i] $\cup$ rowElements;
13.   if (rowElements != null) // sanity check
14.      for j = 0 to retrieveRowFromTCMatrix.length do
15.         colElements[j] = retrieveColumnFromTCMatrix(j);
16.         $F_{affected}$ [i] = $F_{affected}$ [i] $\cup$ colElements[j];
17.      end for
18.   end if
19. end for
20. Return $F_{affected}$;
```

Table 3.4.7.1.3. Change Impact Algorithm for Added API Functions

3.4.7.2 Component's CIDL Firewall and Change Impact Algorithms

The information discussed in this section is related to the CIDL of a CBSS, which is discussed in detail in Appendix A. Readers are encouraged to review Appendix A for a better understanding of the CIDL.

When changes are made to any component's CII of a CBSS, the CDG would also be affected. In this case, we also need to figure out the impact of the changes. Again, the firewall concept discussed in the previous section can now be applied to analyze the change impact resulting from changes made to any component's CII. Let R_{CIDL} be a binary relation for a component's CII functions. The component's CIDL firewall can be defined as follows

$$R_{CIDL} = \{<Ci.F_i, Cj.F_j> \mid <Ci.F_i, Cj.F_j> \in V \wedge <Ci.F_i, Cj.F_j, d> \in E\} \qquad (1).$$

When changes are made to any function of a component, the changed node of the CDG must also be identified. We next define the component CIDL function firewall relation, denoted as R_{CIDLFW}, which represents the changes made to a CBSS.

$$R_{CIDLFW} = R_F \cap (V'_{CIDL} \times V'_{CIDL}) \cap (V_{CIDL} \times V_{CIDL}) \qquad (2).$$

Now, let R'_{CIDLFW} be the transitive closure of R_{CIDL}. The component's CIDL function firewall for a modified function F_{mod}, denoted as $C_{CIDL}FW$, can be defined as follows

$$C_{CIDLFW}\ FW\ (C_{mod}.F_{mod}) = \{C_k.F_k \mid <C_{mod}.F_{mod}, C_k.F_k> \in R'_{CIDLFW}\} \qquad (3).$$

For a deleted function F_{del}, the component's CIDL function firewall, denoted as $C_{CIDL}FW$, can be defined as follows

$$C_{CIDL}FW\ (C_i.F_{del}) = \{C_k.F_k \mid (\exists F_{del}, \exists F_k)\ [<C_i.F_{del}, C_k.F_k, d>] \in (E - E') \wedge <C_i.F_{del}, C_k.F_k> \in R'_{CIDLFW}\} \qquad (4).$$

Similarly, the component's CIDL function firewall for an added function F_{add}, denoted as $C_{CIDL}FW$, can be defined as follows

$$C_{CIDL}FW\ (Ci.F_{add}) = \{Ck.F_k \mid (\exists F_{add}, \exists F_k)\ [<Ci.F_{add}, Ck.F_k, d>] \in (E' - E) \wedge <Ci.F_{add}, Ck.F_k> \in R'_{CIDLFW}\} \qquad (5).$$

With the component's CII firewall defined thus far, we can design an algorithm to identify the changes introduced into a CBSS, reconstruct a new CDG, and analyze the change impacts. Table 3.4.7.2.1 shows the algorithm for identifying the functions that would be affected by a modified function $C_i.F_i$ of any component's CII in a CIDL-based CBSS.

```
ChangeImpactAlgoForModifiedFunctInCIDLSyst($C_i.F_i$, G)

1. Input:
2. $F_M$ = array containing M modified functions from later version of the CBSS
3. G = transitive closure graph of the previous version of the CBSS with N nodes;
4. Output: $F_{affected}$[], which is an array of vector of the affected functions/nodes

5. Begin:
6. String found = null; //hold a modified CII function
7. Vector rowElements = null; //hold elements of a row retrieved from the TC matrix
8. Vector [] colElements = null; //hold elements of a column retrieved from the TC matrix
9. For i = 1 to number of elements in Fc do
10.    found = $C_i.F_i$ [i];
11.    rowElements = retrieveRowFromTCMatrix(found);
12.    $F_{affected}$ [i] = $F_{affected}$ [i] $\cup$ rowElements;
13.   if (rowElements != null) // sanity check
14.       for j = 0 to retrieveRowFromTCMatrix.length do
15.           colElements[j] = retrieveColumnFromTCMatrix(j);
16.         $F_{affected}$ [i] = $F_{affected}$ [i] $\cup$ colElements[j];
17.       end for
18.    end if
19. end for
20. Return $F_{affected}$;
21. end
```

Table 3.4.7.2.1. Change Impact Algorithm for Any Function Modified from Any Component's CII of a CIDL-Based CBSS.

For any deleted function specified in the changed component's CII, the algorithm to compute the change impact is similar to that of presented in Table 3.4.7.2.1. The only difference is that the algorithm used to compute the change impact resulting from any deleted CII function would take in an array of deleted CII functions for the computation instead of an array of modified CII functions. Table 3.4.7.2.2 shows the algorithm for the change impact computation of any function modified in any component's CII.

```
ChangeImpactAlgoForDeletedFunctInCIDLSyst(Ci.Fi, G)

1. Input:
2. Ci.Fi = array containing M deleted functions from later version of the CBSS
3. G = transitive closure graph of the previous version of the CBSS with N nodes;
4. Output: Faffected[], which is an array of vector of the affected functions/nodes

5. Begin:
6. String found = null; //hold a deleted function
7. Vector rowElements = null; //hold elements of a row retrieved from the TC matrix
8. Vector [] colElements = null; //hold elements of a column retrieved from the TC matrix

9. For i = 1 to number of elements in Fc do
10.    found = Ci.Fi [i];
11.    rowElements = retrieveRowFromTCMatrix(found);
12.    Faffected [i] = Faffected [i] ∪ rowElements;
13.   if (rowElements != null) // sanity check
14.       for j = 0 to retrieveRowFromTCMatrix.length do
15.           colElements[j] = retrieveColumnFromTCMatrix(j);
16.          Faffected [i] = Faffected [i] ∪ colElements[j];
17.       end for
18.    end if
19. end for
20. Return Faffected;
21. end
```

Table 3.4.7.2.2. Change Impact Algorithm for Any Function Deleted from Any component's CII of a CIDL-Based CBSS.

When any CII function is added to the CII of any component of a CIDL-based CBSS, component users and system testers also need to identify the changes made and figure out the change impact. Again, the key to compute the change impact is the transitive closure graph.

The algorithm for the change impact computation of any function added to the CII of any component of a CIDL-based CBSS is shown in Table 3.4.7.2.3.

```
ChangeImpactAlgoForAddedFunct($C_i.F_i$, G)

1. Input:
2. $C_i.F_i$ = array containing M added CII functions from a later version of the CBSS
3. Tc = transitive closure graph of the newer version of the CBSS with N nodes.
4. Output: $F_{affected}$[], which is an array of vector of the affected functions
5. Begin:
6. String found = null; //hold a function
7. Vector rowElements = null; //hold elements of a row retrieved from the TC matrix
8. Vector [] colElements = null; //hold elements of a column retrieved from the TC matrix
9. For i = 1 to M do
10.   found = $C_i.F_i$ [i];
11.   rowElements = retrieveRowFromTCMatrix(found);
12.   $F_{affected}$ [i] = $F_{affected}$ [i] $\cup$ rowElements;
13.   if (rowElements != null) // sanity check
14.      for j = 0 to retrieveRowFromTCMatrix.length do
15.         colElements[j] = retrieveColumnFromTCMatrix(j);
16.         $F_{affected}$ [i] = $F_{affected}$ [i] $\cup$ colElements[j];
17.      end for
18.   end if
19. end for
20. Return $F_{affected}$;
```

Table 3.4.7.2.3. Change Impact Algorithm for Any Function Deleted from Any component's CII of a CIDL-Based CBSS.

3.4.7.3 Test Change Firewall

When changes are made to a component, regardless of its component's API or CIDL, component users and system testers would also be interested in seeking the answers to the following questions

- Which test case can be reused?

- Which test case is outdated?
- Which new test case should be added to the system test suite?

Recall that a CBSS test suite has already been defined in Definition 3.3.4. From this definition we know that each component's API function or CII function represents a node in the CBSS CDG. Another important point of Definition 3.3.4 is that a test set is uniquely designed for each node of the CDG, and this is the key that helps answer the three questions mentioned at the beginning of this section.

When changes are made to any API or CII function, the test set associated with that function is certainly affected. Also, the changes would affect the system test suite. In order to identify the affected test cases, we define the test change firewall that helps component users and system testers identify the test suite impact. Let $C_i.F_i$ be a changed function F_i of a component C_i, and let T_i be the test set associated with F_i. The test change firewall for the CIDL-based CBSS can be defined as follows

$R_{CIDL} = \{<C_i.F_i.T_{i.}, C_j.F_j.T_i> \mid <C_i.F_i.T_i, C_j.F_j.T_i> \in V \wedge <C_i.F_i.T_i, C_j.F_j.T_i> \in E\}$ (1).

When changes are made to any function of a component, the changed node of the CDG must also be identified. We next define the component CIDL test firewall relation, denoted as $R_{CIDLTFW}$, which represents the changes made to the CIDL of a CBSS.

$R_{CIDLTFW} = R_F \cap (V'_{CIDL} \times V'_{CIDL}) \cap (V_{CIDL} \times V_{CIDL})$ (2).

Now, let $R'_{CIDLTFW}$ be the transitive closure of $R_{CIDLTFW}$. The test change firewall for a modified function F_{mod}, denoted as $C_{CIDL}TFW$, can be defined as follows

$C_{CIDL}\ TFW\ (C_i.F_{mod}.T_i) = \{C_k.F_k.T_k \mid <C_i.F_{mod}.T_i, C_k.F_k.T_k> \in R'_{CIDLTFW}\}$ (3).

For a deleted function F_{del}, the test change firewall, denoted as $C_{CIDL}TFW$, can be defined as follows

$$C_{API}\ TFW\ (C_i.F_{del}.T_i) = \{C_k.F_k.T_k \mid (\exists\ F_{del}, \exists\ F_k)\ [<C_i.F_{del}.T_i,\ C_k.F_k.T_i>] \in (E - E') \wedge <C_i.F_{del}.T_i,\ C_k.F_k.T_i> \in\ R'_{CIDLTFW}\} \quad (4).$$

Similarly, the test change firewall for an added function F_{add}, denoted as $C_{CIDL}TFW$, can be defined as follows

$$C_{CIDL}TFW\ (C_i.F_{add}.T_i) = \{C_k.F_k.T_k \mid (\exists\ F_{add}, \exists\ F_k)\ [<C_i.F_{add}.T_i,\ Ck.F_k.T_k>] \in (E' - E) \wedge <C_i.F_{add}.T_i,\ C_k.F_k.T_k> \in\ R'_{CIDLTFW}\} \quad (5).$$

3.5 Component Function Sequence Representation

A component function sequence diagram is a graph that represents both the external stimuli (i.e., input from users) and the interaction sequence among the functions of the components in a CBSS. Each node represents a distinct function that either receives the input from the user or invokes another function for help on completing a service requested. Nodes that receive users' input are connected to one another by bi-directional links while nodes representing function dependency among the components are connected to one another by directed arcs. Each directed edge from node i to node j implies that function i and function j depend on each other to perform a specific task.

Definition 3.2.5.1 Component Function Sequence Diagram—a component function sequence diagram is a quadruple $G = (F_u, F_d, E_u, E_d)$ where

- $F_u = \{ F_{u1}, F_{u2}, \ldots, F_{un}\}$ is a finite set of API functions through which users can interact.
- $F_d = \{ F_{d1}, F_{d2}, \ldots, F_{dn}\}$ is a finite set of API functions; each represents a function of a component that would help carry out some specific task in response to a call from a function $F_u i$.
- $E_u = \{E_{u1}, E_{u2},\ldots,E_{un}\}$ is a finite set of bidirectional links. Each edge $Eui \in E_u$ connects a function $F_u i$ to a function $F_u j$.
- $E_d = \{E_{d1}, E_{d2},\ldots,E_{dn}\}$ is a finite set of directed arcs. Each arc $E_d i \in E_d$ connects a function $F_u i$ or $F_d i$ to a function $F_d j$.

3.6 Coverage Criteria

Coverage criteria can be defined as a set of rules that act as the guidelines to help determine the testing thoroughness of a CBSS. Since the test model is graph-based, a number of well-known test coverage criteria can be used. These include node coverage, link coverage, conditional link coverage, and path coverage criteria. Because of the timing constraint, it is impossible to address all of these coverage criteria. Our goal is to focus on only three coverage criteria: node, path, and edge. Each of these coverage criteria will be discussed shortly.

3.6.1 Node Coverage Criterion

If P is defined as a set of complete paths of a digraph G, P satisfies the all-node criterion if and only if it includes a set of all nodes in G. This coverage criterion can easily be achieved by observing the following three key points of the test model

- The CDG is represented as a graph.
- The data-function dependency is recorded in the corresponding transitive closure matrix.
- The function-function dependency is recorded in the corresponding transitive closure matrix.

The three key points just discussed mean that node coverage can be obtain directly from the transitive closure matrix of the CDG.

3.6.2 Path Coverage Criterion

If P is defined as a set of complete paths of a digraph G, P satisfies the all-path criterion if and only if it includes a set of all paths in G. If only simple paths are considered, this criterion can easily be achieved using either of the 2 well-known search algorithms: Breadth-First-Search (BFS) or Depth-First-Search (DFS). If a path exists between node i and node j, we want to make sure that there is at least one test case covering that path. We use BFS as the algorithm to find whether there exists a path from one node to another. The algorithm for traversing a CDG and finding a path from one node to another using BFS is outlined in Table 3.6.2.1.

Note that the algorithm shown in Table 3.6.2.1 can only find one path beginning from the source node and ending at the destination node. In order to find all simple paths in a graph G, we need to call the pathSearch algorithm with all possible combination of the source and destination nodes. Again, since the dependency among the API and CII functions of a CBSS is stored in the transitive closure matrix, it's relatively easy to come up with an algorithm to accomplish the task. The algorithm to find the set of all the simple paths of a graph G is called TraverseAll, which is shown in Table 3.6.2.2.

```
Procedure pathSearch (src, dest, Graph G)

Input: src, dest;  // source node and destination node
         G; // graph
Output: array stored the path from the source to the destination node

String pathHolder [];
pathHolder[0] = src;
if (src == dest)
   return pathHolder;
String pathReturn[];
Queue Q;
Q.insert(src); //insert src node at the tail of the Q
//recursively search for path to destination using BFS
While (!Q.isEmpty())
{
   String aNode = Q.remove(); //remove a vertex from the head of the Q
   int aNodeIndex = G.getNodeIndex(aNode); //obtain the index of the node
   // iterator to traverse the graph
   Iterator iter = G.iterator(aNode);
   while (iter.hasNext())  //iterate until has no neighbor
   {
      String nextNode = iter.next();
      Int nextNodeIndex = G.getNodeIndex(nextNode);
      if (src != dest && pathHolder[nextNodeIndex] == null)
      {
         Q.insert(nextNode); //has no neighbor, so insert to Q
         pathHolder[nextNodeIndex] = aNode;
      }
   }
}
   // store path in pathReturn array
   for (int i = pathHolder.length; i  >=0;  i--)
   {
      pathReturn[i] = pathHolder[i];
   }
   pathReturn[0] = src;
   return pathReturn;
}
```

Table 3.6.2.1. Algorithm to Find a Path from One Node to Another

```
Procedure traverseAll ( Graph G )

len = G.length();
for (int i = 0; i < len; i++)
   for (int j = 0; j < len; j++)
      pathSearch(i, j);
```

Table 3.6.2.2. Path between All-Node-Pair Algorithm.

3.6.3 Link Coverage Criterion

If N is defined as a finite set of paths of a digraph G, N satisfies the all-link criterion if and only if it includes all the links in G. Again, this goal can also be achieved by traversing G using either BFS or DFS. Since a link e_i is just an arc that connects two nodes j and k, all-link coverage would be achieved if all-node coverage is guaranteed. In fact, the algorithms shown in Tables 3.6.2.1 and 3.6.2.2 guarantee all-node coverage, so the all-link coverage is automatically covered. Also, we can implement DFS as the search method and traverseAllDFS as an algorithm to achieve the all-link criterion. Because of the timing constraint and for the sake of simplicity, we decided to choose the former approach to satisfy the all-link coverage criterion.

4. Case Study—the Simulated Elevator System

In this chapter, we discuss how the test model can be applied on a simulated elevator system. The simulated elevator system is a CBSS that has been designed by a group of graduate students at SJSU. It consists of 4 components: floor panel, operation panel, elevator, and up-down controller. This system is shown in Figure 4.0.1.

The simulated elevator is a Graphical User Interface (GUI) CBSS in which the two components—OperationPanel and floorPanel—act as the user's access points (i.e., users can interact with these two components). The OperationPanel provides users with 7 buttons. When users press the buttons labeled 1, 2, or 3, the elevator would move to the appropriate floor level. The OpenDoorButton and CloseDoorButton can be used to open or close the elevator's door. Similarly, users can press the StopButton button any time to stop the elevator. The dialButton is provided so that users can make a call for help in case of emergency. The floorPanel component provides component users with 4 buttons: upButtonL1, upButtonL2, downButtonL2, and downButtonL3 (we show this component with only 2 buttons in Figure 4.0.1 to simplify the presentation). All the 4 buttons just mentioned will be shown in Figure 4.2.3.2. Users can press any of these buttons to request the elevator to move to the appropriate floor level.

The other two components—Elevator and Controller—are invisible to component users. They function behind the "scene" to help the OperationPanel and the Floorpanel components to fulfill users' requests.

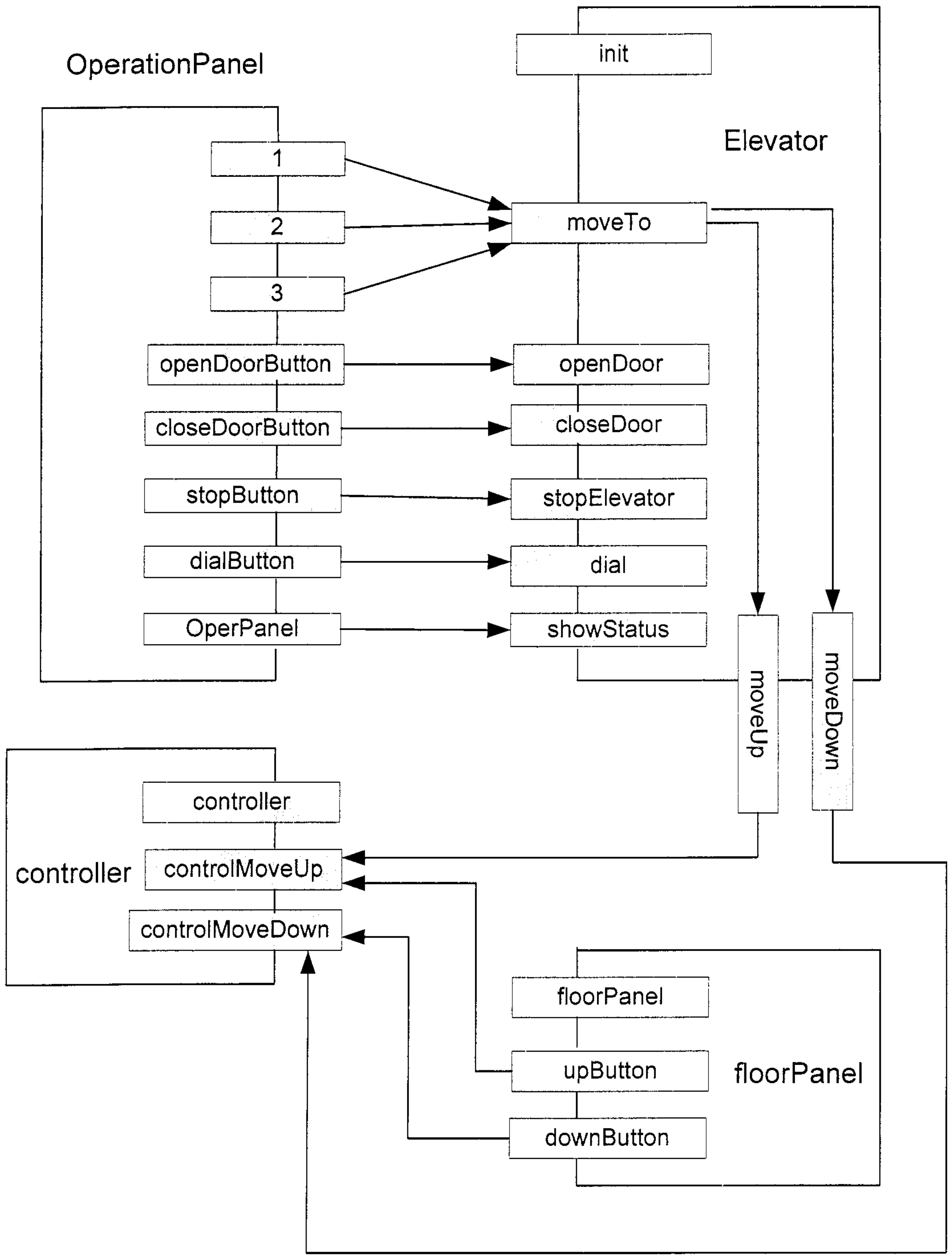

Figure 4.0.1. The Simulated Elevator CBSS.

In this chapter, we only show the CIDL of the API functions of the OperationPanel component and the CII for the closeDoorButton of the elevator component. The complete CIDL description of the elevator system can be found in Appendix B.

```
Component-API: Comp-operationaPanel, V. 1.0
{
  Function-Signatures:
     Button1;
     Button2;
     Button3;
     openDoorButton;
     closeDoorButton;
     stopButton;
     dialButton;
     OperPanel;
}
```

Figure 4.0.2. API of the OperationPanel

Figure 4.0.2 shows the API of the OperationPanel component, and Figure 4.0.3 partially presents the CII of the OperationPanel and the Elevator components

```
Component-interact: Comp-OperationPanel, V. 1.0; Comp-Elevator, V. 1.0;
{

  Port: Comp-OperationPanel, V. 1.0; Comp-Elevator, V. 1.0;
  {
   Comp-OperationPanel:closeDoorButton R*
   CompElevator-V1:closeDoor();
  }
}
```

Figure 4.0.3. CII of the OperationPanel and the Elevator Components

4.1 Identifying the Simulated Elevator's Components

Figure 4.1 is a pictorial representation of the simulated elevator system. From that Figure, one can easily identify not only the component constituents of the component-based elevator system, but also the dependency between one component and the others. For example, the openDoorButton API function of the OperationPanel component depends on the openDoor function of the Elevator component to fulfill a request to open the elevator's door.

4.2 Dependency Analysis

According to the API and specifications, the simulated elevator CBSS has 3 global variables: currentLevel, doorStatus, and elevatorStatus. The function-function dependency associated with each of these global variables will be discussed in a separate section shortly.

4.2.1 Dependency Associated with CurrentLevel

Figure 4.2.1.1 shows the data-function dependency associated with the currentLevel variable among the components in the simulated elevator. All the functions shown in this Figure define, use, or both define and use the currentLevel variable. For example, the function moveTo (a function of the Elevator component) uses the variable currentLevel. Also, the function moveUp (also a function of the Elevator component) both defines and uses currentLevel. So, changes occur to currentLevel mean that all of these functions will be affected.

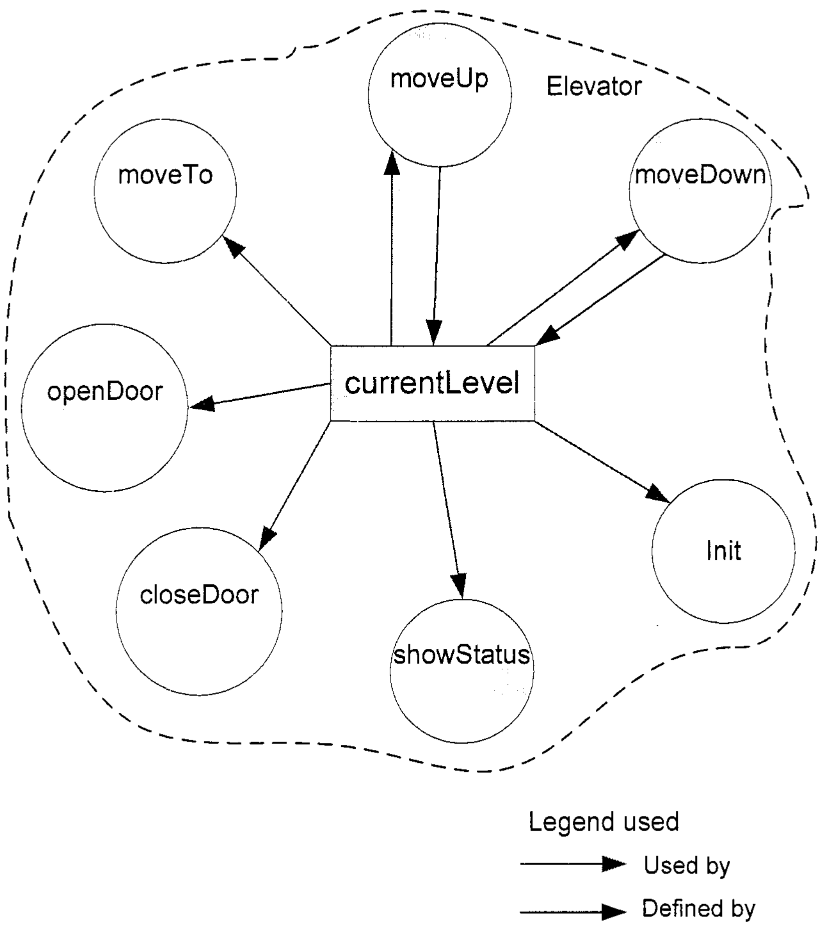

Figure 4.2.1.1. Data-Function Dependency Associated with CurrentLevel

Figure 4.2.1.2 is a pictorial representation of the function-function dependency associated with the currentLevel variable.

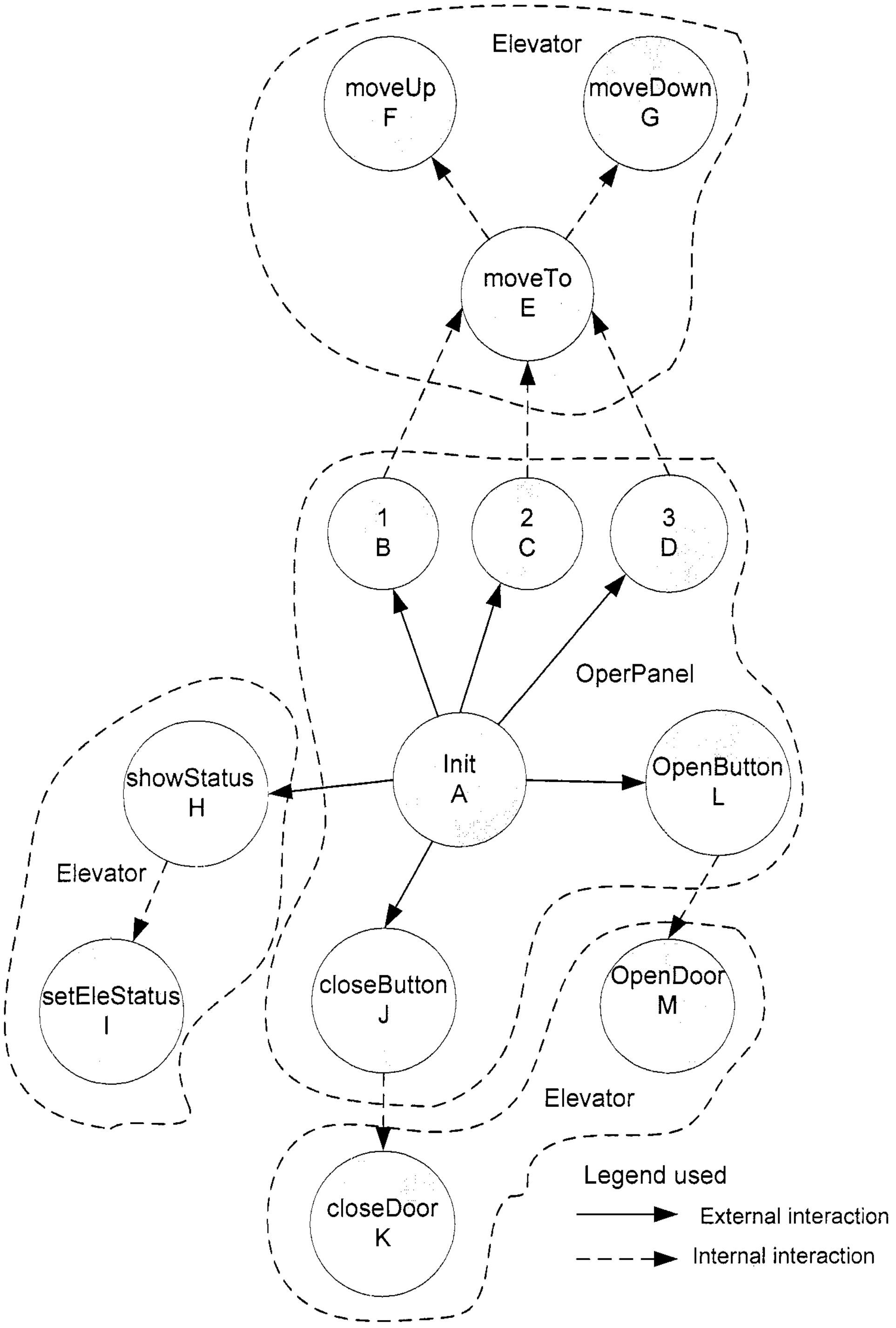

Figure 4.2.1.2. Function-Function Dependency Associated with CurrentLevel

Figure 4.2.1.3 shows the adjacency matrix whose information is derived from Figure 4.2.1.2. This matrix shows all the direct dependency among the functions of the component-based elevator system. All these functions either use or define the currentLevel variable. Later on, the transitive closure algorithm will be used to find all the indirect dependency associated with this variable.

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	0	1	1	1	0	0	0	1	0	1	0	1	0
B	0	0	0	0	1	0	0	0	0	0	0	0	0
C	0	0	0	0	1	0	0	0	0	0	0	0	0
D	0	0	0	0	1	0	0	0	0	0	0	0	0
E	0	0	0	0	0	1	1	0	0	0	0	0	0
F	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	0	0	0	0	0
H	0	0	0	0	0	0	0	0	1	0	0	0	0
I	0	0	0	0	0	0	0	0	0	0	0	0	0
J	0	0	0	0	0	0	0	0	0	0	1	0	0
K	0	0	0	0	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	0	0	0	0	0	0	0	1
M	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4.2.1.3. Adjacency Matrix for CurrentLevel

4.2.2 Dependency Associated with DoorStatus

Figure 4.2.2.1 shows the data-function dependency associated with the doorStatus variable among the components in the simulated elevator. Again, all the functions shown in this Figure either define or use the doorStatus variable. So, if any change occurs to the doorStatus variable, all of these functions will be affected. That is, any change occurred to doorStatus will affect all of these functions no matter if they define or use doorStatus. This is the ripple effect that has been mentioned throughout this paper.

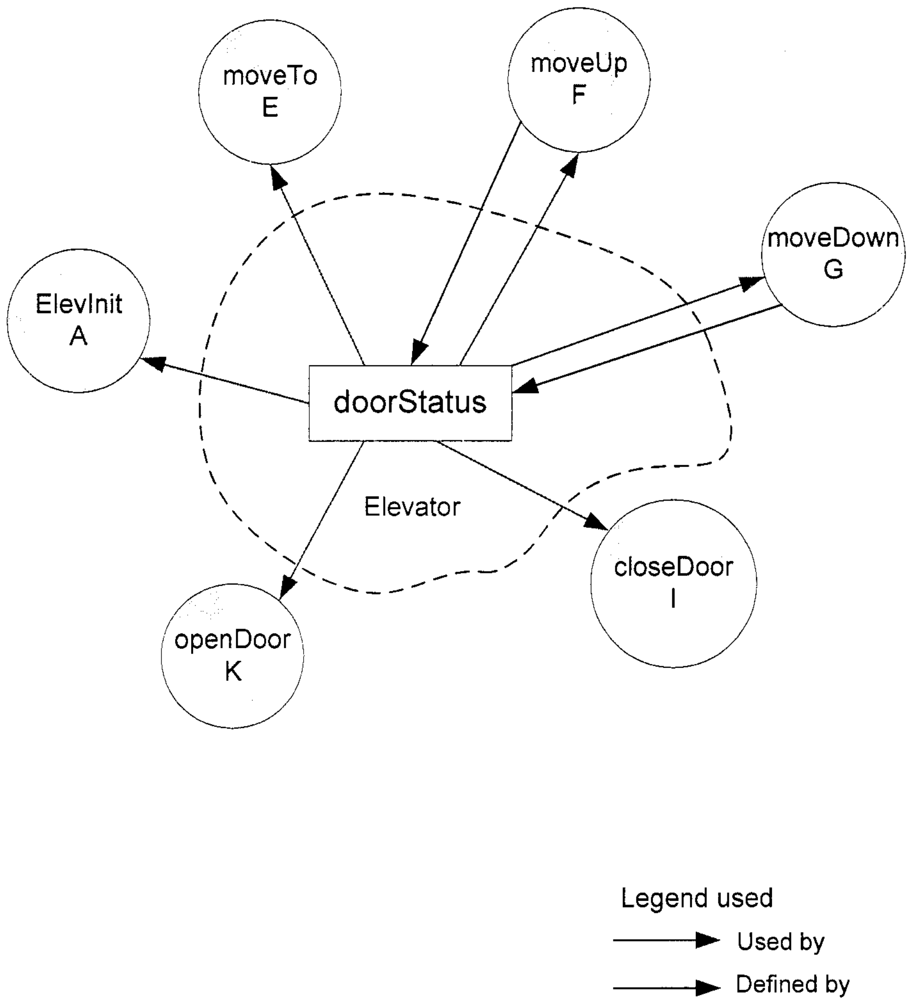

Figure 4.2.2.1. Data-Function Dependency Associated With the DoorStatus Variable.

Figure 4.2.2.2 shows the function-function dependency associated with the doorStatus variable among the components in the simulated elevator system. Again, all the functions shown in this Figure either define or use the doorStatus variable. So, if any change occurs to the doorStatus variable, all of these functions will be affected.

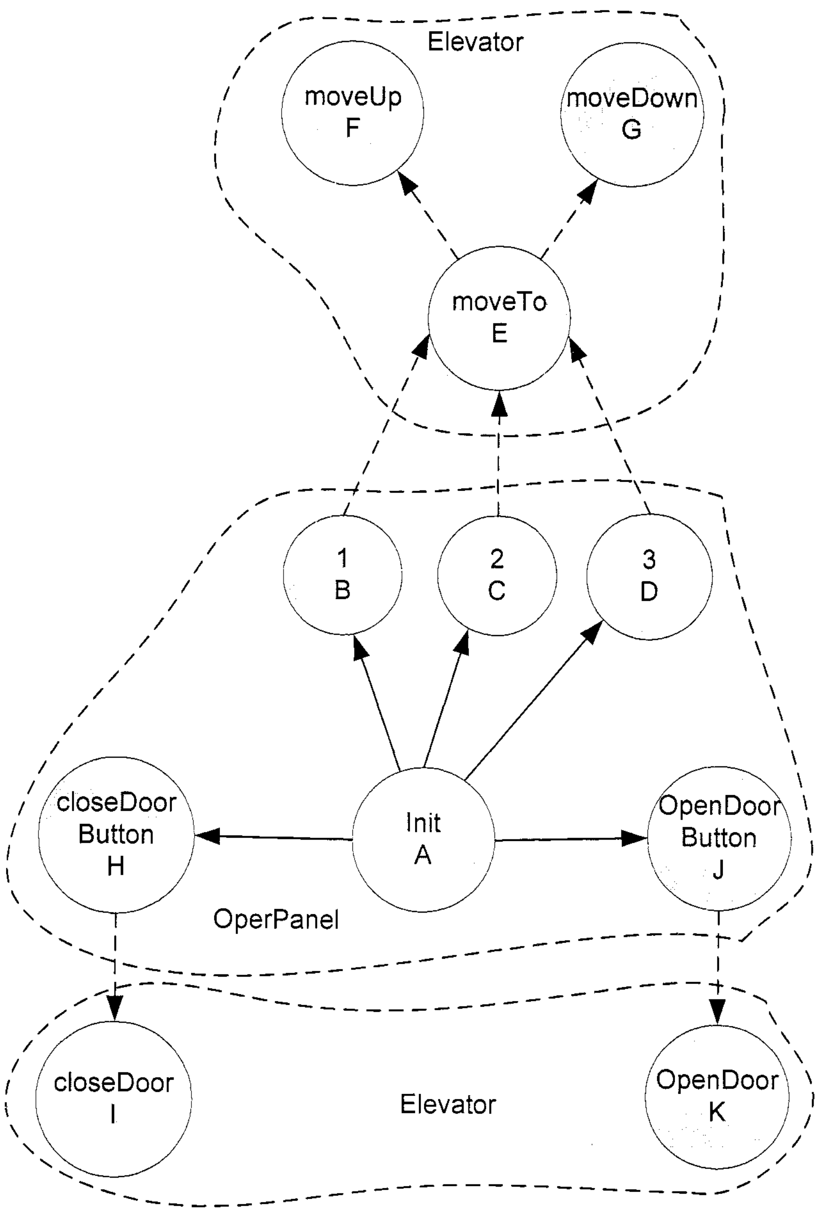

Figure 4.2.2.2. Data-Function Dependency Associated with The DoorStatus Variable

Figure 4.2.2.3 is the adjacency matrix whose dependency information is directly taken from Figure 4.2.2.2. Again, this matrix only shows all the direct dependency among the component constituents of the elevator system.

	A	B	C	D	E	F	G	H	I	J	K
A	0	1	1	1	0	0	0	1	0	1	0
B	0	0	0	0	1	0	0	0	0	0	0
C	0	0	0	0	1	0	0	0	0	0	0
D	0	0	0	0	1	0	0	0	0	0	0
E	0	0	0	0	0	1	1	0	0	0	0
F	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	0	0	0
H	0	0	0	0	0	0	0	0	1	0	0
I	0	0	0	0	0	0	0	0	0	0	0
J	0	0	0	0	0	0	0	0	0	0	1
K	0	0	0	0	0	0	0	0	0	0	0

Figure 4.2.2.3. Adjacency Matrix for DoorStatus

4.2.3 Dependency Associated with ElevatorStatus

Figure 4.2.3.1 shows the data-function dependency associated with the elevator Status variable among the components in the simulated elevator.

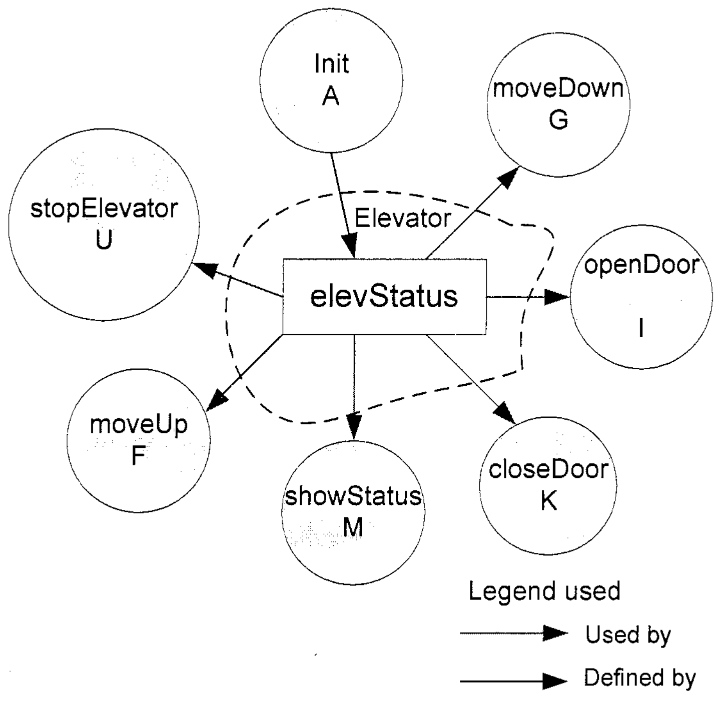

Figure 4.2.3.1. Data-Function Dependency Associated with ElevatorStatus

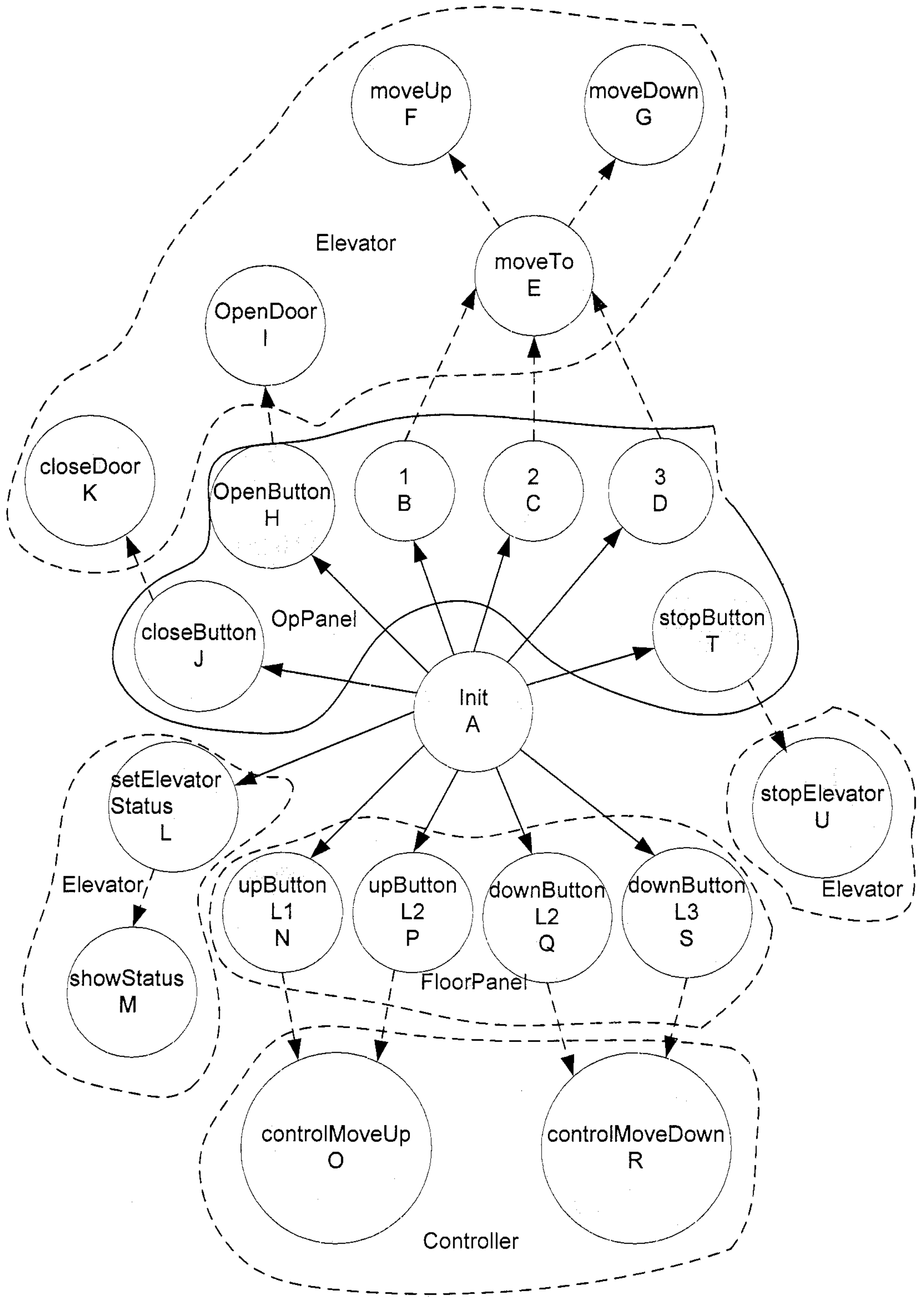

Figure 4.2.3.2. Function-function Dependency Associated with ElevatorStatus

Figure 4.2.3.2 shows the function-function dependency associated with the elevatorStatus variable among the components in the simulated elevator system, and Figure 4.2.3.3 is the adjacency matrix whose dependency information is directly taken from Figure 4.2.3.2.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
A	0	1	1	1	0	0	0	1	0	1	0	1	0	1	0	1	1	0	1	1
B	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
D	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
F	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
H	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
I	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
K	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
M	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
N	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
O	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
Q	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
T	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
U	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4.2.3.3. Adjacency Matrix for ElevatorStatus

4.2.4 Complete Elevator System Dependency

The complete picture showing the relationship among all the functions of the component constituents of the component-based elevator system is shown in Figure 4.2.4.1.

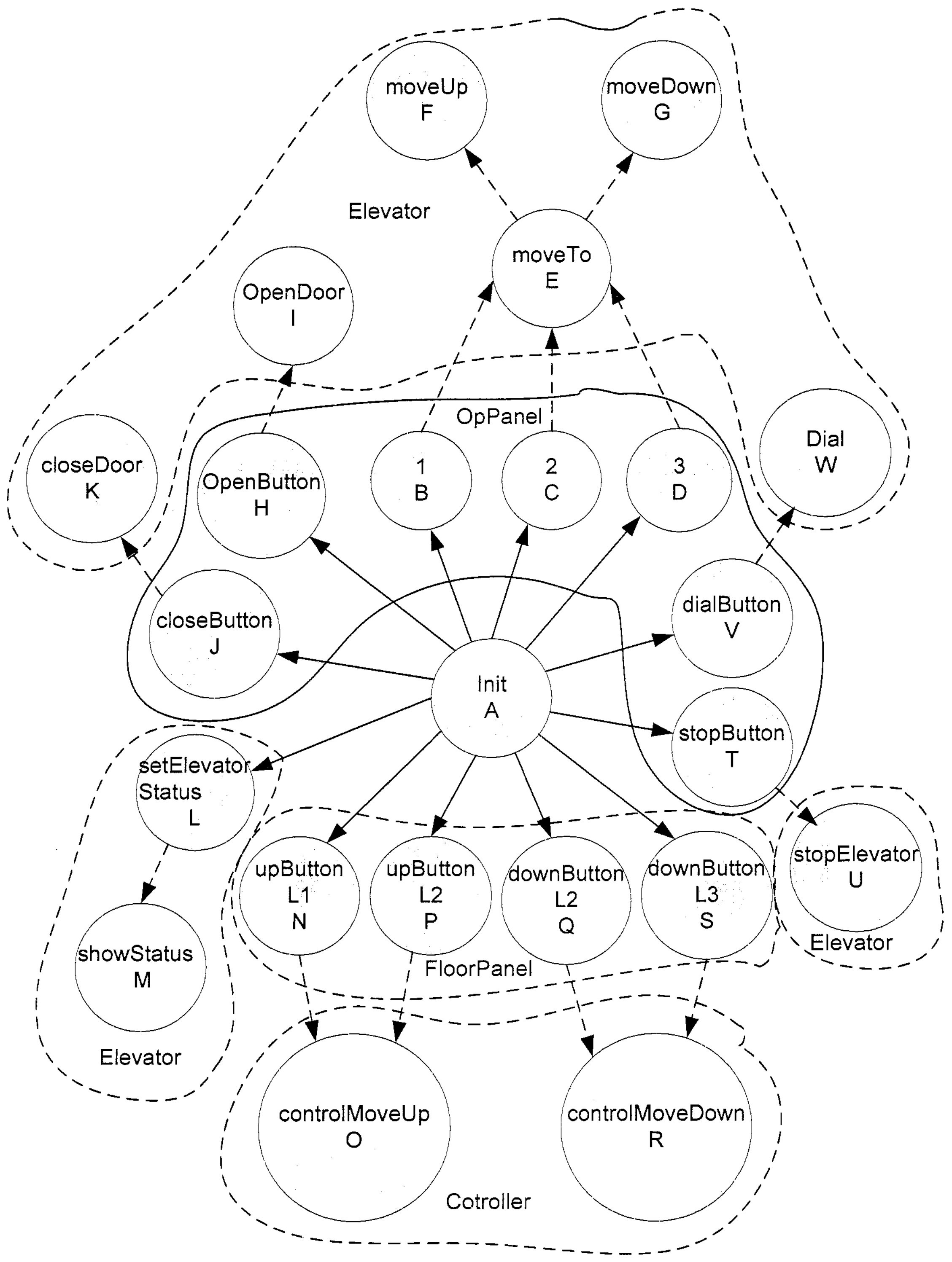

Figure 4.2.4.1. Complete Elevator System Dependency

Figure 4.2.4.2 shows the adjacency matrix whose information is extracted from Figure 4.2.4.1.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
A		1	1	1	0	0	0	1	0	1	0	1	0	1	0	1	1	0	1	1
B	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
D	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
F	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
H	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
I	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
K	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
M	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
N	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
O	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
Q	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
T	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
U	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
V	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
W	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4.2.4.2. Adjacency Matrix for the Simulated Elevator System

The adjacency matrix of the simulated elevator system, after being input into the transitive closure algorithm, would yield the transitive closure matrix as shown in Figure 4.2.4.3. Note that this Figure displays all the direct as well as indirect dependency among the component constituents of the elevator system. The information presented in this matrix will later be used as the input the Change Impact Analysis Algorithm to figure out the possible impacts on the system if any changes have been made to any of functions of any components.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
A		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
B	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
C	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
D	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
F	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
H	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
I	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
K	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
M	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
N	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
O	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
Q	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
T	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
U	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
V	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
W	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4.2.4.3. Transitive Closure Matrix for the Simulated Elevator System

4.3 Change Impact Analysis

The information showing so far is extracted from version 1 of the elevator system. In version 2, a few data and function changes were done on the system. Now, let's see how the test model can be used to help component users and testers verify the effectiveness of those changes.

Function added	Function deleted	Function modified	Data added	Data deleted	Data modified
cntlMoveTo	moveTo	moveUp			elevatorStatus
		moveDown			
		cntlMoveDown			
		cntlMoveUp			

Figure 4.3.1. Changes Made to Version 2.

Let's take the change made to the function moveTo as an example to illustrate how the test model can be used to determine the change impact. Figure 4.3.2 shows the selected parts of the elevator system for the purpose of change impact discussion. Note that this Figure indicates moveTo is the node/function that has been removed. Now, let's see how effective the Function Change Impact Algorithm shown in Table 3.4.7.2 can be used to identify the change impact of moveTo on the elevator system. Following are the steps the algorithm would cover to identify the change impact

1) Step 1: go to row E (i.e., corresponding to the moveTo node) of the transitive closure matrix representing the elevator system, retrieve all elements with a 1 value, and store it in some buffer. Let's call this set of elements as RowE. Clearly, RowE = {F, G}.

2) Step 2: for every element in RowE, go to the column corresponding to that element, retrieve all elements with a 1 value, and store it in some buffer. Assuming elements with a 1 value in column F is named ColumnF set and that of for column G is named ColumnG set, we obtain the following results

   - ColumnF set = {A, B, C, D, E}, and
   - ColumnG set = {A, B, C, D, E}.

3) Step 3: the affected functions are the results of the union of RowE, ColumnF, and ColumnG sets. These functions include A (not shown in Figure 4.3.2), B, C, D, E, F, and G.

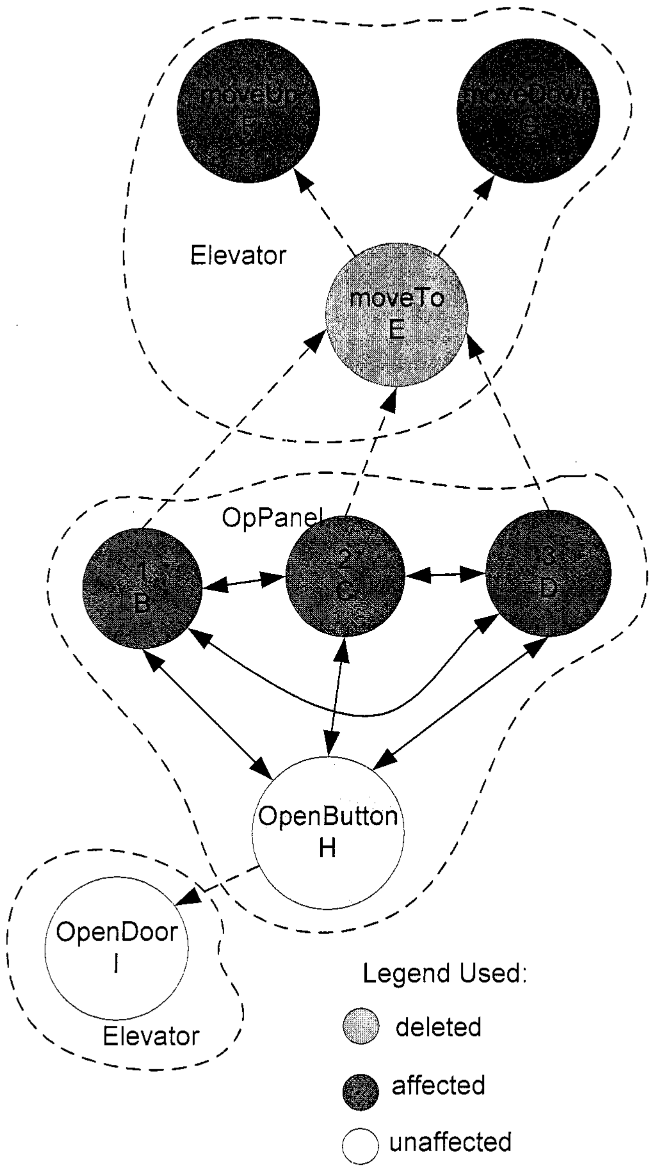

Figure 4.3.2. Change Impact Analysis for the Removal of the MoveTo Node

5 Conclusion and Future Directions

5.1 Conclusion

Although the trend towards CBSS has been increasingly growing since 1990, much of the work thus far has focused on issues related to architecture and design. Very few papers have addressed CBSS testing even though the cost of testing and maintaining a software system is more half its total cost.

This paper proposes a systematic interface-based testing approach for testing CBSS's. The test model is constructed from the information provided in the component's well-defined interfaces and specifications. By going through the steps necessary to build the test model, represented as a component dependency graph, system testers can better understand the functionality of a CBSS and learn how test cases should be designed to test it. Overall, the test models offer software system testers the following advantages:

- Construct a test model for any CBSS, independent of platform and programming language.
- Gain better understanding of the functionality and features provided by a CBSS under test.
- Learn how to identify components in a CBSS and how to represent the whole system as a graph that shows the interaction among the components.
- Observe and learn how to find the direct dependency among the components of a CBSS.
- Observe and learn how to find the indirect dependency among the components of a CBSS.

- Select test coverage criteria to meet the timing and budget constraints.
- Provide guidelines for change impact analysis, which is necessary for maintaining a CBSS.

5.2 Future Directions

Even though the proposed method can be used to effectively build a test model for CBSS's, the steps involved might become a tedious process in constructing a test model for large and complex CBSS's. Future work would involve the developing of a technique that can automate the tasks of component identification and dependency detection. One way to automate these tasks is the need of establishing some standard for specifying and representing the component dependency in the specification or API. In this case, CIDL (the topic of Appendix A) is a promising standard that would help facilitate the test automation.

While waiting for such a standard to become available, we will further enhance our technique and conduct an empirical study on large-size CBSS's. Also, we will consider expanding the capability of the test model so that it can dynamically interact with the CBSS under test to better understand the internal interaction of a component. Furthermore, we would consider supporting the more challenging coverage criteria such as path coverage and condition-link coverage.

References

Beydeda, S. & Gruhn (2001), V. *An Integrated Testing Technique for Component-Based Software*. ACS/IEEE International Conference on 25-29 June 2001, 328 – 334.

Cechich, A., Piattini, M., Vallecillo, A. (2003). *Component-Based Software Quality Methods and Technique.* New York: Springer 2003.

Crnkovic, I. & Larsson, M. (2002). *Building Reliable Component-Based Software System.* Boston: Artech House.

D'Souza D. F. & Wills A. C. (1997). *Objects, Components, and Frameworks with UML—the Catalysis Approach.* Mass.: Addison-Wesley.

Gao, J. Z., Tsao, J., Wu, Y. (2003). *Testing and Quality Assurance for Component-Based Software.* MA: Artech House.

Goulad, M., Brito, E. A. F (2002). *The Quest for Software Components Quality.* Proceedings of the 26th Annual International, 26-29 Aug. 2002, 313-318.

Kung, D. C., Gao, J., Hsia, P., Lin, J., Toyoshima, Y (1995). Class Firewall, Test Order and Regression Testing for Object-Oriented Programs. *Journal of Object-Oriented Programming*, 51-65.

Haddox J. M., Kapfhammer, G. M., Michael, C. C. (2002). *An Approach for Understanding and Testing Third Party Software Components*. Proceedings of the 48th Reliability and Maintainability Symposium. Seattle, WA, January, 2002

Harrold, M. J. (2000). *Testing: A Roadmap*. International Conference on Software Engineering; Proceedings of the Conference on The Future of Software Engineering, Limerick, Ireland, May 2000, 61 - 72

Lau, K. K. (2004). *Component-Based Software Development: Case Studies*. New Jersey: World Scientific.

Harrold, J. M., Orso, A., Rosenbum, D. Rothermel, G. (2001). *Using Component Metadata to Support the Regression Testing of Component-Based Software*. Proceedings of the IEEE International Conference on Software Maintenance on 7-9 Nov. 2001, 716 - 725

Mehta, A. & Heineman, G. T. (2002). *Evolving Legacy System Features into Fine-Grained Components*. Proceedings of the 24th International Conference on Software Engineering, 2002, 417 – 427.

Orso. A., Harrold M. J., Rosenbum D. (2000). *Component Metadata for Software Engineering Tasks*. Second International Workshop on Engineering Distributed Objects, 129-144.

Rosenblum S. D. (1997). *Adequate Testing of Component-Based Software*. Retrieved October 17, 2005, from www.ics.uci.edu/~dsr/ics9734.pdf

Strembeck, M. & Zdun, U. (2005). *Scenario-based Component Testing Using Embedded Metadata* Retrieved on October 20, 2005, from http://wi.wu-wien.ac.at/~uzdun/publications/tecos04.pdf

Szyperski, C., Gruntz, D., Murer, S. (2002). *Component Software Beyond Object-Oriented Programming*. Boston: Addison-Wesley.

Voas, J. M. (June 1998), Certifying Off-The-Shelf Components. *Computer, 31* (6), 53-59

Voas, J. M. (June 1998). The Challenges of Using COTS Software in Component-Based Development. *Computer, 31* (6), 44-45.

Weyuker, J. E. (1998). Testing Component-Based Software: A Cautionary Tale. *IEEE Software*, 15 (5), 54 - 59

Xie, G. (2004). *Decomposition Verification of Component-Based Systems—A Hybrid Approach*. Proceedings of the 19th International Conference on Automated Software Engineering on 2004, 414-417.

Xie, G. & Zhe, D. (2004). *Model-Checking Driven Black-Box Testing Algorithm for Systems with Unspecified Components.* Retrieved on October 27, 2005, from http://arxiv.org/PS_cache/cs/pdf/0404/0404037.pdf

Ye W., Offutt, J. (2003). *Maintaining Evolving Component-Based Software with UML.* Proceedings of the Seventh European Conference on 26-28 Mar. 2003, 133-142.

Appendix A. Component Interface Description Language

This Appendix includes detailed descriptions of the Component Interface Description Language (CIDL), which is an innovative approach being proposed by Dr. Jerry Gao at SJSU to facilitate the task of CBSS testing.

A.1 Motive

As mentioned throughout this dissertation, CBSS testing is very difficult for two main reasons. First, components are available on the market as COTS. Being unable to access the source code of a CBSS means that users have no choice other than performing black-box testing to ensure the quality and reliability of that CBSS. The other dilemma is that the documents shipped with those components (i.e., component's API and functional specifications) are not well written. This lack of well-defined APIs and specifications has proved to make CBSS testing become a very difficult task.

Dr. Gao has explained the need for establishing the CIDL standard as follows

- Provide a standard representation for components.
- Help separate a component's interface and the interaction interface from its implementation.
- Specify the interaction of the components in their interfaces instead of hiding it as in the conventional approach.

A.2 Component Interface Description Language

To overcome the two problems mentioned in the previous section, Dr. Gao proposes that each component should have two interfaces: component API and Component Interaction Interface (CII). Component API refers to the API of a component in which all of its externally accessible functions are defined and accessible by component users. In other words, the API of a component is the interface through which users can access.

The CII is a new concept. Its name implies the specification of the interaction between one function of a component and the function of another component of a CBSS. Through this interface, component users can easily identify the dependency among all the components of a CBSS.

Figure A.2.1 is a pictorial representation of a CBSS, which have an API and a CII. This diagram shows that the component example consists of 4 components: Component Under Test (CUT), component 1, component 2, and component 3. Note that the CUT has its own API, which is accessible by component users. In Addition, each component has its own CIDL interface, in which the dependency between one function of a component and the function of another component is specified. For example, CIDL-1 clearly tells users how the CUT would interact with Comp-1. The interaction between the CUT and Comp-2 or the dependency between the CUT and Comp-3 can be explained in a similar fashion.

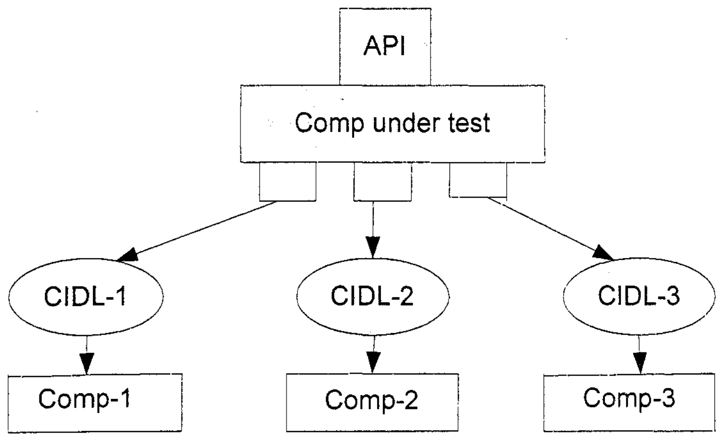

Figure A.2.1. CBSS Described by CIDL

The main advantages of establishing the CIDL standard are pointed by Dr. Gao as follows

- Increase component testability and controllability
- Facilitate component unit, integration, and system testing.
- Facilitate system assembly, change, control, and deployment.

Figure A.2.2 shows an example of a two-component CBSS. The component Comp-1 is the under-test component. The functions F1 and F2 are its API while port 1 is its CII. Its API offers the functions F1 and F2 through which component users can access. The CII specifies the interaction between Comp-1 and Comp-2. The arrow from port 1 to G1 indicates the dependency between Comp-1 and Comp-2.

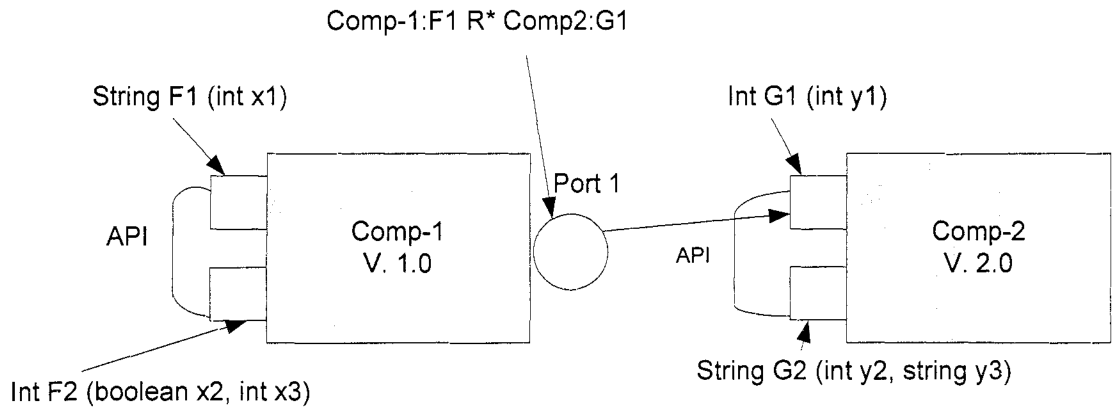

Figure A.2.2. A Two-Component CBSS Example.

Figure A.2.3 and Figure A.2.4 show the API of Comp-1 and the CII of Comp-1 and Comp-2, respectively. Note that the information provided by both the API and the CII can be scanned by a parser so that the dependency between these two components can be stored in a file or a database. This information can later be used to construct the adjacency matrix, transitive closure matrix, and the CDG for testing purposes.

```
Component-API: Comp-1, V. 1.0
{
  Function-Signatures:
      String F1 (input-list: int x1;),
      Int F2 ( input-list: int x2;
               Output-list: int x3;)
}
```

Figure A.2.3. Comp-1's API

```
Component-interact: Comp-1, V. 1.0
{
  Port: Comp-2, V. 2.0;
  {
    Comp-1:F1 R* Comp2-V2:G1
  }
}
```

Figure A.2.4. CII of Comp-1 and Comp-2.

Appendix B. Supporting Data for the Elevator System

This section is intended to present the complete CIDL of the elevator system.

```
Component-API: Comp-operationaPanel, V. 1.0
{
  Function-Signatures:
     Button1;
     Button2;
     Button3;
     openDoorButton;
     closeDoorButton;
     stopButton;
    dialButton;
    upButton;
    downButton;
    OperPanel;
}
```

Figure B.1. API of the OperationPanel.

```
Component-interact: Comp-OperationPanel, V. 1.0; Comp-Elevator, V. 1.0;
{
  Port: Comp-OperationPanel, V. 1.0; Comp-Elevator, V. 1.0;
  {
   Comp-OperationPanel, V. 1.0 : Button1 R* Comp-Elevator, V. 1.0:
   Boolean moveto(int fromLevel, int toLevel);
  }
}
```

Figure B.2. CII between the Button1 Function of the OperationPanel Component and the MoveTo Function of the Elevator Component.

```
Component-interact: Comp-OperationPanel, V. 1.0; Comp-Elevator, V. 1.0;
{
  Port: Comp-OperationPanel, V. 1.0; Comp-Elevator, V. 1.0;
  {
   Comp-OperationPanel, V. 1.0:Button2 R* Comp-Elevator, V. 1.0:
   Boolean moveto(int fromLevel, int toLevel);
  }
}
```

Figure B.3. CII between the Button2 Function of the OperationPanel Component and the MoveTo Function of the Elevator Component.

```
Component-interact: Comp-OperationPanel, V. 1.0; Comp-Elevator, V. 1.0;
{
  Port: Comp-OperationPanel, V. 1.0; Comp-Elevator, V. 1.0;
  {
   Comp-OperationPanel, V. 1.0:Button3 R* Comp-Elevator, V. 1.0:
   Boolean moveto(int fromLevel, int toLevel);
  }
}
```

Figure B.4. CII between the Button3 Function of the OperationPanel Component and the moveTo Function of the Elevator Component.

```
Component-interact: Comp-OperationPanel, V. 1.0; Comp-Elevator, V. 1.0;
{
  Port: Comp-OperationPanel, V. 1.0; Comp-Elevator, V. 1.0;
  {
   Comp-OperationPanel, V. 1.0:openDoorButton R* CompElevator, V. 1.0:
   Boolean openDoor();
  }
}
```

Figure B.5. CII between the OpenDoorButton Function of the OperationPanel Component and the OpenDoor Function of the Elevator Component.

```
Component-interact: Comp-OperationPanel, V. 1.0; Comp-Elevator, V. 1.0;
{
  Port: Comp-OperationPanel, V. 1.0; Comp-Elevator, V. 1.0;
  {
   Comp-OperationPanel, V. 1.0:closeDoorButton R* CompElevator, V. 1.0:
   Boolean closeDoor();
  }
}
```

Figure B.6. CII between the CloseDoorButton Function of the OperationPanel Component and the CloseDoor Function of the Elevator Component.

```
Component-interact: Comp-OperationPanel, V. 1.0; Comp-Elevator, V. 1.0;
{
  Port: Comp-OperationPanel, V. 1.0; Comp-Elevator, V. 1.0;
  {
   Comp-OperationPanel, V. 1.0:stopButton R* CompElevator, V. 1.0:
   Boolean stopElevator();
  }
}
```

Figure B.7. CII between the StopButton Function of the OperationPanel Component and the StopElevator Function of the Elevator Component.

```
Component-interact: Comp-OperationPanel, V. 1.0; Comp-Elevator, V. 1.0;
{
  Port: Comp-OperationPanel, V. 1.0; Comp-Elevator, V. 1.0;
  {
   Comp-OperationPanel, V. 1.0:dialButton R* CompElevator, V. 1.0:
   Boolean dial();
  }
}
```

Figure B.8. CII between the DialButton Function of The OperationPanel Component and the Dial Function of the Elevator Component.

```
Component-interact: Comp-OperationPanel, V. 1.0; Comp-Elevator, V. 1.0;
{
  Port: Comp-OperationPanel, V. 1.0; Comp-Elevator, V. 1.0;
  {
   Comp-OperationPanel, V. 1.0 :OperationPanel R* CompElevator, V. 1.0:
   void showStatus(String curStatus);
  }
}
```

Figure B.9. CII between the OperationPanel Function of the OperationPanel Component and the ShowStatus Function of the Elevator Component.

```
Component-interact: Comp-Elevator, V. 1.0; Comp-Controller, V. 1.0;
{
  Port: Comp-Elevator, V. 1.0; Comp-Controller, V. 1.0;
  {
   Comp-Elevator, V. 1.0:Boolean moveUp (int fromLevel, int toLevel) R*
   Comp-Controller, V. 1.0: void controlMoveUp(int fromLevel, int toLevel);
  }
}
```

Figure B.10. CII between the moveUp Function of the Elevator Component and the ControlMoveUp Function of the Controller Component.

```
Component-interact: Comp-Elevator, V. 1.0; Comp-Controller, V. 1.0;
{
  Port: Comp-Elevator, V. 1.0; Comp-Controller, V. 1.0;
  {
   Comp-Controller, V. 1.0:Boolean moveDown (int fromLevel, int toLevel) R*
   Comp-Controller, V. 1.0: void controlMoveDown(int fromLevel, int toLevel);
  }
}
```

Figure B.11. CII between the MoveDown Function of the Elevator Component and the ControlMoveDown Function of the Controller Component.

```
Component-API: Comp-floorPanel, V. 1.0
{
  Function-Signatures:
  upButtonL1;
  upButtonL2;
  downButtonL2;
  downButtonL3;
}
```

Figure B.12. API of the FloorPanel.

```
Component-interact: Comp-FloorPanel, V. 1.0; Comp-Controller, V. 1.0;
{
  Port: Comp-FloorPanel, V. 1.0; Comp-Controller, V. 1.0;
  {
   Comp-FloorPanel, V. 1.0: upButtonL1 R* Comp-Controller, V. 1.0:
   void controlMoveUp(int fromLevel, int toLevel);
  }
}
```

Figure B.13. CII between the UpButtonL1 Function of the FloorPanel Component and the ControlMoveUp Function of the Controller Component.

```
Component-interact: Comp-FloorPanel, V. 1.0; Comp-Controller, V. 1.0;
{
  Port: Comp-FloorPanel, V. 1.0; Comp-Controller, V. 1.0;
  {
   Comp-FloorPanel, V. 1.0: upButtonL2 R* Comp-Controller, V. 1.0:
   void controlMoveUp(int fromLevel, int toLevel);
  }
}
```

Figure B.14. CII between the UpButtonL2 Function of the FloorPanel Component and the ControlMoveUp Function of the Controller Component.

```
Component-interact: Comp-FloorPanel, V. 1.0; Comp-Controller, V. 1.0;
{
  Port: Comp-FloorPanel, V. 1.0; Comp-Controller, V. 1.0;
  {
   Comp-FloorPanel, V. 1.0: downButtonL2 R* Comp-Controller, V. 1.0:
   void controlMoveDown(int fromLevel, int toLevel);
  }
}
```

Figure B.15. CII between the DownButtonL2 Function of the FloorPanel Component and the ControlMoveDown Function of the Controller Component.

```
Component-interact: Comp-FloorPanel, V. 1.0; Comp-Controller, V. 1.0;
{
  Port: Comp-FloorPanel, V. 1.0; Comp-Controller, V. 1.0;
  {
   Comp-FloorPanel, V. 1.0: downButtonL3 R* Comp-Controller, V. 1.0:
   void controlMoveDown(int fromLevel, int toLevel);
  }
}
```

Figure B.16. CII between the DownButtonL3 Function of the FloorPanel Component and the ControlMoveDown Function of the Controller Component.

Appendix C. The SPARC Emulator Case Study

C.1 Overview of the SPARC Emulator

The SPARC emulator is a tool that allows SPARC assembly language beginners to explore the SPARC Instruction Set Architecture (ISA), write SPARC assembly code, run the written program, and step through the executing program for debugging purposes. It's a Graphical User Interface (GUI) application that makes it very convenient for users to load any SPARC assembly program for execution. During the execution of an SPARC assembly program, the tool displays all the major SPARC registers and a DEBUG button that makes it possible for users to singly step through the assembly file for debugging purposes. Through the GUI, users can visually examine the contents of the affected registers after any particular instruction has been executed. The GUI displays the computed results using all 4 well-known radices: decimal, binary, hexadecimal, and octal.

The SPARC emulator is a relatively large program. Version 1.0 consists of 17 components, and version 2.0 has 33 components. Because of the large size of the system, the dependency and interaction among the components of this system will be presented using the CIDL instead of CDG. The change impact results will be presented in tabular form.

The main component of the SPARC emulator is the SparcEmulator. It is a GUI component through which users can interact. In other words, users can access all the features provided by the emulator on this GUI. As soon as receiving any user's request, the SparcEmulator would direct that request to an appropriate component for help. Depending on the dependency among the components for a particular service, the just called component may fulfill the user's request itself, or it may send the request to another component for help until the requested service is served.

C.2 CIDL and CDG of the SPARC Emulator, Version 1

In this section, we present the dependency among the components of the SPARC Emulator, version 1, using the CIDL.

```
Component-API: Comp-SparcEmulator, V. 1.0
{
  Function-Signatures:
     openFileButton;
     resetButton;
     setBreakpointMouseClick;
     helpButton;
     registerButton;
     registerTableButton;
     stepButton;
     runButton;
     pauseButton;
}
```

Figure C.2.1. API of the SparcEmulator (the CUT).

```
Component-interact: Comp-SparcEmulator, V. 1.0 ; Comp-LoadFile, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:openFileButton R* Comp-LoadFile, V1:
   StatementHolder[] readSource(String pathname)
  }
}
```

Figure C.2.2. CII between the OpenFileButton of the SparcEmulator Component and the ReadSource Function of the LoadFile Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0 ; Comp-StatementHolder, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:resetButton R* Comp-LoadFile, V1:
   void setCurrentStatement(boolean value);
  }
}
```

Figure C.2.3. CII between the ResetButton of the SparcEmulator Component and the SetCurrentStatement Function of the StatementHolder Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0 ; Comp-SparcData, V. 1.0
{
  Port: Comp-SparcData, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:resetButton R* Comp-SparcData, V1:
   void reset();
  }
}
```

Figure C.2.4. CII between the ResetButton of the SparcEmulator Component and the Reset Function of the SparcData Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-Addxx, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-Addxx, V. 1.0:
   void addNorm (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.5. CII between the StepButton Function of the SparcEmulator Component and the AddNorm Function of the Addxx Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-Addxx, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-Addxx, V. 1.0:
   void addx (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.6. CII between the StepButton Function of the SparcEmulator Component and the Addx Function of the Addxx Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-Addxx, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-Addxx, V. 1.0:
   void addxcc (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.7. CII between the StepButton Function of the SparcEmulator Component and the Addxcc Function of the Addxx Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-Addxx, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-Addxx, V. 1.0:
   void addNormcc (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.8. CII between the StepButton Function of the SparcEmulator Component and the AddNormcc Function of the Addxx Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-SubtractX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-SubtractX, V. 1.0:
   void sub (String rs1, String rs2, String rd)
  }
}
```

Figure C.2.9. CII between the StepButton Function of the SparcEmulator Component and the Sub Function of the SubtractX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-SubtractX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-SubtractX, V. 1.0:
   void subx (String rs1, String rs2, String rd)
  }
}
```

Figure C.2.10. CII between the StepButton Function of the SparcEmulator Component and the Subx Function of the SubtractX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-SubtractX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-SubtractX, V. 1.0:
   void subxcc (String rs1, String rs2, String rd)
  }
}
```

Figure C.2.11. CII between the StepButton Function of the SparcEmulator Component and the Subxcc Function of the SubtractX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-SubtractX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-SubtractX, V. 1.0:
   void subcc (String rs1, String rs2, String rd)
  }
}
```

Figure C.2.12. CII between the StepButton Function of the SparcEmulator Component and the Subcc Function of the SubtractX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-MultX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-MultX, V. 1.0:
   void umul (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.13. CII between the StepButton Function of the SparcEmulator Component and the Umul Function of the MultX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-multX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-multX, V. 1.0:
   void umulcc (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.14. CII between the StepButton Function of the SparcEmulator Component and the Umulcc Function of the MultX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-multX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-multX, V. 1.0:
   void smul (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.15. CII between the StepButton Function of the SparcEmulator Component and the Smul Function of the MultX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-MultX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-MultX, V. 1.0:
   void smulcc (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.16. CII between the StepButton Function of the SparcEmulator Component and the Smulcc Function of the MultX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-DivX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-DivX, V. 1.0:
   void divSign (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.17. CII between the StepButton Function of the SparcEmulator Component and the DivSign Function of the DivX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-DivX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-DivX, V. 1.0:
   void divUsign (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.18. CII between the StepButton Function of the SparcEmulator Component and the DivUsign Function of the DivX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-DivX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-DivX, V. 1.0:
   void divSigncc (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.19. CII between the StepButton Function of the SparcEmulator Component and the DivSigncc Function of the DivX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-DivX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-DivX, V. 1.0:
   void divUsigncc (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.20. CII between the StepButton Function of the SparcEmulator Component and the DivUsigncc Function of the DivX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-AndX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-AndX, V. 1.0:
   void andNorm (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.21. CII between the StepButton Function of the SparcEmulator Component and the Andnorm Function of the AndX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-AndX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-AndX, V. 1.0:
   void andNormcc (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.22. CII between the StepButton Function of the SparcEmulator Component and the AndNormcc Function of the AndX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-AndX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-AndX, V. 1.0:
   void andN (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.23. CII between the StepButton Function of the SparcEmulator Component and the AndN Function of the AndX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-ORX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-ORX, V. 1.0:
   void ORnorm (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.24. CII between the StepButton Function of the SparcEmulator Component and the ORnorm Function of the AndX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp- ORX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp- ORX, V. 1.0:
   void ORNormcc (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.25. CII between the StepButton Function of the SparcEmulator Component and the ORnormcc Function of the ORX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp- ORX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp- ORX, V. 1.0:
   void ORcc (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.26. CII between the StepButton Function of the SparcEmulator Component and the ORcc Function of the ORX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-ORX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-ORX, V. 1.0:
   void ORN (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.27. CII between the StepButton Function of the SparcEmulator Component and the ORN Function of the AndX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-ORX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-ORX, V. 1.0:
   void ORNcc (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.28. CII between the StepButton Function of the SparcEmulator Component and the ORNcc Function of the AndX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-XORX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-ORX, V. 1.0:
   void XOR (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.29. CII between the StepButton Function of the SparcEmulator Component and the XOR Function of the XORX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-XORX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-ORX, V. 1.0:
   void XORcc (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.30. CII between the StepButton Function of the SparcEmulator Component and the XORcc Function of the XORX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-XNORX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-XNORX, V. 1.0:
   void XNOR (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.31. CII between the StepButton Function of the SparcEmulator Component and the XNOR Function of the XNORX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-XNORX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-XNORX, V. 1.0:
   void XNORcc (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.32. CII between the StepButton Function of the SparcEmulator Component and the XNORcc Function of the XNORX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-ShiftX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-ShiftX, V. 1.0:
   void SLL (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.33. CII between the StepButton Function of the SparcEmulator Component and the SLL Function of the ShiftX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-ShiftX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-ShiftX, V. 1.0:
   void SRL (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.34. CII between the StepButton Function of the SparcEmulator Component and the SRL Function of the ShiftX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0; Comp-ShiftX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0;
  {
   Comp-SparcEmulator, V. 1.0:stepButton R* Comp-ShiftX, V. 1.0:
   void SRA (String rs1, String rs2, String rd);
  }
}
```

Figure C.2.35. CII between the StepButton Function of the SparcEmulator Component and the SRA Function of the ShiftX Component.

The dependency of the runButton on the other components is the same as the dependency between the stepButton and the other components. So, we will present the dependency between the runButton and the other components in tabular form as shown in Table C.2.1

Component under Study	Directly Dependent on
SparcEmulator.runButton	Addxx.addnorm (String rs1, String rs2, String rd);
	Addxx.addx (String rs1, String rs2, String rd);
	Addxx.addxcc (String rs1, String rs2, String rd);
	SubtractX.sub (String rs1, String rs2, String rd)
	SubtractX.subx (String rs1, String rs2, String rd)
	SubtractX.subxcc (String rs1, String rs2, String rd)
	SubtractX.subcc (String rs1, String rs2, String rd)
	MultX.umul (String rs1, String rs2, String rd);
	MultX.umulcc (String rs1, String rs2, String rd);
	MultX.smul (String rs1, String rs2, String rd);
	MultX.smulcc (String rs1, String rs2, String rd);
	DivX.divSign (String rs1, String rs2, String rd);
	DivX.divUSign (String rs1, String rs2, String rd);
	DivX.divSigncc (String rs1, String rs2, String rd);
	DivX.divUSigncc (String rs1, String rs2, String rd);
	AndX.andNorm (String rs1, String rs2, String rd);
	AndX.andN (String rs1, String rs2, String rd);
	AndX.andNormcc (String rs1, String rs2, String rd);
	AndX.andNcc (String rs1, String rs2, String rd);
	ORX.ORnorm (String rs1, String rs2, String rd);
	ORX.ORnormcc (String rs1, String rs2, String rd);
	ORX.ORN (String rs1, String rs2, String rd);
	ORX.ORNcc (String rs1, String rs2, String rd);
	XORX.NOR (String rs1, String rs2, String rd);
	XORX.NORcc (String rs1, String rs2, String rd);
	ShiftX.SLL (String rs1, String rs2, String rd);
	ShiftX.SRL (String rs1, String rs2, String rd);
	ShiftX.SRA (String rs1, String rs2, String rd);

Table C.2.1. Direct Dependency between the SparEmulator.runButton and Other Components in the SPARC Emulator System, Version 1.

The stepButton and runButton of the SparEmulator component also depend on the functions of the StatementHolder component. Table C.2.2 shows their relationship.

Component under Study	Directly Dependent on
SparcEmulator.stepButton	StatementHolder.String getLabel()
	StatementHolder.int getAddress()
or	StatementHolder.String getInstruction()
	StatementHolder.String getParam(int paramNum)
SparcEmulator.runButton	StatementHolder.String getErrorMessage()
	StatementHolder.Boolean setIsInvalid()
	StatementHolder.boolean isComment()
	StatementHolder.setIsComment(boolean value)
	StatementHolder.void setIsInvalid(boolean value)
	StatementHolder.void setCurrentStatement(boolean value)
	StatementHolder.boolean isCurrentStatement()

Table C.2.2. Dependency of the StepButton or RunButton on the Functions of the StatementHolder Component.

```
Component-interact: Comp-LoadFile, V. 1.0; Comp-StatementHolder, V. 1.0;
{
  Port: Comp-LoadFile, V. 1.0; Comp-StatementHolder, V. 1.0;
  {
   Comp-LoadFile, V. 1.0:StatementHolder[] readSource(String pathname) R*
   Comp-StatementHolder, V. 1.0: StatementHolder(String str);
  }
}
```

Figure C.2.36. CII between the LoadFile Function of the LoadFile Component and the StatementHolder Function of the StatementHolder Component.

```
Component-interact: Comp-LoadFile, V. 1.0; Comp-StatementHolder, V. 1.0;
{
  Port: Comp-LoadFile, V. 1.0; Comp-StatementHolder, V. 1.0;
  {
   Comp-LoadFile, V. 1.0:StatementHolder[] readSource(String pathname) R*
   Comp-StatementHolder, V. 1.0: StatementHolder(String str1, String str2);
  }
}
```

Figure C.2.37. CII between the LoadFile Function of the LoadFile Component and the StatementHolder Function of the StatementHolder Component.

```
Component-interact: Comp-LoadFile, V. 1.0; Comp-StatementHolder, V. 1.0;
{
  Port: Comp-LoadFile, V. 1.0; Comp-StatementHolder, V. 1.0;
  {
   Comp-LoadFile, V. 1.0:StatementHolder[] readSource(String pathname) R*
   Comp-StatementHolder, V. 1.0:
   StatementHolder(String str1, String str2, String str3);
  }
}
```

Figure C.2.38. CII between the ReadSource Function of the LoadFile Component and the StatementHolder Function of the StatementHolder Component.

```
Component-interact: Comp-LoadFile, V. 1.0; Comp-StatementHolder, V. 1.0;
{
  Port: Comp-LoadFile, V. 1.0; Comp-StatementHolder, V. 1.0;
  {
   Comp-LoadFile, V. 1.0:StatementHolder[] readSource(String pathname) R*
   Comp-StatementHolder, V. 1.0:
   StatementHolder(String str1, String str2, String str3, String str4);
  }
}
```

Figure C.2.39. CII between the ReadSource Function of the LoadFile Component and the StatementHolder Function of the StatementHolder Component.

```
Component-interact: Comp-LoadFile, V. 1.0; Comp-StatementHolder, V. 1.0;
{
  Port: Comp-LoadFile, V. 1.0; Comp-StatementHolder, V. 1.0;
  {
   Comp-LoadFile, V. 1.0:StatementHolder[] readSource(String pathname) R*
   Comp-StatementHolder, V. 1.0:
   StatementHolder(String label, String str1, String str2, String str3, String str4);
  }
}
```

Figure C.2.40. CII between the ReadSource Function of the LoadFile Component and the StatementHolder Function of the StatementHolder Component.

```
Component-interact: Comp-LoadFile, V. 1.0; Comp-StatementHolder, V. 1.0;
{
  Port: Comp-LoadFile, V. 1.0; Comp-StatementHolder, V. 1.0;
  {
   Comp-LoadFile, V. 1.0:StatementHolder[] readSource(String pathname) R*
   Comp-StatementHolder, V. 1.0:
   void setIsLabel (String value);
  }
}
```

Figure C.2.41. CII between the LoadFile Function of the LoadFile Component and the SetIsLabel Function of the StatementHolder Component.

```
Component-interact: Comp-LoadFile, V. 1.0; Comp-StatementHolder, V. 1.0;
{
  Port: Comp-LoadFile, V. 1.0; Comp-StatementHolder, V. 1.0;
  {
   Comp-LoadFile, V. 1.0:StatementHolder[] readSource(String pathname) R*
   Comp-StatementHolder, V. 1.0:
   void setIsComment (Boolean value);
  }
}
```

Figure C.2.41. CII between the ReadSource Function of the LoadFile Component and the SetIsComment Function of the StatementHolder Component.

```
Component-interact: Comp-LoadFile, V. 1.0; Comp-StatementHolder, V. 1.0;
{
  Port: Comp-LoadFile, V. 1.0; Comp-StatementHolder, V. 1.0;
  {
   Comp-LoadFile, V. 1.0:StatementHolder[] readSource(String pathname) R*
   Comp-StatementHolder, V. 1.0:
   void setIsInvalid (Boolean value);
  }
}
```

Figure C.2.42. CII between the ReadSource Function of the LoadFile Component and the SetIsInvalid Function of the StatementHolder Component.

```
Component-interact: Comp-LoadFile, V. 1.0; Comp-StatementHolder, V. 1.0;
{
  Port: Comp-LoadFile, V. 1.0; Comp-StatementHolder, V. 1.0;
  {
   Comp-LoadFile, V. 1.0:StatementHolder[] readSource(String pathname) R*
   Comp-StatementHolder, V. 1.0:
   void setIsInvalid (Boolean value);
  }
}
```

Figure C.2.43. CII between the ReadSource Function of the LoadFile Component and the SetIsInvalid Function of the StatementHolder Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0 ; Comp-StatementHolder, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0 ; Comp-StatementHolder, V. 1.0
  {
   Comp-SparcEmulator, V. 1.0: setBreakpointMouseClick R*
   Comp-StatementHolder, V1: void setBreakPoint(Boolean val);
  }
}
```

Figure C.2.44. CII between the SetBreakpointMouseClick of the SparcEmulator Component and the SetBreakPoint Function of the Comp-StatementHolder Component.

The dependency between any SPARC mnemonic function and the other components of the SPARC emulator system is similar. Figure C.2.45 and C.2.46 show the direct dependency between the addcc function of the Addxx component and the functions of the SparcData component. The dependency between any other SPARC mnemonic function (except for the branch instructions) and the functions of other components is shown in Table C.2.3.

```
Component-interact: Comp-Addxx, V. 1.0 ; Comp-SparcData, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0 ; Comp-StatementHolder, V. 1.0
  {
   Comp-addcc, V. 1.0: void addx (String rs1, String rs2, String rd) R*
   Comp-SparcData, V1: void setRegisterValue(String regName, int val);
  }
}
```

Figure C.2.45. CII between the Addcc Function of the Addxx Component and the SetRegisterValue Function of the Comp-SparcData Component.

```
Component-interact: Comp-Addxx, V. 1.0 ; Comp-SparcData, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0 ; Comp-StatementHolder, V. 1.0
  {
   Comp-addcc, V. 1.0: void addx (String rs1, String rs2, String rd) R*
   Comp-SparcData, V1: long setRegisterValue(String register);
  }
}
```

Figure C.2.46. CII between the Addcc Function of the Addxx Component and the SetRegisterValue Function of the Comp-SparcData Component.

Dependant	Supporters of the SparcData component

Addxx.addnorm (String rs1, String rs2, String rd);	void setRegisterValue(String regName, int val);
	
	or
	
	long getRegisterValue(String register);

Addxx.addx (String rs1, String rs2, String rd);	
Addxx.addxcc (String rs1, String rs2, String rd);	
SubtractX.sub (String rs1, String rs2, String rd)	
SubtractX.subx (String rs1, String rs2, String rd)	
SubtractX.subxcc (String rs1, String rs2, String rd)	
SubtractX.subcc (String rs1, String rs2, String rd)	
MultX.umul (String rs1, String rs2, String rd);	
MultX.umulcc (String rs1, String rs2, String rd);	
MultX.smul (String rs1, String rs2, String rd);	
MultX.smulcc (String rs1, String rs2, String rd);	
DivX.divSign (String rs1, String rs2, String rd);	
DivX.divUSign (String rs1, String rs2, String rd);	
DivX.divSigncc (String rs1, String rs2, String rd);	
DivX.divUSigncc (String rs1, String rs2, String rd);	
AndX.andNorm (String rs1, String rs2, String rd);	
AndX.andN (String rs1, String rs2, String rd);	
AndX.andNormcc (String rs1, String rs2, String rd);	
AndX.andNcc (String rs1, String rs2, String rd);	
ORX.ORnorm (String rs1, String rs2, String rd);	
ORX.ORnormcc (String rs1, String rs2, String rd);	
ORX.ORN (String rs1, String rs2, String rd);	
ORX.ORNcc (String rs1, String rs2, String rd);	
XORX.NOR (String rs1, String rs2, String rd);	
XORX.NORcc (String rs1, String rs2, String rd);	
ShiftX.SLL (String rs1, String rs2, String rd);	
ShiftX.SRL (String rs1, String rs2, String rd);	
ShiftX.SRA (String rs1, String rs2, String rd);	

Table C.2.3. Dependency between a Mnemonic Function And the Functions of the SparcData Component.

Figure C.2.47 and C.2.48 show the CII for the BA function of the BranchX component. The dependency between the runButton/resetButton and the other branch functions are displayed in Table C.2.4.

```
Component-interact: Comp-SparcEmulator, V. 1.0 ; Comp-BranchX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0 ; Comp-BranchX, V. 1.0
  {
   Comp-SparcEmulator, V. 1.0: stepButton R*
   Comp-BranchX, V1: void BA(String addr);
  }
}
```

Figure C.2.47. CII between the StepButton of the SparcEmulator Component and the BA Function of the Comp-BranchX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0 ; Comp-BranchX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0 ; Comp-BranchX, V. 1.0
  {
   Comp-SparcEmulator, V. 1.0: runButton R*
   Comp-BranchX, V1: void BA(String addr);
  }
}
```

Figure C.2.48. CII between the RunButton of the SparcEmulator Component and the BA Function of the Comp-BranchX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0 ; Comp-BranchX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0 ; Comp-BranchX, V. 1.0
  {
   Comp-SparcEmulator, V. 1.0: runButton R*
   Comp-BranchX, V1: void BN(String addr);
  }
}
```

Figure C.2.49. CII between the RunButton of the SparcEmulator Component and the BN Function of the Comp-BranchX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0 ; Comp-BranchX, V. 1.0
{
  Port: Comp-SparcEmulator, V. 1.0 ; Comp-BranchX, V. 1.0
  {
   Comp-SparcEmulator, V. 1.0: stepButton R*
   Comp-BranchX, V1: void BN(String addr);
  }
}
```

Figure C.2.49. CII between the StepButton of the SparcEmulator Component and the BN Function of the Comp-BranchX Component.

Component	Component's functions
BranchX	void BA (String label);
	void BN (String label);
	void BNE (String label);
	void BE (String label);
	void BG (String label);
	void BLE (String label);
	void BGE (String label);
	void BL (String label);
	void BGU (String label);
	void BL (String label);
	void BGU (String label);
	void BLEU (String label);
	void BCC (String label);
	void BCS (String label);
	void BPOS (String label);
	void BNEG (String label);
	void BVC (String label);
	void BVS (String label);

Table C.2.4. The Branch Component.

The getRegisterValue and the setRegisterValue functions of the SparcData component are also dependent on some functions of the RegisterHolder component. Their relationship is shown in Figures C.2.50 and C.2.51.

```
Component-interact: Comp-SparcData, V. 1.0 ; Comp-RegisterHolder, V. 1.0
{
  Port: Comp-SparcData, V. 1.0 ; Comp-RegisterHolder, V. 1.0
  {
   Comp-SparcData, V1: long setRegisterValue(String register) R*
   Comp-RegisterHoler, V2: void setValue(long value);
  }
}
```

Figure C.2.50. CII between the SetRegisterValue Function of the SparcData Component and the SetValue Function of the Comp-RegisterHolder Component.

```
Component-interact: Comp-SparcData, V. 1.0 ; Comp-RegisterHolder, V. 1.0
{
  Port: Comp-SparcData, V. 1.0 ; Comp-RegisterHolder, V. 1.0
  {
   Comp-SparcData, V1: long getRegisterValue(String register) R*
   Comp-RegisterHoler, V2: long getValue();
  }
}
```

Figure C.2.51. CII between the GetRegisterValue Function of the SparcData Component and the GetValue Function of the Comp-RegisterHolder Component.

As mentioned earlier, it's impossible to show the CDG of the whole system in this context because of the large size of the Sparc emulator CBSS. Instead, the two critical paths of the CDG are shown in Figure C.2.52 and Figure C.2.53 to demonstrate the function-function dependency among the components of the Sparc Emulator. These two paths are carefully selected to provide readers with the core structure of the Sparc Emulator.

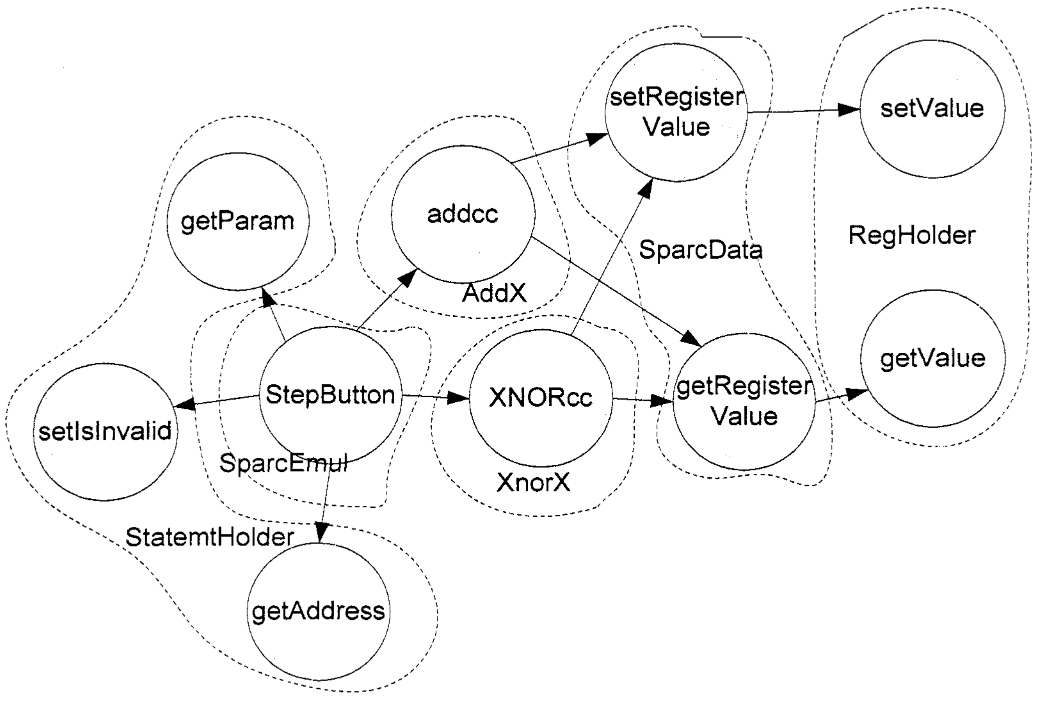

Figure C.2.52. Partial CDG of the Sparc Emulator Showing the Execution Path When the StepButton Is Pressed.

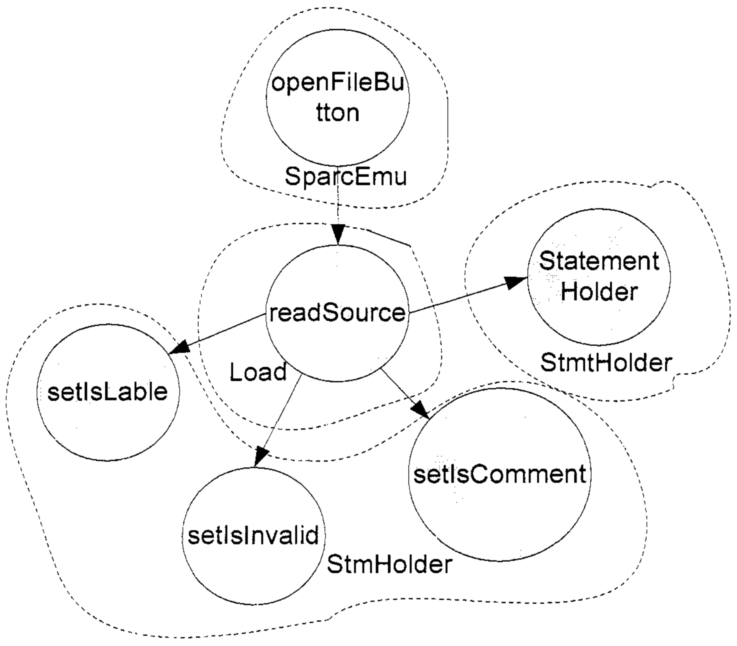

Figure C.2.53. Partial CDG of the CBSS Showing the Execution Path of the openFilebutton

C.3 CIDL Descriptions of the SPARC Emulator, Version 2

In version 2, a number of components are added to the SPARC Emulator in addition to the modified components. To save space, only part of the changes will be shown using CIDL. The remainder will be displayed in tabular form.

C.3.1 Added and Modified Components

The new components introduced into version 2 of the SPARC emulator are shown in Table C.3.1.

Components added to the CBSS	Component's functions
TestX	void testReg(String register)
SetX	void setReg(String register, long value)
NotX	void notNegate(String rs, String rd)
ClearX	clearReg(String rd)
MovX	void mov(String rs, String rd)
Call	void call (String addr)
JMPL	void JmpL(String addr, String rd)
Save	void save(String rs1, String rs2, String rd)
Restore	void restore(String rs1, String rs2, String rd)
SyntaxChecker	Boolean isInstruction()

Table C.3.1. Components Added to Version 2.

The runButton and stepButton of the SparcEmulator depend on the functions of these new components for a particular task. We will next show the CII of the new components.

```
Component-interact: Comp-SparcEmulator, V. 1.0 ; Comp-TestX, V. 2.0
{  Port: Comp-SparcEmulator, V. 1.0 ; Comp-RegisterHolder, V. 2.0
   {
    Comp-SparcEmulator, V1: stepButton (String register) R*
    Comp-TestX, V2: void TST(String register);
   }
}
```

Figure C.3.1. CII between the StepButton Function of the SparcEmulator Component and the TST Function of the Comp-TestX Component.

```
Component-interact: Comp-SparcEmulator, V. 1.0 ; Comp-TestX, V. 2.0
{
  Port: Comp-SparcEmulator, V. 1.0 ; Comp-RegisterHolder, V. 1.0
   {
    Comp-SparcEmulator, V1: runButton (String register) R*
    Comp-TestX, V2: void TST(String register);
   }
}
```

Figure C.3.2. CII between the RunButton Function of the SparcEmulator Component and the TST Function of the Comp-TestX Component.

The dependency between the stepButton/runButton and the functions of any new component is shown in Table C.3.2.

Components under Study	Directly dependent on
SparcEmulator.stepButton	TestX.void testReg(String register)
or	SetX.void setReg(String register, long value)
SparcEmulator.runButton	NotX.void notNegate(String rs, String rd)
	ClearX.clearReg(String rd)
	MovX.void mov(String rs, String rd)
	Call.void call (String addr)
	JMPL.void JmpL(String addr, String rd)
	Save.void save(String rs1, String rs2, String rd)
	Restore.void restore(String rs1, String rs2, String rd)

Table C.3.2. Dependency between the StepButton/RunButton of the SparcEmulator Component and the Functions of the New Components.

The relationship between each function of the added components and the functions of the old components is shown in Figure C.3.3, Figure C.3.4, and Table C.3.3.

```
Component-interact: Comp-TestX, V. 2.0 ; Comp-SparcData, V.1.0
{
  Port: Comp-TestX, V. 2.0 ; Comp-SparcData, V.1.0
  {
   Comp-TestX, V. 2.0 : void testReg(String register) R*
   Comp-SparcData, V. 1.0 : long getRegisterValue(String register);
  }
}
```

Figure C.3.3. CII between the TestReg Function of the TestX Component and the GetRegisterValue Function of the SparcData Component.

```
Component-interact: Comp-TestX, V. 2.0 ; Comp-SparcData, V.1.0
{
  Port: Comp-TestX, V. 2.0 ; Comp-SparcData, V.1.0
  {
   Comp-TestX, V. 2.0 : void testReg(String register) R*
   Comp-SparcData, V. 1.0 : void setRegisterValue(String register, int value);
  }
}
```

Figure C.3.4. CII between the TestReg Function of the TestX Component and the SetRegisterValue Function of the SparcData Component.

Dependant (version 2)	Supporters of the RegisterHolder component
TestX.void testReg(String register)	void setRegisterValue(String regName, int val); or long getRegisterValue(String register);
SetX.void setReg(String register, long value)	
NotX.void notNegate(String rs, String rd)	
ClearX.clearReg(String rd)	
MovX.void mov(String rs, String rd)	
Call.void call (String addr)	
JMPL.void JmpL(String addr, String rd)	
Save.void save(String rs1, String rs2, String rd)	
Restore.void restore(String rs1, String rs2, String rd)	

Table C.3.3. Dependency between the Added Components and the Old Components

The introduction of the SyntaxChecker component into the SPARC emulator causes the changes shown in Figure C.3.5 and Figure C.3.6.

```
Component-interact: Comp-LoadFile, V. 1.0; Comp-SyntaxChecker, V. 2.0;
{
  Port: Comp-LoadFile, V. 1.0; Comp- SyntaxChecker, V. 2.0;
  {
   Comp-LoadFile, V. 1.0:StatementHolder[] readSource(String pathname) R*
   Comp-SyntaxChecker, V. 2.0: Boolean isInstruction();
  }
}
```

Figure C.3.5. CII between the ReadSource Function of the LoadFile Component and the SetIsInvalid Function of the StatementHolder Component.

```
Component-interact: Comp-SyntaxChecker, V. 2.0; Comp-StatementHolder, V. 1.0;
{
  Port: Comp-SyntaxChecker, V. 2.0; Comp-StatementHolder, V. 1.0;
  {
   Comp-SyntaxChecker, V. 2.0 : Boolean isInstruction()  R*
   Comp-StatementHolder, V. 2.0: StatementHolder(String str1);
  }
}
```

Figure C.3.5. CII between the ReadSource Function of the LoadFile Component and the SetIsInvalid Function of the StatementHolder Component.

From the specifications, we know that all the functions of the Addx and subtractX components incorrectly set the value of any register with an integer type (i.e., 32 bits) instead of a long type (i.e., 64 bits). Because of that change, all the functions that have the transitive closure relationship with the addxxx and subxxx functions are affected.

C.4 Change Impact Analysis

Once again, since the SPARC emulator is a relatively large system, we are unable to show the impact of the changes on the entire system using a CDG. Figure C.4.1 and Figure C.4.2 are carefully selected to demonstrate the change impact in two critical paths of the CBSS. Figure C.4.1 shows the impact on part of the system CDG because of the addition of the isInstruction function.

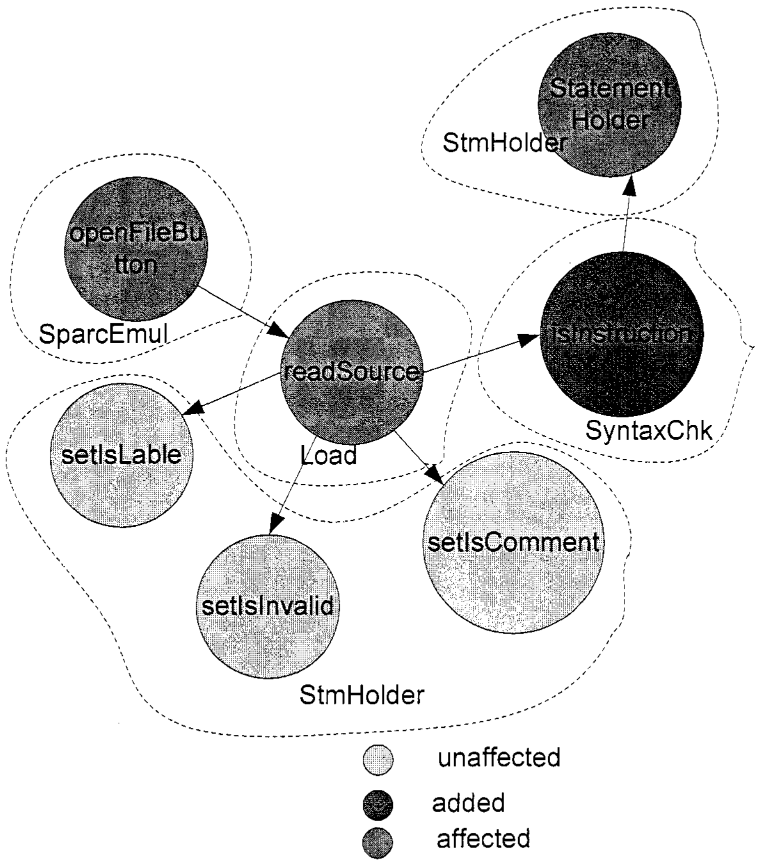

Figure C.4.1. Impact on Part of the System CDG due to the Added IsInstruction Function

Figure C.4.2 shows part of the system CDG caused by the modification of the addcc function.

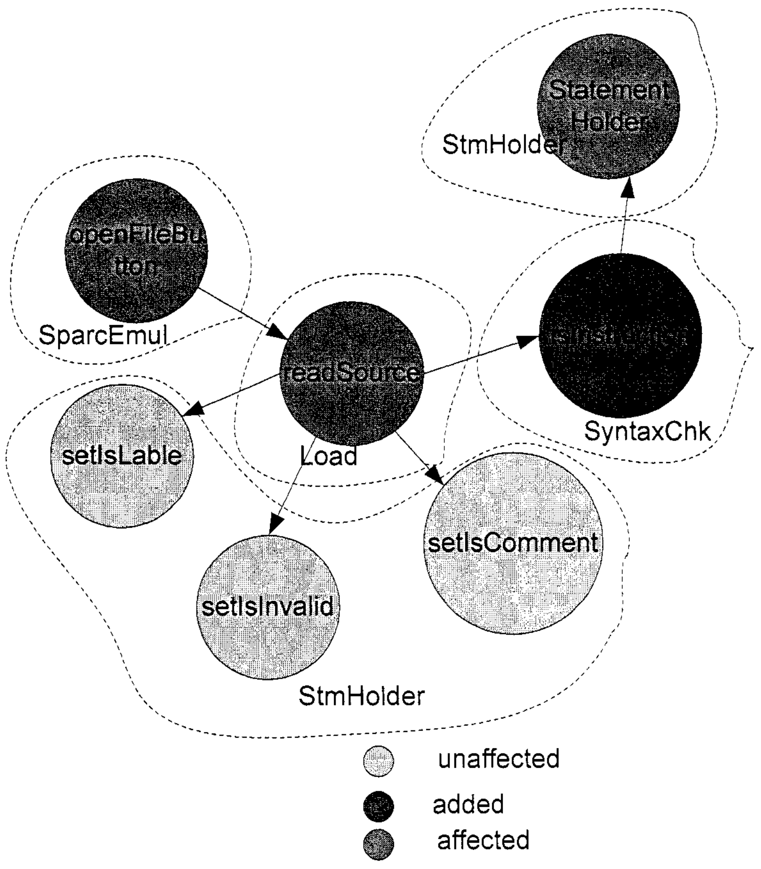

Figure C.4.2. Impact on Part of the System CDG due to the Modified Addcc Function

Figure C.4.3 shows the change impact analysis due to the modified addcc function of the Addx component. The effect of the other modified "add" functions of the Addx component on the system is similar to that of the addcc function and is omitted for the sake of simplicity.

Also, note that the change impact of all the modified "subtract" functions of the SubX component can be explained in a similar fashion to that of the modified addcc function. Figure C.4.4 shows the impact of the modified subcc function of the SubX component.

Added Component	Modified Function	Affected Nodes	Reused Test cases	Obsolete Test cases	New Test cases
AddX	addcc	SparcEmulator-> stepButton, SparcEmulator-> runButton, SparcData-> getRegisterValue, SparcData-> setRegisterValue, RegisterHolder-> getValue ResigterHolder-> setValue	Nodes that don't have component test firewall With the addcc node.	None.	None.

Figure C.4.3. Change Impact due to the Modified Addcc Function of the AddX Component.

Added Component	Modified Function	Affected Nodes	Reused Test cases	Obsolete Test cases	New Test cases
SubX	subcc	SparcEmulator-> stepButton, SparcEmulator-> runButton, SparcData-> getRegisterValue, SparcData-> setRegisterValue, RegisterHolder-> getValue ResigterHolder-> setValue	Nodes that don't have component test firewall With the addcc node.	None.	None.

Figure C.4.4. Change Impact due to the Modified Subcc Function of the SubX Component.

Added Component	Added Function	Affected Nodes	Reused Test cases	Obsolete Test cases	New Test cases
SettX	setReg	SparcEmulator-> stepButton, SparcEmulator-> runButton, SparcData-> getRegisterValue, SparcData-> setRegisterValue, RegisterHolder-> getValue ResigterHolder-> setValue	Nodes that don't have component test firewall With testReg	None.	Component test firewall for all nodes having transitive closure relationship with the setx node (shown in the Affected Nodes column).

Figure C.4.5. Change Impact due to the New SetXComponent.

Added Component	Added Function	Affected Nodes	Reused Test cases	Obsolete Test cases	New Test cases
TestX	testReg	SparcEmulator-> stepButton, SparcEmulator-> runButton, SparcData-> getRegisterValue, SparcData-> setRegisterValue, RegisterHolder-> getValue ResigterHolder-> setValue	Nodes that don't have component test firewall With testReg	None.	Component test firewall for all nodes having transitive closure relationship with the Tst node (shown in the Affected Nodes column).

Figure C.4.6. Change Impact due to the New TestX Component.

Added Component	Added Function	Affected Nodes	Reused Test cases	Obsolete Test cases	New Test cases
SettX	setReg	SparcEmulator-> stepButton, SparcEmulator-> runButton, SparcData-> getRegisterValue, SparcData-> setRegisterValue, RegisterHolder-> getValue ResigterHolder-> setValue	Nodes that don't have component test firewall With testReg	None.	Component test firewall for all nodes having transitive closure relationship with the setx node (shown in the Affected Nodes column).

Figure C.4.7. Change Impact due to the New SetX Component.

Added Component	Added Function	Affected Nodes	Reused Test cases	Obsolete Test cases	New Test cases
NotX	NotNegate	SparcEmulator-> stepButton, SparcEmulator-> runButton, SparcData-> getRegisterValue, SparcData-> setRegisterValue, RegisterHolder-> getValue ResigterHolder-> setValue	Nodes that don't have component test firewall With NotNegate	None.	Component test firewall for all nodes having transitive closure relationship with the setx node (shown in the Affected Nodes column).

Figure C.4.8. Change Impact due to the New NotX Component.

Added Component	Added Function	Affected Nodes	Reused Test cases	Obsolete Test cases	New Test cases
ClearX	clearReg	SparcEmulator-> stepButton, SparcEmulator-> runButton, SparcData-> getRegisterValue, SparcData-> setRegisterValue, RegisterHolder-> getValue ResigterHolder-> setValue	Nodes that don't have component test firewall With clearReg	None.	Component test firewall for all nodes having transitive closure relationship with the setx node (shown in the Affected Nodes column).

Figure C.4.9. Change Impact due to the New ClearX Component.

Added Component	Added Function	Affected Nodes	Reused Test cases	Obsolete Test cases	New Test cases
MovX	mov	SparcEmulator-> stepButton, SparcEmulator-> runButton, SparcData-> getRegisterValue, SparcData-> setRegisterValue, RegisterHolder-> getValue ResigterHolder-> setValue	Nodes that don't have component test firewall With mov	None.	Component test firewall for all nodes having transitive closure relationship with the setx node (shown in the Affected Nodes column).

Figure C.4.10. Change Impact due to the New MoveX Component.

Added Component	Added Function	Affected Nodes	Reused Test cases	Obsolete Test cases	New Test cases
Call	call	SparcEmulator-> stepButton, SparcEmulator-> runButton, SparcData-> getRegisterValue, SparcData-> setRegisterValue, RegisterHolder-> getValue ResigterHolder-> setValue	Nodes that don't have component test firewall With call	None.	Component test firewall for all nodes having transitive closure relationship with the setx node (shown in the Affected Nodes column).

Figure C.4.11. Change Impact due to the New Call Component.

Added Component	Added Function	Affected Nodes	Reused Test cases	Obsolete Test cases	New Test cases
JMPL	jmpL	SparcEmulator-> stepButton, SparcEmulator-> runButton, SparcData-> getRegisterValue, SparcData-> setRegisterValue, RegisterHolder-> getValue ResigterHolder-> setValue	Nodes that don't have component test firewall With jmpL	None.	Component test firewall for all nodes having transitive closure relationship with the setx node (shown in the Affected Nodes column).

Figure C.4.12. Change Impact due to the New JMPL Component.

Added Component	Added Function	Affected Nodes	Reused Test cases	Obsolete Test cases	New Test cases
Save	save	SparcEmulator-> stepButton, SparcEmulator-> runButton, SparcData-> Save(),	Nodes that don't have component test firewall With node save	None.	Component test firewall for all nodes having transitive closure relationship with the setx node (shown in the Affected Nodes column).

Figure C.4.13. Change Impact due to the New Save Component.

Added Component	Added Function	Affected Nodes	Reused Test cases	Obsolete Test cases	New Test cases
Restore	NotNegate	SparcEmulator-> stepButton, SparcEmulator-> runButton, SparcData-> Save()	Nodes that don't have component test firewall With node notNegate	None.	Component test firewall for all nodes having transitive closure relationship with the setx node (shown in the Affected Nodes column).

Figure C.4.14. Change Impact due to the New NotX Component.

Added Component	Added Function	Affected Nodes	Reused Test cases	Obsolete Test cases	New Test cases
SyntaxChecker	isInstruction	SparcEmulator-> openFileButton, LoadFile-> readSource, SyntaxChecker-> isInstruction, StatementHolder -> StatementHolder	T.openFile->readSource All other nodes with component test firewall	T.readSource-StatementHolder, T.openFileBut -> readSource -> StatementHolder.	T.IsInstruction-> StatementHolder, T.readSource-> IsInstruction-> StatementHolder, T.openFileButton readSource-> IsInstruction-> StatementHolder,

Figure C.4.15. Change Impact due to the New JMPL Component.