MODELING THE IMPACT OF CHANGE ON SOFTWARE INTENSIVE SYSTEMS

By

Lian Wen

B.Sc. (Peking University),

M.Eng. (Chinese Academy of Space Technology)

A Dissertation Submitted in Fulfillment of

The Requirement for the Degree of

Doctor of Philosophy

School of Computer and Information Technology

Faculty of Engineering and Information Technology

Griffith University, Australia

October, 2006

To My Father,

Your love is living in my heart, forever.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, Professor Geoff Dromey. He has continuously led me, supported me and encouraged me in the last five years, so that I can finish this thesis, which is my greatest achievement so far. What I have learned from him and through him will be the most valuable treasure in the rest of my life.

Special thanks go to Dr. Chuk You, without his introduction and encouragement, it is impossible for me to have this opportunity to undertake this PhD study.

Appreciation is also extended to my school mates and the members of the DCCS group, which include: Herman Chan, Xueling Zheng, Saad Zafar, Dr. Lars Grunske, Dr. Kirsten Winter, Ms. Nisansala Yatapanage, Dr. Robert Colvin, Professor Ian Hayes and Professor Peter Lindsay. I have benefited greatly from the discussion with them. Gratitude also goes to Yanyan and Xiulan who helped me to proofread a few chapters in this thesis.

Finally, I would like to thank my wife Mary, my sons, Jeremy and Bill and other family members for their support and understanding throughout the last five years.

Abstract

Most significant software-intensive systems undergo substantive change/evolution during their life time of service. Managing the consequent software changes is a difficult and costly task. In this thesis, we use two different approaches to investigate system change and its impact on the architecture and design of the system.

The first approach involves traditional software change impact analysis. We propose a new and different traceability model, which is based on Genetic Software Engineering (GSE). The proposed traceability model exploits some features of GSE to create a number of advanced properties that are rare in other traceability models. For example, once a software change has been fully captured, some other design documents including the component architecture and component behavior can be automatically generated/updated. All the consequent change impacts are presented in a clear way. We have also introduced the concept of evolutionary design documents that show the evolution process of a system's architecture as well as the design of individual components. Using this proposed traceability model, a practical method to normalize and simplify the component architecture of software intensive systems has been developed. An important result we have proved is that the component architecture of a software system is independent to the functional requirements of the system. We claim that a normalized software system is easier to maintain and change.

The second approach starts from a macro view. Rather than exploring the details of the change impacts from individual changes, this approach focuses on the common properties of the architecture evolution of complex systems; it stresses the topological structure from an evolutionary viewpoint. For this investigation we use scale-free networks and hierarchy theory as the major tools. Hierarchy is a natural structure for diverse large and complex systems, and recent studies reveal that many large networks from different domains are scale-free. In this research, we have discovered that the component dependency networks of many software systems are scale-free; we have also found that there is a close connection between the scale-free feature and the optimization of sorting algorithms. These results imply that there are fundamental rules working behind the evolution of large systems including software intensive systems, and that the scale-free property can be used as a possible index for the optimization level of the structure of a system.

Software change and software evolution are critical aspects of software engineering. This thesis has used a macroscopic and technical, formal approach to make positive contributions to understanding and accommodating change of software-intensive systems.

Statement of Authorship

I declare that this work has not previously been submitted for a degree or diploma in any university. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made in the thesis itself.

Signed

Date

Related Publications

Wen, L., Dromey, R.G., "*From Requirement Change to Design Change*", 2nd IEEE International Conference on Software Engineering and Formal Methods, pp.104-113, 2004

Wen, L., Dromey, R.G., "*Architecture Normalization for Component-Based Systems*", Proceedings of the International Workshop on Formal Aspects of Component Software 2005, pp: 247-262, this paper will also be published in ENTCS.

Wen, L., Colvin, R., Lin, K., Seagrott, J., Yatapanage, N., Dromey, G., 2007, "*'Integrare', a Collaborative Environment for Behavior-Oriented Design*", in Proceedings of the Fourth International Conference on Cooperative Design, Visualization and Engineering, LNCS 4674, pp. 122-131, 2007a.

Wen, L., Kirk, D. , Dromey G., "*Software Systems as Complex Networks*", in Proceedings of The 6th IEEE International Conference on Cognitive Informatics, IEEE CS Press., 2007b

Wen, L., Kirk, D. , Dromey G., "*A Tool to Visualize Behavior and Design Evolution*", in Proceedings of The International Workshop on Principles of Software Evolution (IWPSE2007), 2007c

Table of Content

List of Figures

Chapter 1 Introduction

1.1 Motivition and Approaches

"Complexity is a sign that our methods are no longer adequate to the task"

--- McWhinney

Our world is ever changing. It does not matter whether we are talking about society or software systems. *"Commercial and industrial firms have been adopting open system management at a higher rate over the past few years – flatter organizations, more use of teams, more concern with customers"* (McWhinney, 1997). Possible drivers for this change are globalization and competition. Similar things have been happening in software engineering. The development of the Internet has provided a global platform that theoretically can link any two software systems together if they are running on computers connected to the Internet. New technologies such as modeling languages, web services, component-based design, middleware and distributed data and processing enhance a modern software system's capability to integrate with other systems. In the last half century, software systems have become larger, more complex and more connectable (Albin 2003). It is clear that this trend will continue, especially with the rapid development of the Internet, the trend for software

globalization is clear. We can therefore expect the continuing emergence of even larger and more complex systems. How to control and manage the change in the architecture of software systems (especially of large scale) is therefore a challenge that we will continue to face in software engineering.

Traditional software systems are more like closed systems that are defined as "*mechanically self-sufficient, neither importing nor exporting*" (Rice 1963). Following this definition strictly, a real software system probably never is a closed system, but for many old single user software systems, the keyboard and the screen are likely to be their only importing source and exporting target. For this kind of simple system, usually the designers have total control over the software design and the architecture. However most large modern systems are open systems that "*exist and can only exist by the exchange of materials with their environment"* (Miller 1993). These kinds of systems have to integrate with components from other systems through interfaces that follow certain protocols to form super systems (Stafford 2001). Many super systems are nationally-scaled or even globally-scaled. Examples are the WWW and national defense systems.

Usually, a globally-scaled super system consists of a number of large systems that have been designed and developed independently by different groups over many years. There is no single pre-defined blueprint for the architecture of such a super system. It evolves due to effects from many different and unexpected sources, and sometimes the evolution process even appears to be random. However, there are laws that work behind the evolution of these large or super systems. An attempt to understand and study the laws that dominate the change and evolution of large software systems is one of the major motivations that has inspired this research.

In this research, the concept of components is used as the cornerstone to build software systems. The basic line of this research is to investigate the change impact on component architecture and the architecture evolution which is the effect of a series of changes. We have conjectured that the laws, which rule the evolution of other types of complex systems, may also rule the evolution of large and complex software intensive systems. Some results from this research has support this conjecture.

Component-based software design and development is one of the major trends to handle large and complex software systems (Szyperski 1999, Wadler 1999). Even though there is no universally agreed definition of the term "component", it is frequently used in papers on software engineering.

In this thesis, the concept of component is kept at its most abstract level. A system is composed of many components. Each component has its own component level functionality and can be integrated with other components. All the components work together to form a system. Then the system must work in an environment. In this environment, there are other systems. If the environment is treated as a high level system, then the original system will be one of the high level system's components. Similarly, for the original system, it can be treated as the environment of its components and each component may be a low level system with it own components. In this way, the concepts of components, systems and environments form a hierarchy, which can be extended both upward and/or downward.

To study the laws of change of large software systems, this thesis has used two

different approaches. The first is more traditional and it treats each time of software change as one event and investigates the change impact from this event on a software system. The second approach tries to tackle the change problem from a broader view. In this view, we will not concentrate on the change impact of a single change; instead we focus on the change impact of a sequence of changes, in other words, software evolution. Through software evolution, not only are individual systems growing and becoming more complex, but many software systems are also merging together to form larger systems. In these large systems, because they include so many components, an individual component may become less important or even invisible from the high level view. What is most important is the architecture of the system. Therefore, this approach is to understand the topological structure and the evolutionary process that affects the architecture of complex systems. Then with a better understanding of the rules that underpin the evolution of the architecture of large system, practical methods that might help to optimize and manage those systems could emerge.

1.1.1 The Software Change Impact Analysis Approach

For large component-based software systems, the component architecture (or the software component dependency network) is a critical factor to determine the quality and maintainability of the system. When a software system is changed due to the changes of the functional requirements, the software architecture is usually affected. The question is *when a software system is changed due to the modification of the system's functional requirements, what is the change impact on the architecture*?

The fundamental methodology we will use to approach the question is Genetic

Software Engineering (GSE) (Dromey 2003). GSE is a newly developed formal method to enable component-based software design from functional requirements; it allows designers to create a design *out of* the functional requirements. The details will be introduced in Chapter 3. GSE provides a tree-like graphic notation called behavior trees and uses behavior trees to describe the behaviors of a system and components; the architecture and other design diagrams can be projected out through mathematically defined procedures from a large behavior tree (called design behavior tree) which describes all the integrated behaviors of a system. The GSE method has been applied to many aspects of component-based software system research and rich results have been achieved (Dromey 2005, Gonzalez-Perez 2005, Glass 2004, Colvin 2006, Winter 2004, Zafar 2005, Wen 2004, 2005 and Zheng 2003).

The traditional study to target the question raised in this section is called software change impact analysis (Bohner 1995, 1996). Software change impact analysis studies the ripple effects of changes in software systems. One of the branches, called traceability analysis, investigates the traceability between different types of software artifacts in a software system. Once an artifact is changed, we know what other artifacts could be affected. GSE allows us to develop a new traceability model that can identify the change impact on software architecture and even automatically update different types of design documents when some of the functional requirements are changed. For large software systems, the development procedure is incremental because new functional requirements are gradually added into an existing system. When a traceability model can be automatically applied, it is possible to build a model to control the version and review the evolution of the architecture of the software system.

1.1.2 Complex Systems Approach

For large and complex systems, we have selected hierarchy theory (Ahl 1996) and scale-free networks (Barabási 2002) as the major research tools.

Hierarchy is a general structure for managing large systems, from large companies with thousands of employees to multiple millions of people in nations. The most important features of a hierarchy are scalability and simplicity. The complexity of each single component can be limited even though system growth is nearly unlimited. The same concept can be applied to large software systems that may have as many as hundreds or thousands of components.

Many large software systems are grown by "incremental" or "iterative" development (Mills 1971, Boehm 1988), the dependent relationships among the components are weaved as a complex network. Just like many other complex networks (for example, human relationship networks, the internet router networks), the dependency network is not fixed and not totally pre-designed. All such networks are gradually built up or evolved by affects from so many aspects that it is usually impossible to predict what will be the final topological structure. However, observations suggest that most of the large-scale networks from different areas are scale-free (Barabási 2002). The ubiquity of scale-free networks inspired us to investigate the topological structure of software component dependency networks with the expectation that they would be scale-free as well. We also expect the study of scale-free networks will help us to understand more about the evolutionary process and its impact on the

software architecture. This leads to a proposal for an optimized form for software component dependency networks.

During our research, the sorting problem was also explored. Sorting is one of the fundamental problems of computer science. In the last 50 years, hundreds of sorting algorithms have been invented (Knuth 1997c). What is interesting is that the process of sorting is similar to the process for constructing a complex system. If we treat individual records as components, and the whole sequence as a "system", then the function of the "system" is to make itself sorted. For a general sorting algorithm, one of the inevitable actions is to compare the key values of two records. Therefore we can treat the comparison of two records' key values as making a connection between the two components. In order to make the sequence fully sorted, we need a series of comparisons which results in all the records being connected into a sorting comparison network. This process is similar to making a system achieve its system level functions, by connecting all of its components into a network. Studying the evolution of a sorting comparison network may help us to understand the evolution of other complex networks.

Some of the advantages of using sorting as a research method are scalability and repeatability. For good sorting algorithms, it is not hard to sort sequences with thousands or even millions of records. Also, once the sorting algorithm is determined and the given sequence is fixed, the sorting procedure is fixed and repeatable. Different sorting algorithms have different efficiency and the difference may be reflected in their comparison networks. Studying the different topological structure may provide clues on good forms for a complex system architecture.

In summary, the main motivation has been to study the change impact and evolution process of large complex software systems. The first approach has been to invent a new practical traceability model to identify the change impact on software architecture when a software system's functional requirements are changed. The other approach has been to study the evolution and optimization of the architectures of large software systems. The elementary platform we used is GSE; the associated techniques and research directions cover software change impact analysis, hierarchy theory, sorting algorithm and scale-free networks. The expected outcomes will be beneficial to understanding of the evolutionary process for large software systems' architecture, the topological structure, and the optimization of the architecture. The goal is through these understandings to reduce the cost of software maintenance.

1.2 Summary of Contributions

In summary, the thesis concentrates on software change, the change impact on software architecture, the topological structure and evolution process for software architectures of large complex systems and optimization of a software systems's architecture. The main contribution of this thesis is to introduce a **traceability model** and its extension model to manage software change and traceability; to prove the **software architecture independent theorem**; based on this theorem, to propose the **universal optimized software architecture**; suggest that during the evolution of a software system, a **tree** is a **possible optimized software architecture**; through the study of different sorting algorithm, to discover that the close relationship between **scale-free networks, software architecture and optimized sorting algorithms**. Therefore conjecture a scale-free network plays an

important role in optimizing software architecture and the software evolution. **Some software tools** have been developed to support the research. Some explanations are listed below:

- **New Traceability Models**: Based on GSE, *a new traceability model is introduced to map changes from the requirements domain to the design domain, and this model is also expanded to handle multiple changes over time, control of versions, and review of the evolutionary history of the architecture and the design.* GSE provides a formal process to translate individual functional requirements into corresponding behavior trees and integrate them into a design behavior tree[1]. Then, from this design behavior tree, different design diagrams, which cover the architecture and the low level designs of individual components, are projected out. In our traceability model, if the functional requirements are changed, we use a tree comparison algorithm to merge the old design behavior tree and the new design behavior tree into an edit behavior tree. From the edit behavior tree, different edit design diagrams including the component architecture can be projected out. On these edit design diagrams, changes in the architecture, the component behaviors and component interfaces caused by the changes in requirements are clearly marked. One substantial advantage in this method is that, except for translating functional requirements into behavior trees, the entire process can be supported by automation tools and in many cases completely automated. Further more, this traceability model is expanded to handle multiple changes over time. Multiple versions of design behavior trees

[1] A behavior tree (BT) is a tree form graph to describe a piece of behavior of a system. A design behavior tree (DBT) is a tree form graph to capture all the behavior of a system. More information about behavior trees and design behavior trees are given in Chapter 3.

can be merged into an evolutionary design behavior tree. From the evolutionary design behavior tree, different types of evolutionary design diagrams can be projected out. These evolutionary design diagrams record multiple versions of a design of a software system. From these diagrams, any versions of the design documents as well as the difference between any two designs can be generated by software tools. The main ideas of this work are published in the paper (Wen 2004)

- **Architecture Independency Theorem**: By using GSE, *it is proved that the component dependency network*[2] *or software architecture can be independent of the software's functional requirements*. In GSE, the algorithm to project the component dependency network out from the design behavior tree is clearly defined. Once the design behavior tree is determined, the associated dependency network is determined. However, we find that by inserting bridge component states in the design behavior tree, which do not change the functional requirements at the functional level, we can modify the associated dependency network. Finally, we have proved that by inserting suitable bridge component states in the design behavior tree, the corresponding dependency network can be adjusted to any pre-defined form. In contrast to the obvious assumption that the functional requirements determine the dependency network or the architecture, our result proves that they can be independent. The independence of the component architecture plus our proposed traceability model *enable us to develop a maintenance method which can keep the component architecture stable while the software system is under*

[2] A component dependency network (CDN) is also called a component integration network (CIN). In GSE, a system is composed of many components; these components are integrated or dependent on each other and these integration or dependency relationships form a network. We call this network the CDN (or CIN) of the system. More information about CDN and CIN is given in Chapter 3.

repeated changes.

- **Possible Optimized Software Architecture**: Furthermore, the independence property of component architecture opens up the possibility to investigate the universally optimized form of a software architecture that is independent to the software system's functional requirements. A *tree-structured hierarchical dependency network is proposed as an optimized form for dependency network or software architecture for large software systems,* because of the unique features of trees. We find that a tree-structured hierarchy has some features that are also shown in scale-free networks, but it still has some other features such as the least number of links, a unique path between any two nodes that are unique for trees. These features make software systems of this form much easier to understand and maintain. We suggest that this kind of structure can be gradually implemented into large or even super software systems as an optimization for their architectures.
- **Scale-Free Networks and Software Architecture***: The class dependency networks of several Java packages are investigated and they are found to be scale-free networks*. A scale-free network is a new model differing from the traditional random network model that has dominated the network theory for about 40 years. In the recent years, many complex networks have been discovered to be scale-free. We have tested several different Java packages and found all the dependency networks are scale-free. This result backs up the theory in the previous contribution. This result also benefits the design, maintenance and study of the optimization of large software systems.
- **Scale-free Networks and Optimized Sorting Algorithm***: We have shown that for some of the efficient sorting algorithms, the associated sorting comparison networks are scale-free networks*. For a general sorting algorithm, one unavoidable operation is

to compare the key values of two records in the target sequence. If we treat each record in the target sequence as a node, and each comparison as a link between the two compared records, when the target sequence is sorted, we will be able to draw a network and this network is called *sorting comparison network*. In our research, we have examined 5 common sorting algorithms (bubble sort, quicksort, heapsort, binary insertion and merge insertion), and we find that the scale-free property is only noticeable in the sorting networks of highly efficient sorting algorithms[3], and the sorting network is more like a random network for less efficient sorting algorithms such as bubble sort. This result suggests that a scale-free property is an indicator for the efficiency of sorting algorithms, and it also provides a more deterministic approach to study scale-free networks as well as the evolution of the architecture for complex systems.

- **Software Tools:** In this research, several software tools have been developed to simulate the GSE process, collect data and prove conjectures. The first tool is called "Genetic Software Engineering Toolkit" (GSET) which is used to simulate the GSE process and demonstrate the proposed traceability model and the architecture normalization. The second tool is called "Class Network" which is used to investigate the class dependency network of Java packages. The third tool is called "Sorting Comparison Network Explorer" (SCNE) which is used to investigate the sorting comparison networks of different sorting algorithms. In this thesis, many testing results and diagrams are obtained through the usage of these software tools. Some of the tools can be freely downloaded through the Internet and they could be valuable for other researchers as well (Tool Download 2006a, 2006b, 2006c).

[3] Here we define the efficiency of a sorting algorithm only by the number of comparisons. A sorting algorithm with less number of comparisons is regarded as more efficient.

1.3 Thesis Structure

Chapter 2 first reviews some existing techniques for software change impact analyses that focus on the consequences for a software system when some parts are changed. Software change impact analysis includes two major branches, dependency analysis and traceability analysis. The second branch is particularly concerned with the traceability among different types of software artifacts when a software system has been changed. At the end of chapter 2, a brief review and analysis of model driven architectures (MDA) is also presented.

Chapter 3 reviews the concepts, notations and processes of genetic software engineering (GSE) that is used as the essential platform for the majority part of the research in the thesis. GSE is a formal method for component-based design from the functional requirements. The main concept in GSE is to use behavior trees to describe the desired behaviors of a target system. The component-based design can then be retrieved from the integrated requirement behavior tree which is also called design behavior tree (DBT) through clearly defined procedures. Most parts of the GSE processes can be implemented using automated tools.

Chapter 4 introduces a new traceability method that can map software changes from the problem domain to the solution domain. The expanded version of this model can even handle multiple changes of a system in time and present a record of how a system has evolved over time. Traditional traceability analysis techniques require manual definition of the relationship between different software artifacts and when some parts are changed, they only indicate what other parts may be affected but

they cannot update those parts automatically. In the proposed method, most of the tracing can be done using the automated tools. A tool, which has been introduced in Chapter 7, has actually been built to demonstrate this functionality.

Chapter 5 elaborates on the concept and proof of a major theorem in this research. The general concept of this theorem is that the architecture of a software system can be (or can be made) independent to its functional requirements. GSE is used to prove this theorem. From this result, two important deductions have been explored. The first is that based on the traceability model in Chapter 4, we can introduce a model that can reduce the change impact on the software architecture when the functional requirements of the system have been changed, so that the architecture of the system can be stable during its lifecycle while the functional requirements of the system have been continuously changed. Another deduction is the possibility of creating universal optimized software architectures that can be independent of the software's functional requirements. In this Chapter, we have proposed a tree-structured architecture as an optimized form.

The first part of chapter 6 reviews the latest developments in network theory, especially scale-free networks. The concept of Java class dependency networks is then introduced. The Java class dependency network is equivalent to the component dependency network in component-based software system. In this chapter, we present the discovery that all the tested Java class dependency networks of different packages are scale-free networks. From this result, we propose the conjecture that software dependency networks or software architecture for large software systems are scale-free. The rest of this chapter studies sorting comparison networks. We have discovered that the sorting comparison networks of high efficient sorting

algorithms (sorting algorithms with less number of comparisons) tend to be scale-free. This result implies that a scale-free network is a possible optimized form for networks. The study of sorting comparison networks also provides an approach to study scale-free networks and network evolution.

Chapter 7 introduces the three software tools developed for the research. Genetic Software Engineering Toolkit (GSET), Class Network and Sorting Comparison Network Explorer (SCNE).

Chapter 8 is the summary of this thesis. It includes a brief review of the contributions of the research, more discussions about the results and suggestions for the possible future studies.

Chapter 2 Software Change and Software Architecture

Most software systems undergo continuous changes during their lifetime and many software designers and developers realize the difficulties of changing a large software system, especially when the changes involve the software architecture. Whenever a system's architecture is changed, many parts of the system will be affected and need to be re-designed, re-developed and re-tested. Without a well implemented software management system (Royce 1998) and a good design methodology, it will be complicated to trace all the ripple effects of changes, and it may take a long time to identify and eliminate all the bugs caused by the changes. Our research is mainly focused on software change and change impacts especially on software architecture.

In this chapter, some recent research directions and results related to software change and software architecture are presented and reviewed.

2.1 Software Change

Software change for large systems is very costly. According to statistics reported in

1995, software change consumed up to 90 percent of software resources. The software development cost for the U.S. Air Force F-16 jet fighter was US$85 million, but the estimated lifetime software maintenance cost was US$250 million (Suydam 1987).

To deal with this problem, some technologies such as object oriented design (Booch 1993, Jacobson 1992, Rumbaugh 1991), component-based systems (Szyperski 1999, Akşit 2002) and the recent design approach, Model Driven Architecture (MDA) (Poole 2001, ORMSC 2001), have been investigated. Even though the main incentive for most of these technologies may not be maintenance (Bengtsson 1999), they help to simplify maintenance. Apart from those technologies mentioned above, software change impact analysis (or impact analysis), which is mainly focused on software changes is possibly most relevant to this issue. In this chapter, some of the latest researches about software change are reviewed.

2.1.1 Reasons for Software Change

One of the most important reasons of why software change and software maintenance are so expensive is because of the difficulties in changing software. Nearly all software systems undergo some changes in their lifetime. There usually are four major reasons for software change or maintenance (Sommerville 2004, Buckley 2005) :

- Adaptive – changes in the software environment
- Perfective – new user requirements
- Corrective – fixing errors
- Preventive – prevent problems in the future

According to Bennett (2000), *"the incorporation of new user requirements is the core problem for software evolution and maintenance"*, which suggests that the change of user requirements is one of the major reasons for software change.

For large and complex software systems, when the software requirements are changed, it can be very difficult to map the changes (in the problem domain) into the corresponding changes of the source code (in the solution domain), related design documents and many other software lifetime objects (SLO). It is very time-consuming to trace individual requirements in the source code and other documents and, at the same time, determine the ripple effects of any proposed changes. Other reasons that make the task difficult include: the maintenance team may not have enough knowledge and experience; the documentation might be incomplete and/or inconsistent; and the architecture of the system might be too complex or too specific. Additionally, repeated changes might destroy the architecture of a system, thus destroying the correspondence between the documentation and implementation, and introducing new defects. These issues increase the cost of maintenance and eventually people may find that continuously maintaining an aged system is even more expensive than developing a new one.

2.1.2 Minicycle of Software Change

According to Rajlich (1999), Software change is a process consisting of several phases:

- Request for change
- Planning phase:
  - Program comprehension
  - Change impact analysis

- Change implementation
  - Restructuring for change
  - Change propagation
- Verification
- Redocumentation

The first phase is request for change, according to the previous subsection, there are four different types of reasons that lead to a change, but we will mainly deal with the change caused by new or modified functional requirements. The next phase is planning phase, which includes program comprehension and change impact analysis. This is the area that our work mostly contributes to. The minicycle presented in this section is only one of the typical processes. It has been presented in different forms by other researchers (Yan 1978). However, the basic concepts are the same.

Even though our research is focused on the change impact analysis, our work has positive affects in other phases as well, e.g. the program comprehension and redocumentation phases. This will be discussed later.

2.1.3 Software Stage Model

Regarding the changeability of a software system, the whole lifecycle of a software system can be identified as five different stages (Bennett 2000). A simple version of the stage model is shown in Figure 1.

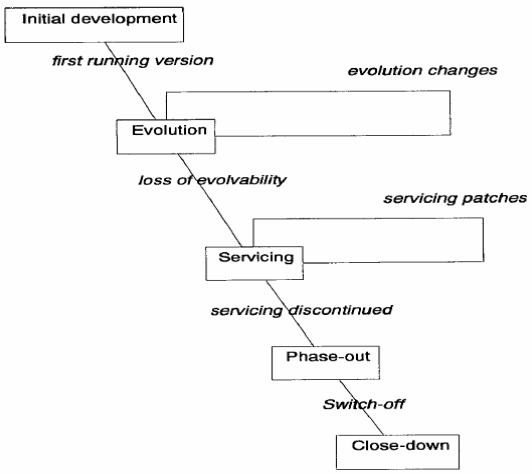


Figure 1. The simple stage model

In this stage model, there are two stages, in which a software system is changeable, the evolution stage and the servicing stage. While in other stages, either the software system has not been delivered or it becomes unchangeable. The difference between the evolution stage and the servicing stage is that the evolution stage directly follows the initial development stage (Lehman 1985); in this stage, the software system is evolving to trace the ever-changing user requirements. Performance is improved and faults are also corrected in this stage. During this stage, the software team has sufficient knowledge of the software architecture as well as the business domain knowledge so it is possible to make substantial changes in the software without damaging the architectural integrity. The system reaches its highest vitality in its lifecycle. Continuous change of the system may eventually damage the software architecture. Especially, if some key personnel leave a project, it may cause the loss of the necessary knowledge for system evolution. In this situation, the software system enters the servicing stage. In this stage, the changes in the software will lead to a faster deterioration of the architecture, and substantial changes become impossible. According to Bennett (2000), a software system will eventually shift from the evolution stage to the serving stage, but it is irreversible from the servicing stage back to the evolution stage.

The stage model points out two different stages where a software system can be changed to match the evolution of user requirements. However, the change in the evolution stage is much easier, has fewer side effects on the system architecture and can be more fundamental, so it is a stage that can keep the system in a healthy form. But in the servicing stage, the change will destroy the architecture of the system and will gradually make the system a mess so that no further change is possible. The essential elements that determine the stages are the traceability and the architecture knowledge. Therefore, a good traceability model that can trace the evolution between different software artifacts will provide sufficient information to keep a software system in the evolution stage and increase the lifespan of the system – that is one of our motivations for this research.

2.2 Software Change Impact Analysis

Software change impact analysis (impact analysis for short) is a critical phase in the minicycle of software change (Rajlich 1999). The aim of it is to estimate what will be affected in the software and related documentation if a proposed software change is made. Impact analysis information can be used for planning changes, making changes, estimating the cost of changes and generally maintaining software.

Typical examples of impact analysis techniques include: (Bohner 1996c)

- Using cross-referenced listings to see what other parts of a program contain references to a given variable or procedure;
- Using program slicing to determine the program subset that can affect the value of a given variable;

- Browsing a program by opening and closing related files;
- Using traceability relationships to identify software artifacts associated with a change;
- Using configuration-management systems to track and find changes;
- Consulting designs and specifications to determine the scope of a change.

Generally, there are two major technology areas for impact analysis: *dependency analysis* and *traceability analysis*. Dependency analysis focuses on the dependency relationships between program entities (variables, logic and models) in low-level software objects such as source code. Traceability analysis focuses on the relationships among all types of software lifecycle objects. It addresses impact analysis from a broader perspective.

Both approaches have their respective advantages. Relatively, dependency analysis is suitable for impact information captured from source code, but it is the most mature impact-analysis technique available because of automated tools that can capture dependency information from source code. Today, software projects are becoming larger and more complex. They are supposed to work in different environments and cooperate with more different systems. These kinds of projects tend to have a large number and wide variety of artifacts. Traceability techniques are needed to model the relationships and the dependency among these different types of software artifacts.

2.2.1 Dependency Analysis

Dependency analysis is one of the two major approaches of software change impact analysis that involves examining detailed dependency relationships between

program entities (variables, logic, and modules). It provides a detailed evaluation of low-level dependencies in code but does little for SLOs of other levels (Bohner 1996). Generally, it starts from decomposing low level SLOs such as the source code. One of the most important supporting techniques used for dependency analysis, *program slicing,* captures "slices" of programs.

A *slice* of a program is taken with respect to a program point *p* and a variable *x*; the slice consists of all statements of the program that might affect the value of *x* at point *p* (Horwitz 1990, Weiser 1984). The concept of "slice" is useful in software change impact analysis, because once the value of *x* is changed, through the slice, we can trace back the statement that caused the change. To retrieve program slices, a technique called the system dependence graph has been introduced (Horwitz 1990). Besides the low level program slice, recent researchers have studied slices of high level software artifacts such as the slice of a software architecture (Zhao 2002).

Other techniques used for dependency analysis include data dependency, control dependency and component dependency (Bohner 1996). *Data dependencies* focus on the dependent relationships between program states that define and use data (Loyall 1993, Ferrante 1987). *Control dependencies* focus on the relationships between program statements that control program execution (Loyall 1993, Podgurski 1990). *Component dependencies* focus on the relationships between software components such as modules (Perry 1989).

2.2.2 Traceability Analysis

For today's software systems, there exist a large number and wide variety of

software artifacts or SLOs such as user requirements, design documents, configuration files, source code, binary resources, and testing reports. These SLOs are stored in a repository and they are related in certain ways. Traceability is an internal relationship among the SLOs. According to IEEE (1993), "A software requirements specification is traceable if (i) the origin of each of its requirements is clear and if (ii) it facilitates the referencing of each requirement in future development or enhancement documentation". In DOD (1985), traceability is defined as "the association of data generated in a particular life-cycle activity with other data generated in predecessor and successor activities; an attribute of software requirements, design, the software product, or documentation indicating that they derive from a higher source and can be allocated to a lower level, if required." In Gotel (1994), traceability is defined as "the ability to describe and follow the life of a requirement, in both a forward and backward direction". Similarly, Bohner and Arnold define traceability as "the ability to trace between software artifacts generated and modified during the software product life cycle" (Bohner 1996c). Traceability analysis focuses on the techniques that build up and utilize the traceability relationships among those SLOs so, if some of the SLOs are changed, the other affected SLOs can be identified and retrieved efficiently.

A typical technique for traceability analysis is called the *document management system*, where different SLOs are stored as different types of documents in centralized software-engineering environments. This system usually implements some query mechanism so it can easily identify and browse related documents (Bohner 1996c). Another more passive technique is to record all the traceability data in traceability matrices and store them in a database. From this data, users can identify the potential impacts of changes.

In the next section, different traceability analysis systems are introduced.

2.3 Different Approaches for Software Change

In this section, a number of different approaches related to software change are reviewed. Most of them are related in different aspects to our proposed software change traceability model. However, of course, none of them are really similar to our work. In Chapter 4, after our model has been formally presented, a brief comparison with all of these approaches is given.

2.3.1 DIF (Document Integration Facility)

Document Integration Facility (DIF) is one of the implementations of traceability analysis. DIF utilizes a hypertext system to define, store and manage different types of software documents of multiple software projects in one integrated environment (Garg 1990).

Virtually, DIF can store any type of document, from the user requirements, design documentation, or source code to test reports. Users can manually create links between documents and those links reflect the relationships between those documents. Each document in DIF is called an object. DIF also provides software-engineering tools to process the information in the objects. By judiciously using links, keywords, and information structure, users can alleviate problems of traceability, consistency and completeness.

The core part in DIF is the System Factory. It can generate eight documents during the software produce process. They are:

1. Requirement specification
2. Functional specification
3. Architectural specification
4. Detailed-design specification
5. Source-code document
6. Testing and quality-assurance document
7. User manual
8. System-maintenance guide.

DIF uses links, keywords, forms and compositions to represent the relationships among documents and build the structure for the whole system. Figure 2 is a typical structure of DIF system.

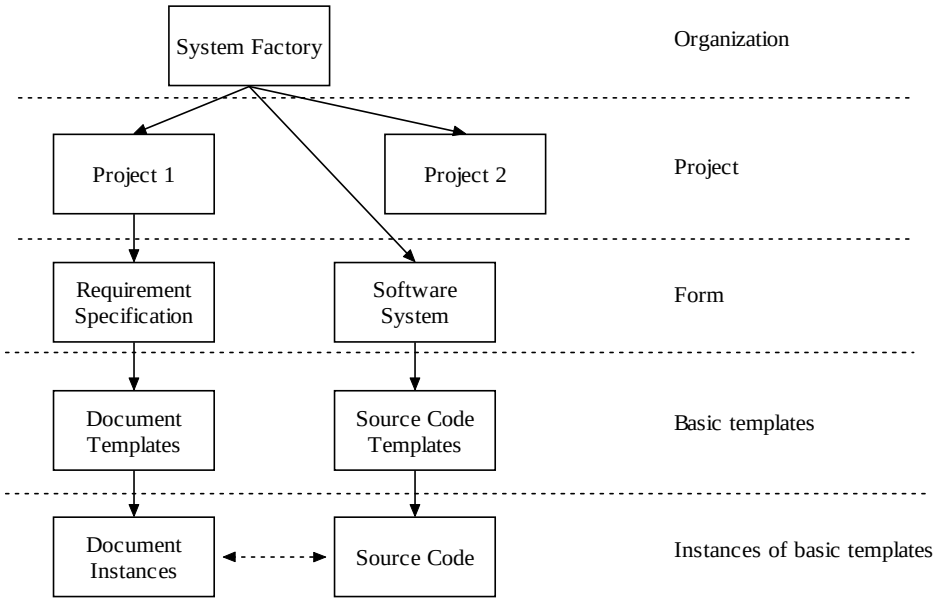


Figure 2. Hypertext structure for life-cycle documents.

DIF, as a management tool for software engineering, has the advantage of providing an integrated environment for users to search, locate and browse related document quickly. It also provides a partial revision facility[4] that can be used to trace the evolution of a large system. When one document in the system is changed, DIF may indicate which other documents might need to be updated to keep the system consistent, but it cannot update those documents automatically or provide hints on how those documents should be updated.

2.3.2 SODOS (Software Document Support)

SODOS is a computerized system which supports the definition and manipulation of documents used in developing software (Horowitz 1986). The central idea of SODOS is to have all information generated at the specification and development phases available to the maintenance personnel in a complete, structured, and traceable form.

The main processes in practicing SODOS include:

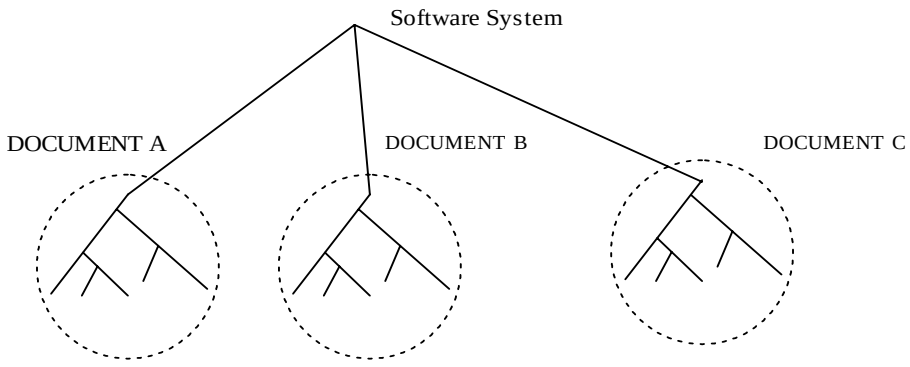
1. Defining documents. In SODOS each of the documents is represented as an instance of a document class. From document classes, other classes such as interface document, requirement document etc, are derived.
2. Defining a document structure. One document may have the following sections:
    a. Introduction
    b. Commands

[4] DIF utilizes both a file system and database to store information. The revision-management facility only supports the file system part but not the database, in which DIF stores keywords and links.

c. Error Recovery
d. Performance Monitoring.

The structure information is stored in a relational database (see Figure 3).

3. Defining document content. The contents of a document may include a set of keywords and/or graphics which can be used to build relations within and between documents.
4. Interrelating documents. The documents are associated with each other based in predefined relationships which depend on the semantic context of the documents (see Figure 4). For example, a system requirement is related to a functional requirement by the "derived-from" relationship. The functional requirement in turn is related to a design module by the "required-by" relationship.



DOCUMENT RELATION

ID	TITLE	TYPE	REVISION	DATE	AUTHOR	STATUS	INSTANCE

Figure 3. Representing document instances and document structure in SODOS

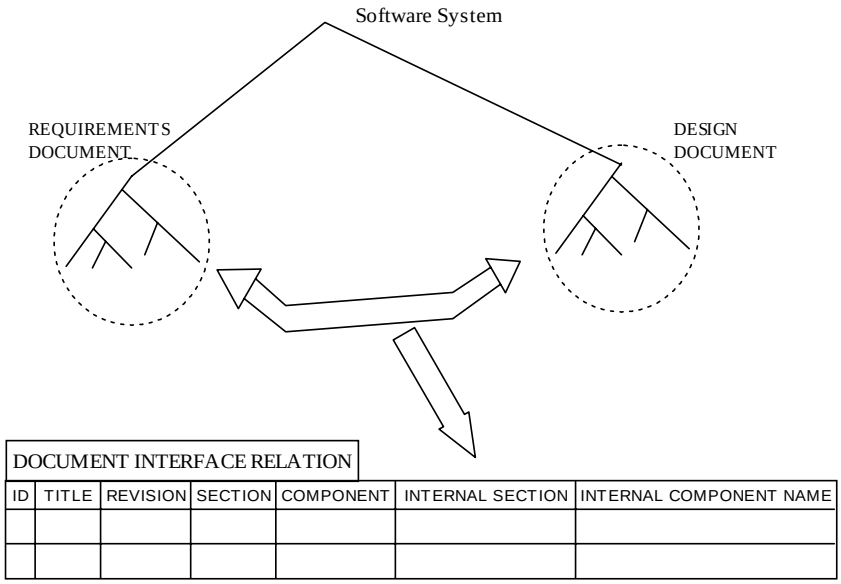


Figure 4. Representing document interfaces in SODOS

2.3.3 Traceability Approach Based on B Model

Bouguet (2005) claimed to have an approach to automatically produce a Traceability Matrix from requirements to test cases. And this approach will benefit change impact analysis by identifying all application elements affected by a requirement change.

The process of test case generation includes 5 steps (See Figure 5)

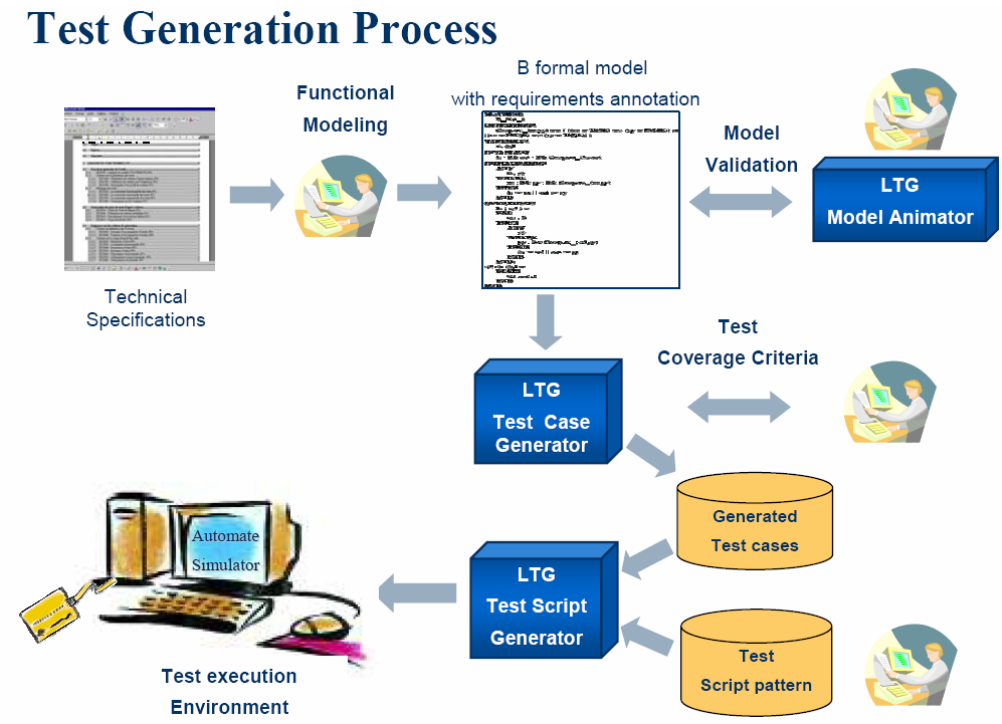


Figure 5. The test generation process based on B formal model

The first step is called "formal model development": In this step, the functional requirements are modeled with a B abstract machine (Abrial 1996, Schneider 2001). In the second step, the formal models created in the first step are validated using the LEIRIOR Test Generator (LTG) symbolic animator. The test cases are generated in the third step by LTG test case generator. In the last two steps, the test script is generated and executed in a test execution environment.

The interesting parts of this approach are the first step and the third step. In the first step, requirements are formalized with B abstract machine; in the third step, test cases are generated from the models to cover all the effects (Legeard 2004) and boundary analysis (Beizer 1995). The details of the LTG generation strategy can be found in (Bouquet 2004) and they are not related to our research. What is relevant to our research is to use B abstract machine to model the requirements and work as

the main thread for traceability. Figure 6 shows the B operation for DISABLE CHV (Figure 7) with requirements traceability annotations added.

```
sw <-- DISABLE_CHV(code_cc) =
PRE
code_cc : CODE
THEN
     IF (blocked_chv1_status = blocked) THEN
          sw := 9840 /*@REQ: DISABLE3 @*/
          ELSE
               IF (enabled_chv1_status = disabled) THEN
                    sw := 9808 /*@REQ: DISABLE2 @*/
                    ELSE
                         IF (code_cc = pin) THEN
                         /*@BEGIN_REQ: DISABLE4 @*/
                         try_counter_chv1 := 3 ||
                         enabled_chv1_status := disabled ||
                         permission_session(chv1) := true
                              /*@REQ: DISABLE1 @*/ ||
                         sw := 9000
                         /*@END_REQ: DISABLE4 @*/
                         ELSE
                              IF (try_counter_chv1 = 1) THEN
                              /*@BEGIN_REQ: DISABLE6 @*/
                              try_counter_chv1 := 0 ||
                              blocked_chv1_status := blocked ||
                              permission_session(chv1) := false ||
                              sw := 9840
                              /*@END_REQ: DISABLE6 @*/
                              ELSE
                              /*@BEGIN_REQ: DISABLE5 @*/
                         try_counter_chv1 :=
                              try_counter_chv1 - 1 ||
                              sw := 9804
                         /*@END_REQ: DISABLE5 @*/
                    END
               END
          END
     END
END;
```

Figure 6. Requirements of DISABLE CHV expressed in B notation

"*The successful execution of this function has the effect that files protected by CHV1 are now accessible as if they were marked "ALWAYS"* **[Disable1]**
The function DISABLE CHV shall not be executed by the SIM when CHV1 is already disabled **[Disable2]** *or blocked* **[Disable3]**.
If the CHV1 presented is correct, the number of remaining CHV1 attempts shall be reset to its initial value 3 and CHV1 shall be disabled **[Disable4]**.
If the CHV1 presented is false, the number of remaining CHV1 attempts shall be decremented and CHV1 remains enabled **[Disable5]**.
After 3 consecutive false CHV1 presentations, not necessarily in the same card session, CHV1 shall be blocked and the access condition can never be fulfilled until the UNBLOCK CHV function has been successfully performed on CHV1 **[Disable6]**.*"*

Figure 7. Function requirements for DISABLE CHV

The advantage of this approach is the LTG tool support, but the disadvantage of this approach is that it does not provide a graph notation to model the functional requirements. To read and understand B notation requires a programming background and the multi-level nests of conditions increase the memory load when reading.

2.3.4 Architectural Slices and Chops

Zhao (2002) has proposed an approach to use architectural slicing and chopping technique to support software change impact on architectural level. According to Zhao, "*many techniques have been proposed to support change impact analysis at the code level of software systems, but little effort has been made for change impact analysis at the architectural level.*" In his approach, he has defined two types of architectural slices, forward slices and backward slices. An informal but simple way to understand architectural slices and chops is to think that a software architecture is presented as an architectural flow graph (AFG). If we select one point in this graph and there is a change in the

selected point, a forward slice is all the other parts of the graph that may be directly or indirectly affected by the change at the initial point; a backward slice of a given point is defined as a portion of the graph, if there is a change within this portion, the given point may be affected by that change. If we select two points in this graph, the chop of the two points in the graph is the set of all paths that connected from the first point to the second point. It can be retrieved as the intersection of the first point's forward slice and the second point's backward slice. The technique of architectural slicing and chopping is helpful to answer the following questions:

1. If a change is made to a component, what other components may be directly or indirectly affected by this change.
2. For a given component, a change on what other components has the potential to affect this component.
3. For two given components *s* and *d*, what are all the components that serve to transmit effects from the source component *s* to the target component *t*.

In Zhao's approach, the software architecture is presented in a type of architecture description languages (ADL), WRIGHT (Allen 1997). Of course, the same principle could be applied to other ADLs such as Rapide (Luckham 1995) and UniCon (Shaw 1995). In GSE, the software architecture is described as a component integration network (CIN), theoretically, it is possible to translate between a CIN and other architecture description languages, and so this change impact model can also be applied by GSE.

2.3.5 Difference and Union of Models

In 2003, an algorithm to merge different models into a final model has been proposed (Alanen 2003). The general purpose of this algorithm is for version

control and change management. The basic idea of the algorithm can be illustrated in Figure 8.

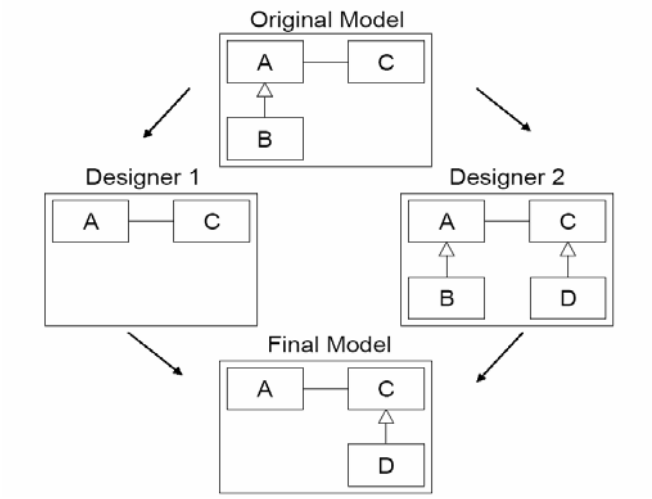


Figure 8. An algorithm to merge different models into a final model

In Figure 8, the top part is the original model and it is modified by two designers, and then the two modified models are merged to generate a final model.

The key points of this algorithm are summarized as following:

- In this algorithm, a targeted model is represented in a so called metamodel layer.
- A model is presented as a graph with linked elements called meta classes and the links are called meta-associations which include association ends and metafeatures.
- The syntax of a metamodel follows the UML standard.
- There are 7 basic operations "new, del, set, insert, remove, insertAt, removeAt" and each operation has a dual operation.
- The process of to change an old model into a new model equals to apply a sequence of operations on the old model. Based on the same model, if there are two different modified models, we can have two sequences of operations. When we want to merge the two modified models into a new model, we need

to merge the two sequences of operations into one sequence of operations and apply this new sequence of operations on the original model.

A major problem in this algorithm is that the orders of some operations are not exchangeable. For example, make a connection to a component must be after that component has been created. In some situations, it may cause conflicts when we try to merge two models together. For example, suppose that the first designer has removed a meta class A in his design and at the same time, the second designer has created a new meta class B under the meta class A. Then if we want to merge the two models together, it will be hard to decide how the meta class can be connected to the system because the connecting point A has been removed by the first designer. In this situation, the conflict must be solved manually by the designers. Generally, the model merge algorithm is similar to the tree merge algorithm in our traceability model, but due to the special features of trees and our different way of handling removed nodes, the conflict problem does not exist in our approach. The details of our approach and the comparison will be introduced in Chapter 4.

2.4 Software Architecture and Components

Another reason for the high cost of software change is the complexity of the software architecture, which is one of the key issues with software systems. Software architecture has attracted much research since the second half of the nineties. However, even though software architecture is the focus of many research and technical articles, there is no universally accepted definition. In Barroca (2000), "software architecture" is defined as the highest levels of a design. In Shaw (1996), it is defined as the computational components and the interactions among those components. Bass (1998) gives a more formal definition: "The software architecture

of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them." Even though there is no uniform definition of software architecture, one thing is clear: software architecture is about the components of a software system and the relationships among these components. For large software systems, where the number of components reaches hundreds or thousands, the relationship between those components can be extremely complex. In our research (Chapter 6), a common Java package, "java" includes only 1172 "components" (classes and interfaces), but has 9453 dependency links between them. If we draw these dependency links and the components as a network, practically, the network is too complex to enable any visualized detailed analysis at all (see Figure 9).

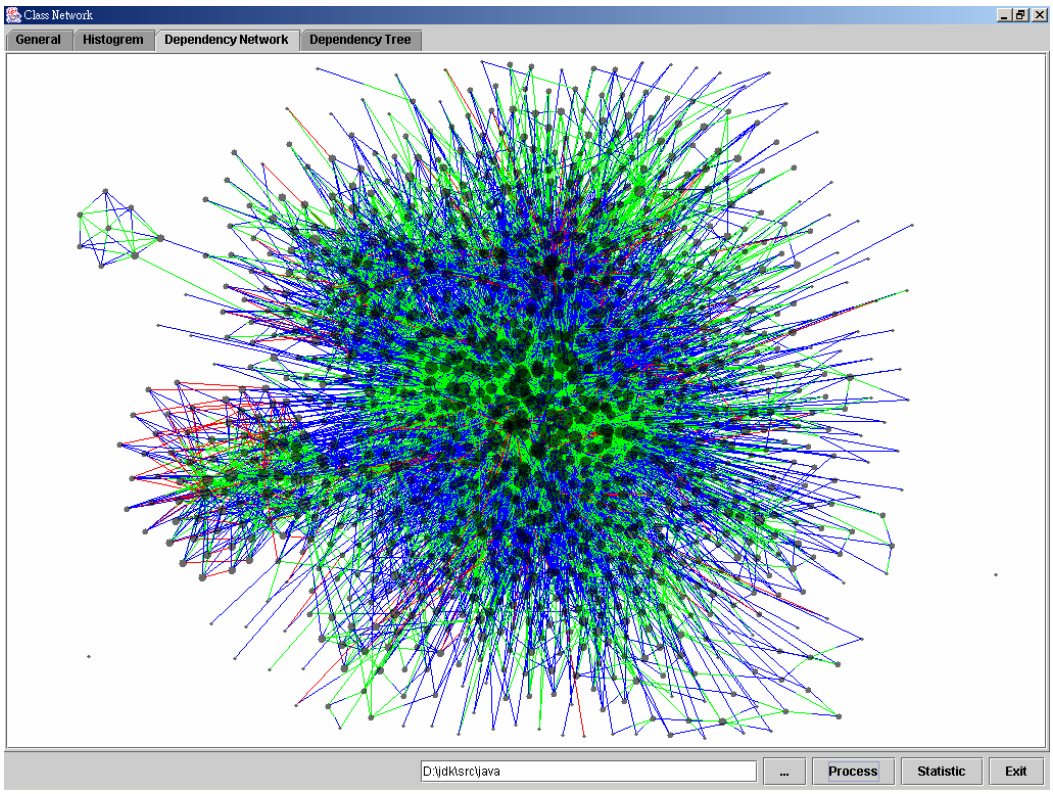


Figure 9. The dependency network among Java classes of Java package "java"

Like software architecture, "component" is another very popular word in software engineering but it does not have a universally accepted definition either. Our research does not provide a strict definition of the term; we intentionally keep this concept at its highest level of abstraction. A component can be a subsystem, a user, a physical object, a class or an object in OO language, or a more specific component in CORBA, Java beans, Microsoft's ActiveX or COMA. The links or the dependency relationships between components are also kept at an abstract level, they could be one object calling a method in another object, a data exchange protocol based on sockets, a pipeline, a function call, or even a user physically pushing a button.

Software architecture is one of the most important issues during the design phase of software development. But when making architecture decisions, designers usually focus on how to make the architecture satisfy the functional requirements and quality attributes rather than the maintenance demands. This may cause the architecture to be too specific or too complex. Then, when new requirements are added, it might be found that the existing architecture is not capable of handling the new requirements, or the change on even a small part will affect a vast portion of the system.

In existing software architecture research, the focus is on different high level views (Bass 1998, Bengtsson 1999, Hofmeister 2000, Albin 2003, Akşit 2002, Barroca 2000) of the system rather than the topological structure of the component dependency network (CDN), which is also called component integration network (CIN). The component dependency network is similar to the module architecture

view (Hofmeister 2000), while the concept of a module is similar to the concept of a component, even though the term "component" and the relationship between two components are more abstract in this research. We have abstracted all the relationships between two components and they are simplified as dependency, which means one component is dependent on the other component to function.

A CDN, which shows the components and the dependency relationships between those components, is one aspect of software architecture. However, the term software architecture covers a much broader range of concepts, so we will use a more specific term "component architecture" to refer the CDN of a software system in this thesis.

The component architecture presents a good view to check how a system realizes its functions through the cooperation of the components. It becomes more important when a system is subject to changes, because when we change a software system, we usually need to add new components, update the functionalities of existing components, add new connections between existing components etc. Using the CDN, we can quickly identify what other components could have been affected by the changed components (Wen 2004). However, CDNs for large systems can be very complex and in this situation, a statistical model may need to be introduced. In our research, we have discovered that the component dependency networks of large software systems are scale-free networks (Barabási 2002) and this result may inspire a new approach to the investigation of component architecture.

2.5 MDA (Model Driven Architecture)

Model Driven Architecture (MDA) is an approach proposed by Object Management Group (OMG) "for designing and building a component-based system that remains decoupled from the languages, platforms and the environments that are eventually used to implement the system" (Parr 2004).

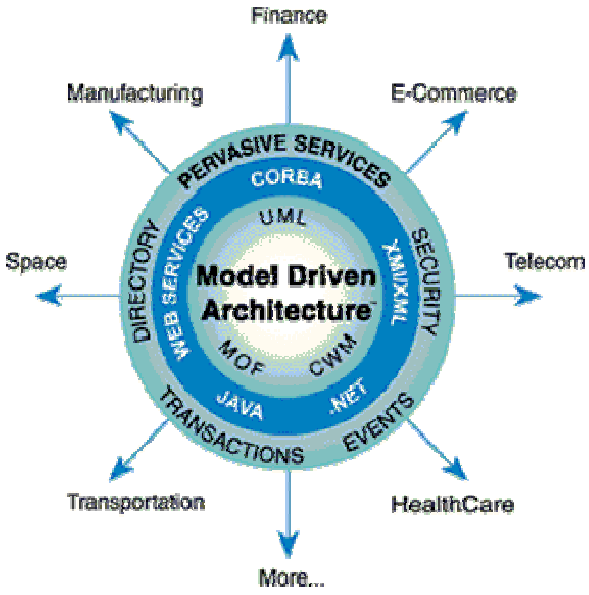


Figure 10. OMG's Model Driven Architecture

Figure 10 shows the OMG's version of the Model Driven Architecture (ORMSC 2001). As the figure shows, MDA has 4 layers and covers a very broad scope. The inner most layer includes Unified Modeling Language (UML), Meta-Object Facility (MOF), and Common Warehouse Meta-model (CWM). It is the core of MDA and the main purpose of this layer is to model a targeted system. The second layer includes XMI/XML, Java, .Net, CORBA and Web Services and the focus of this layer is to list the existing platforms that are supported by MDA. It is unclear why

the XMI/XML, which is used for storing models (see Figure 11), is also included in this layer. The third layer shows the *pervasive services* required by platforms in the previous layer. Finally, the last layer lists the areas for which the MDA technology can be applied to develop applications.

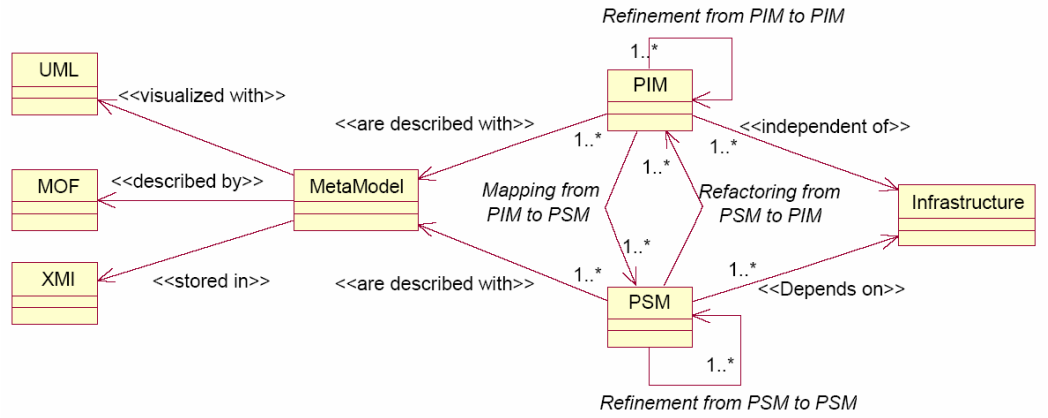


Figure 11. UML diagram summarizing the MDA development approach

One of the most important concepts of MDA is the distinction of the Platform Independent Model (PIM) and the Platform Specific Model (PSM). Using the MDA approach, a system is defined by a PIM that captures all the functional requirements and non-functional (e.g. security, performance) requirements. The PIM provides a completed description of the system's functional requirements without reference to any implementation or platform concerns. A PIM is stored in XMI and visualized with UML.

The ultimate goal of the MDA is to allow a system to move from the PIM specification to the completed system, but in reality this ambition is still some way off (Parr 2004). As a result, a PSM is required as a bridge to link the PIM and the final implementation. As the design phase shifts from the PIM to the PSM, the

focus moves from the system's business logics and functional requirements to implementation details that are embedded in the targeted environment. Once the PIM and PSM have been completed, automated tools can be used, theoretically, to generate the code for the components (OMG 2001). Figure 11 shows the relationship between the PIM and the PSM and how some of the other key technology standards, for example UML and XMI, are utilized by the MDA.

Even though the MDA covers a very broad scope of technologies, the core part is the PIM. The benefits of the introduction of a PIM include: Once a system is migrated into a new platform e.g. from COBRA 2.3 to COBRA 3.0, because the PIM is not specific to the platform on which the system is built, the same PIM can be reused. Also, during the initial design phase, "information technology serves the enterprise best when it focuses on business first, technology second" (Siegel 2001).

In this research, we have used GSE as the general platform to perform component-based software designs. The similarity between GSE and MDA is that both of them have addressed the importance of a software system's business logic or functional requirements; they both provide component-based design based on the system models. The difference between GSE and MDA is that MDA uses UML to visualize the models while GSE implements behavior trees to describe a system's behavior and present the models. Besides PIM, MDA also comprises platform specific models, but GSE has not introduced similar concepts so far. One of the advantages of GSE is that designs are natural properties that have emerged from the model (design behavior tree) so most of the procedures can be easily implemented by automated tools, but neither MDA nor UML provide equivalent features.

Generally, MDA supplies a formal procedure to build component-based software systems. The term "architecture" in MDA means more of the structure of the procedure by which MDA expects a software system to be built rather than the structure of a software system itself, which, however, is one of the major topics in this research.

Based on MDA and UML, a few approaches to handle software changes and software evolutions have already been proposed (Alanen 2002, Brassard 2002, Hearnden 2004). However, because our traceability is based on a totally new design approach, it is different from all the known models.

Even though GSE and MDA are two different approaches targeting component-based software design, they do not conflict. As MDA covers a broad scope of technologies while GSE provides a solid method to model system behaviors, it is possible for these two technologies to merge into a more powerful methodology.

2.6 HLA (the High Level Architecture)

The High Level Architecture (HLA) is a software architecture, which provides a framework for software simulation and can integrate different types of simulations together to form a larger scale simulation (Kuhl 1999). According to Kuhl, the HLA has been adopted by the United States Department of Defense (DoD) for use by all its modeling and simulation activities. That is also one of the reasons why the HLA has also been referred as a possible "green elephant" by some people (Tolk 2002).

Compared to the MDA, the HLA focuses more on the integration of different simulations and this model could be used as one implementation of MDA, and this feature attracts research on the integration of the MDA and the HLA (Parr 2004, Tolk 2002).

In the HLA, a simulation is referred as a *federate*, while the whole group of combined federates is called a *federation*, which is excused in one session called a *federation execution*. The supporting software is called the *Runtime Infrastructure* (*RTI*). The common object model for the data exchanged between federates in a federation is called the *Federation Object Model* (*FOM*).

The software component structure of the HLA is shown in Figure 12.

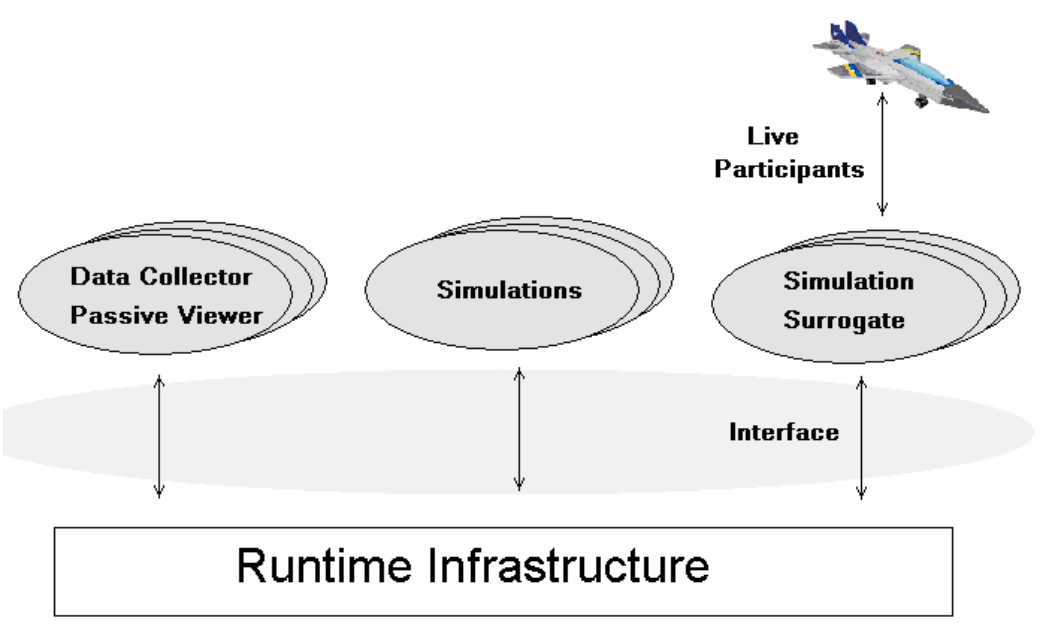


Figure 12. Software Components in the HLA

From Figure 12, we can see that all the simulations (federates), the data collectors and the simulation surrogates, which are also connected to the live participants, are

connected to the RTI through the standardized interfaces. The difference between a Simulation and a Data Collector or between a Simulation and a Simulation Surrogate is exists only in the user side, because they are for different purposes. From the RTI side, there is not much difference as they are all connected to the RTI through similar interfaces and they are all federates. The data model exchanged between a federate and the RTI is called a FOM which is prescribed through the Object Model Template (OMT). The OTM is the meta-model for all FOMs.

If we treat the HLA as a component architecture, a federate will be a component. In the HLA, there is no direct connection between any two federates. All the data are exchanged through the RTI. This feature makes each federate more independent and can be reused in different federations. Typically, a federate is larger and more complex than a common software component. However, if we abstract a component as a constructing unit that can be integrated with other units to form a system, there is no difference between a component and a federate.

GSE is a component-based software design approach and one of the important concepts in the GSE is the CIN (Component Integration Network), which is also referred as the component architecture. If we only look at the topological structure of a CIN, it is very different from that of the HLA in Figure 12, because a CIN is normally a network. However, if we treat a connection between two components in a CIN as a process of event posting and event receiving through a framework, a CIN can be easily fitted into the structure of the HLA. Therefore, the HLA is regarded as one of the possible implementation of the GSE.

Chapter 3 Genetic Software Engineering

Genetic software engineering (GSE) is a formal and systematic procedure to create a software design from its functional requirements (Dromey 2003). In contrast to conventional software engineering, which builds a software system that satisfies a set of requirements, GSE retrieves a software system out of its requirement set.

In traditional software engineering, there is no formal procedure to create a design from the functional requirements, so the designers have to make a design based on their personal experiences, intuitions, or the designs of other systems (usually from the same domain). The two obvious shortcomings of these design activities are the difficulty of proving the fitness of the design, and the lack of the repeatability. To overcome these difficulties in the traditional software engineering, GSE introduces a formal procedure to "translate" a software system's functional requirements into a design. In GSE, there are three major steps; the first step is to convert each individual functional requirement into a or a few corresponding requirement behavior tree(s) (RBT); then all the RBTs are integrated into a design behavior tree (DBT); the third step is to project out different design diagrams from the DBT. These diagrams reveal the software architecture, logic structure within each component and the interfaces of each component.

The most significant advantage of GSE is that, except the first step, the rest parts of the procedure are graph transformations based on restrict rules, so they can be easily realized by tools. The only issue is finding and correcting defects (especially domain knowledge related defects) – aspects of this cannot be easily automated. In addition, a fully syntactic and semantic rule-based system enables the integration of other supporting rules such as validation rules, traceability analysis rules and optimization rules. In other words, we can create automated tools to model check, validate, perform traceability analysis and optimize a software system designed by using GSE. The GSE approach could be a revolutionary idea in software engineering (Glass 2004). Brooks has claimed that "there is no silver" for software engineering (Brooks 1987), but GSE could be a ladder to climb over the "no silver bullet" brick wall (Dromey 2006).

Recently, based on GSE and behavior trees, different aspects of software engineering have been explored. Gonzalez-Perez has offered a comprehensive metamodel that formally describes the main areas of the behavior tree technique (Gonzalez-Perez 2005). Behavior Trees can be translated into other formal specification languages such as CSP (Winter 2004) and Symbolic Analysis Laboratory (SAL) (Grunske 2005), so that model checking can be performed. At the same time, an EBNF styled textual notationed semantic language (BTSL) has been developed (Colvin 2006). Behavior trees have also been explored to detect requirements defects in the early stage (Dromey 2005), and to model some non-functional requirements: for example, the safety requirements and the security requirements in embedded system (Zafar 2005) and an automatic process for Failure Modes and Effects Analysis (FMEA) (Grunske 2005). Other aspects of GSE and behavior tree studies, including software change impact analysis (Wen 2004),

architecture normalization (Wen 2005), requirement defects detection, large scale system case studies and different versions of software tools (Smith 2004) of GSE can be found at the GSE web site (GSE 2005).

In this thesis, traceability and optimization rules based on GSE have been studied and the results have been published in four papers (Wen 2004, 2005, 2007a, 2007c).

In this chapter, we will introduce the main concepts and the fundamental process of GSE in a relatively informal way to readers who may not be familiar to GSE. For a more formal and more complete description of GSE, please refer (Dromey 2003); for the latest development of GSE, please check (GSE 2005). This chapter is organized as following: In section 1, the central concept – behavior trees is introduced. The rules to integrate individual behavior trees into one large design behavior tree are introduced in section 2. In section 3, other design diagrams and the rules to project them out from the DBT are presented, and finally in the last section, a small case study of the Microwave Oven is used to illustrate the whole GSE process.

3.1 Requirement Behavior Trees

3.1.1 Behavior Tree Notation

The fundamental modeling notation in GSE is the Behavior Tree, a tree-formed graph composed of component-states and the logical relations, which is used to describe the behavior of a system and the composed components. One advantage

of behavior tree modeling is that most informally specified functional requirements, usually expressed in flexible natural languages, can be translated into a formal behavior tree in a simple and straightforward sentence-by-sentence, word-by-word basis, e.g., the sentence "whenever the door is open the light turns on" is translated to the behavior tree in Figure 13.

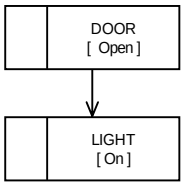


Figure 13. Whenever the door is open, the light turns on

From Figure 13, we can see the behavior tree includes two components "DOOR" and "LIGHT"; the "DOOR" in the "Open" state will cause the "LIGHT" in the "On" state, which matches the sentence "whenever the door is open the light turns on".

Now let us consider a more complex example, which includes a "CAR" component and a "TRAFFICLIGHT" component, "when the car approaches the traffic light, if the red light is on, the car will stop and if the green light is on, the car will go through". This sentence is translated into behavior tree in Figure 14.

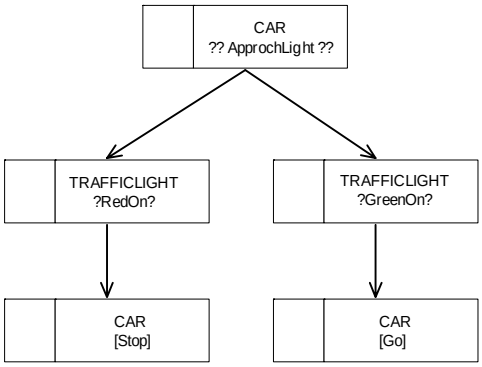


Figure 14. The behavior tree of the car and the traffic light.

The meaning of Figure 14 can be explained as: When the event of "CAR ApproachLight" is happened, if the condition of "TRAFFICLIGHT RedOn" is satisfied, the "CAR" will be in "Stop" state; if the condition of "TRAFFICLIGHT GreenOn" is satisfied, the "CAR" will realize the state of "Go". This example includes three different types of component states in a behavior tree, a [state realization], an ??event?? and a ?condition?. There are still other types of states for a component such as "<data output>", ">data input<" etc (Dromey 2003). However, because those state types are not used in the case study in this thesis, we will not provide further discussion of them.

3.1.2 Translate Functional Requirement into Behavior Tree

Requirements translation is the first formal step in the GSE design process, and it is the step that can only be processed manually[5]. Its purpose is to translate each natural language represented functional requirement, one at a time, into one or more behavior trees. Here we use a simple car-traffic light system, which has three functional requirements, to demonstrate this procedure. Supposing that the three requirements are:

1. When a car approaches the traffic light, the driver needs to check the lights.
2. If the light is red, the driver must stop the car.
3. If the light is green, the driver will drive the car go through the light.

[5] It is possible build tools to assist the process of translating the functional requirements into behavior trees, if the requirement specification is written following certain styles.

The three requirements are translated into three requirement behavior trees (RBTs) shown in Figure 15.

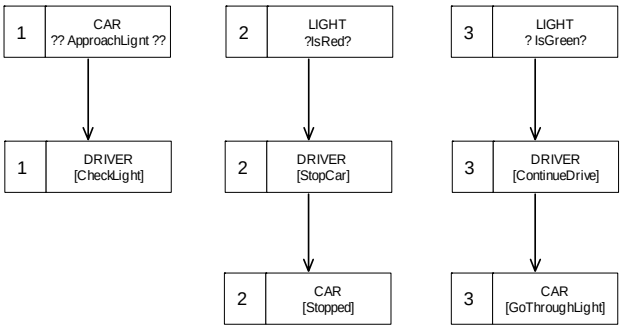


Figure 15. The directly translated requirement behavior trees of the car-traffic light system

It is not difficult to go through the requirements and the corresponding behavior trees, and find out that they are well matched. The numbers (1, 2, and 3) in the behavior trees are tags that are used to trace each individual piece of behavior back to the original functional requirement.

3.2 Integration of Requirement Behavior Trees

After requirements translation has been completed, each individual functional requirement is translated to one or more corresponding RBTs. We can then systematically and incrementally construct a design behavior tree (DBT) that will satisfy all its requirements. A formal description of the integration rules requires the Precondition Axiom and the Interaction Axiom (Dromey 2003). Here we will use an informal way to explain the RBT based requirement integration.

Each behavior tree must have a root node; the root node is actually the precondition for the behavior described by the remaining part of the behavior tree. If the root

node of a tree appears somewhere in other behavior trees, it means the behaviors of other trees may satisfy the precondition of the first tree and then the first tree may be integrated with the other trees. If a tree cannot be integrated with other trees, it may indicate the inconsistency or incompleteness of the requirement specification or simply some missing nodes in the requirement translation.

If we check the three behavior trees in Figure 15, we find none of the three root nodes appears in any other trees. The reason is for the second and third behavior trees, we have missed the implied precondition node "DRIVER-[CheckLight]". To add the missed nodes, the new behavior trees are shown in Figure 16. The "+" sign means these behaviors are implied in the functional requirements. In GSE "-" sign means the behaviors are missed in the functional requirements.

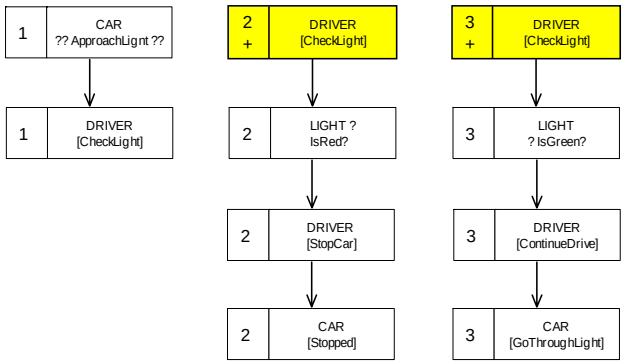


Figure 16. The requirement behavior trees of the car-traffic light system with the implied nodes.

Checking the behavior trees in Figure 16, it is found that the root nodes of the second and the third tree are matching a node in the first tree, so the second and the third trees can be integrated with the first tree using the root node. The integrated tree is shown in Figure 17.

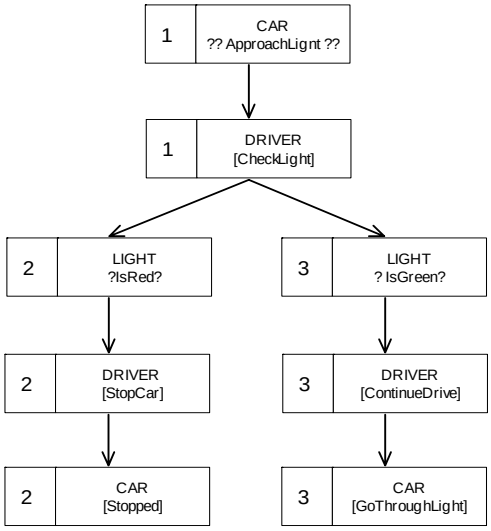


Figure 17. The integrated behavior tree of the car-traffic light system.

A design behavior tree (DBT) is the problem domain view of the "shell of a design" that shows all the states and all the flows of control (and data), modeled as component-state interactions without any of the functionality needed to realize the various states that individual components may assume. It has the genetic property of embodying within its form many emergent properties of a design, including (1) the component-architecture of a system, (2) the behaviors of each of the components, and (3) the interfaces of each of the components in the system. Besides the three list properties, many other properties such as safety and security concerns (Zafar 2005), behaviors in failure modes (Grunske 2005) can also be investigated from a DBT. However, in this thesis, we focus on only the three more general properties: component architecture, component behaviors and component interfaces; the three different properties are visualized by three types of design documents: a component integration network (CIN), component behavior trees (CBT), and component interface diagrams (CID) introduced in the following subsections.

3.3 From Design Behavior Tree to Other Design Diagrams

Once the design behavior tree (DBT) is constructed, it provides a formal and full *problem domain* view of the targeted system. The next step in GSE is to project out different design diagrams from the DBT. There are three types of design diagrams. The first is called component interaction network (CIN), also called component dependency network (CDN), which presents an architecture view of the system on the component level. The second is called a component behavior tree (CBT), which is a behavior tree of one particular component; it shows the internal logical structure of a component. The last type is called component interface diagram (CID), which shows all the interfaces of a component and what other components will call which interfaces and what other components are called by those interfaces.

One of the most interesting properties of these design diagrams is the rules to project them out from a DBT are clear and distinct. Once a DBT is fixed, the design diagrams projected from that DBT are fixed and the procedure of projecting from a DBT to the design diagrams can be implemented by automated tools.

3.3.1 Component Interaction Network

For a software system, the software architecture is one of the most critical issues. According to Bass software architecture is defined as "The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and

the relationships among them" (Bass 1998). In GSE, a component integration network (CIN), which shows the integration (or dependency) relationships between all the components in a software system, is one of the structures of a software system, and it can be treated as a view of software architecture. We call it component architecture and may also be referred as software architecture in this thesis when no confusions are made.

In the DBT representation, a given component may appear in different parts of the tree in different states (e.g., the CAR component may appear in the [Stopped] state in one part of the tree and in the [GoThroughLight] state in another part of the tree). We need to convert a DBT to a component-based design in which each distinct component is represented only once, the same as the integration relationship between an ordered pair of component. A simple algorithmic process may be employed to accomplish this transformation from a tree into a network. Informally, the process starts at the root of the design behavior tree and travels downwards through all the child nodes (it is insignificant whether we use the depth first approach or width first approach). Whenever a new component is reached during the traversal process, that component will be drawn in the CIN. Similarly, if a new connection between two different components is reached, that connection will also be drawn in the CIN. Generally, a connection from component A to component B is treated as a different connection from component B to component A. After every node in a DBT is reached, the corresponding CIN is also created. The CIN emerged from the DBT in Figure 17 is shown in Figure 18.

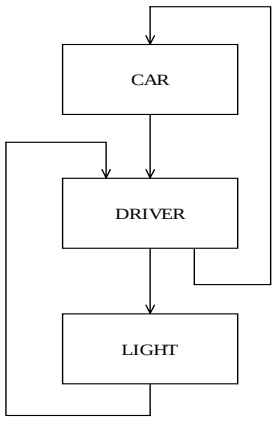


Figure 18. The component integration network (CIN) of the car-traffic light system

Comparing a CIN with architecture presented in other architecture description language (ADL) such as Rapide (Luckham 1995), Wright (Naumovich 1997) and UniCon (Shaw 1995), people may argue that the information shown in a CIN is too simple. The meaning of an arrowed connection in a CIN is not clear; is it a data flow, a control flow, a connector defined in Wright, a channel or a method call? Our answer is that in GSE, we try to model a system in a most abstract way. The concepts of a component and a connection between two components are kept in the most abstract form. Therefore, if there is a connection between component A and component B, what we can say is that component A needs to be integrated with component B or component A is dependent on component B in the system. In spite of the simplicity of a CIN, it still provides sufficient information for dependency analysis and change impact analysis.

One reason for GSE to select a very abstract form to model a system is that it seeks to provide a platform independent model (PIM), in the sense introduced in the model driven architecture (MDA 2006). In GSE, a component can be a hardware device, a traditional component in CORBA, an object in OO system, a federate in

the HLA (Kuhl 1999), or an external system; in the same way, a connection between two components can be a physical connection between two hardware devices, a socket connection between a client and a server, a channel, a process of event submitting and event subscribing, or even a method call in an OO system. Therefore, the focus of GSE is on the modeling of the functional behaviors and the business logics of a system. Further, with additional platform specified information, the models can be mapped into different platform specified models (PSM). One of the future research topics is to investigate how to map the GSE modeling into the implementations for different platforms. To achieve this goal, one possible approach is to study the possibility to translate a CIN into other ADLs.

3.3.2 Component Behavior Tree

In the design behavior tree, the behavior of individual components tends to be dispersed throughout the tree (for example, see the CAR component-states in Figure 17). To implement components that can be embedded in, and operate within, the derived component interaction network, it is necessary to "concentrate" each component's behavior. We can achieve this by systematically projecting each component's behavior tree (CBT) from the design behavior tree. We do this by simply ignoring the component-states of all components other than the one we are currently projecting. The resulting connected "skeleton" behavior tree for a particular component defines the behavior of the component that we will need to implement and encapsulate in the final component-based implementation.

To illustrate the effect and significance of component behavior projection we show the projection of the CAR component in Figure 19. Component behavior

projection is a key design step in the solution domain that needs to be done for each component in the design behavior tree.

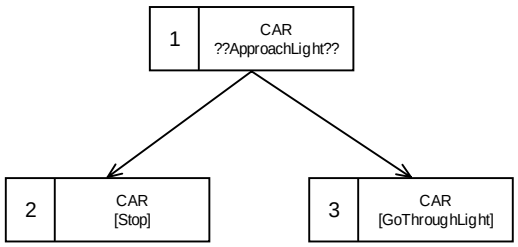


Figure 19. The component behavior tree (CBT) of the CAR component

A component behavior tree shows the behavior, the functional capacity and the logics of the functions of a component. For example, from Figure 19, we know the CAR component can raise the event of ApproachLight and can be in states of Stop and GoThroughLight; we also know that the state of Stop can only be realized after the event of ApproachLight has been raised[6].

Because the car-traffic light system is an over-simplified example that is only being used to explain the process rules of GSE, the CBT of component CAR doesn't include much information. However, for a more complex component, a CBT can be very helpful to understand the component. In GSE, a component can be treated as a state machine and the CBT is actually a state diagram drawn in a tree form. An important issue is that a CBT is not drawn based on the intuition of the designers but based on the behaviors described in the functional requirements. For example,

[6] In this thesis, we have used the original syntax of GSE, which does not explicitly express current threads or alternative threads. In Figure 19, from the DBT in Figure 17 we know that the threads of "CAR [Stop]" and "CAR [GoThroughLight]" are alternative threads, but this point has not been marked. The latest version of GSE notation has improvement in this point.

if system requires a component of STACK, what is the state diagram of the STACK, and what functions may it provide? Based on common knowledge, we know that this STACK component must provide a "push" function and a "pop" function, and a "pop" function can only be called after a "push" function. However, does this STACK component require a "clear" function or "check capacity" function? In GSE, because a CBT is a natural mapping from the design behavior tree, all the required functions as well as the order and logic of those functions are retrieved from system's behaviors of the requirements and there is no redundancy or missed functions unless there are defects in the functional requirements.

3.3.3 Component Interface Diagram

A component interface diagram (CID) shows all the interfaces of a component and what other component will call these interfaces and through these interfaces, what other components will be called.

To project out a given component's CID from the DBT, firstly, highlight all the state nodes of the component in the DBT; then the list of states in these nodes actually is the list of interfaces of the given component. For each interface of the component, at first we identify all the nodes in the DBT that correspond to this component with this interface, and then we list those nodes' parent nodes and child nodes. A component within the parent node set is one of the components that calls the targeted component's given interface and it is the same as that of a component within the child node set which is called by the targeted component's given interface.

For example, let's consider the component DRIVER in the DBT in Figure 17. There are three nodes of DRIVER and the associated interfaces are [CheckLight], [StopCar] and [ContinueDrive]. Therefore, the set of DRIVER's interfaces is {[CheckLight], [StopCar] and [ContinueDrive]}. For interface "DRIVER: [CheckLight]", the parent node is "CAR: ??ApprochLight??" and the child nodes are "LIGHT: ?IsRed?" and "LIGHT: ?IsGreen?", so the component that "calls"[7] "DRIVER: [ChedkLight]" is component CAR and the component called by this interface is component LIGHT. Finally we can draw the CID as Figure 20.

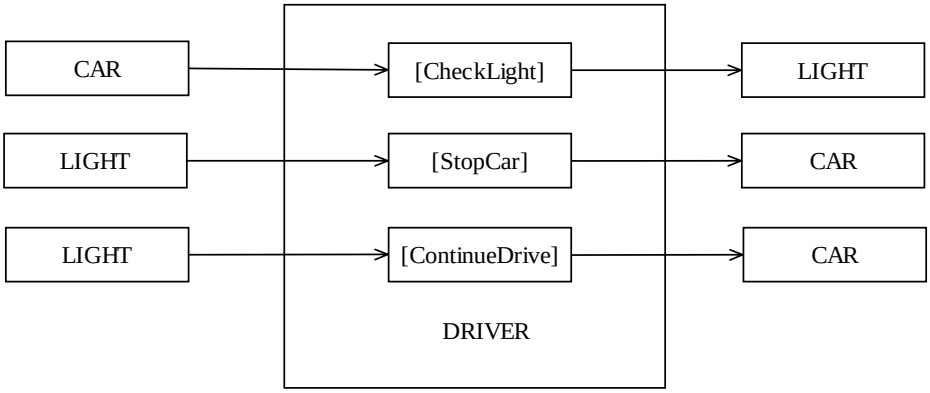


Figure 20. The component interface diagram(CID) of component DRIVER projected out from the light-car system

3.4 Microwave Oven Case Study

In this chapter, a microwave oven system is presented. The original microwave oven case study has been published in (Shlaer 1992), it then has also been used to explain the GSE process (Dromey 2003), a traceability model (Wen 2004), and architecture

[7] For convenience, here we use the term "call", which is brought from the style O-O design. Actually, the "call" here could mean "message passing" or "pass control to" etc. depending on the platform for the system to be implemented.

normalization (Wen 2005). In this thesis, this case study will be explored several times to explain different aspects of our research.

3.4.1 The Requirements

The original microwave oven system includes 7 requirements:

Table 1. The original 7 requirements of the Microwave Oven

- R1. There is a single control button available for the user of the oven. If the oven is idle with the door is closed and you push the button, the oven will start cooking (that is, energize the power-tube for one minute).
- R2. If the button is pushed while the oven is cooking it will cause the oven to cook for an extra minute.
- R3. Pushing the button when the door is open has no effect (because it is disabled).
- R4. Whenever the oven is cooking or the door is open the light in the oven will be on.
- R5. Opening the door stops the cooking.
- R6. Closing the door turns off the light. This is the normal idle state, prior to cooking when the user has placed food in the oven.
- R7. If the oven times-out the light and the power-tube are turned off and then a beeper emits a sound to indicate that the cooking is finished

3.4.2 Behavior Trees

In GSE, the first step is to translate each functional requirement into one or more

corresponding requirement behavior tree(s). The translation of R3, R6 and R7 are shown in Figure 21 and Figure 22. To save paper space, the other requirement behavior trees, which can be found in (Dromey 2003), will not be reprinted here[8]. Because component OVEN is treated as system level component, the states of it are highlighted in double line rectangles, but there is no difference for the following processes.

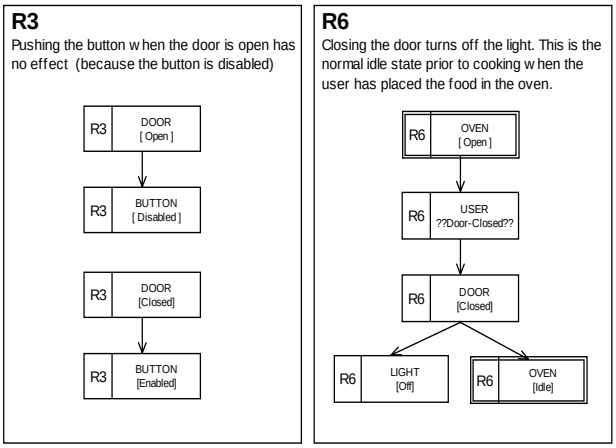


Figure 21. The requirement behavior trees for requirement R3 and R7

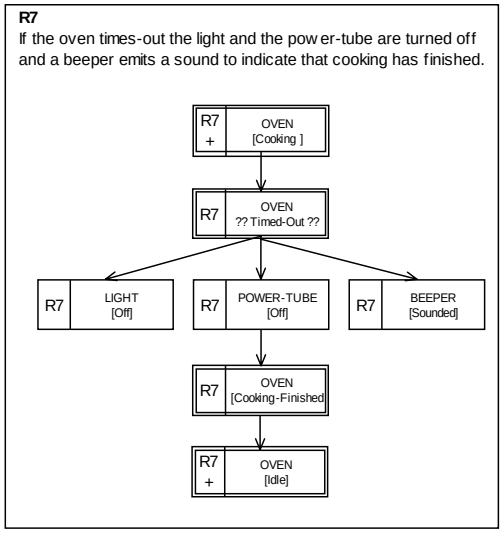


Figure 22. The requirement behavior tree of requirement R7

[8] Because of different refinements, some of the RBTs shown in this thesis are slightly different from that in Dromey's paper (2003). However, this difference will not affect the research results of this theis

After all the functional requirements are translated into RBTs, those RBTs can be integrated into one design behavior tree, shown in Figure 23.

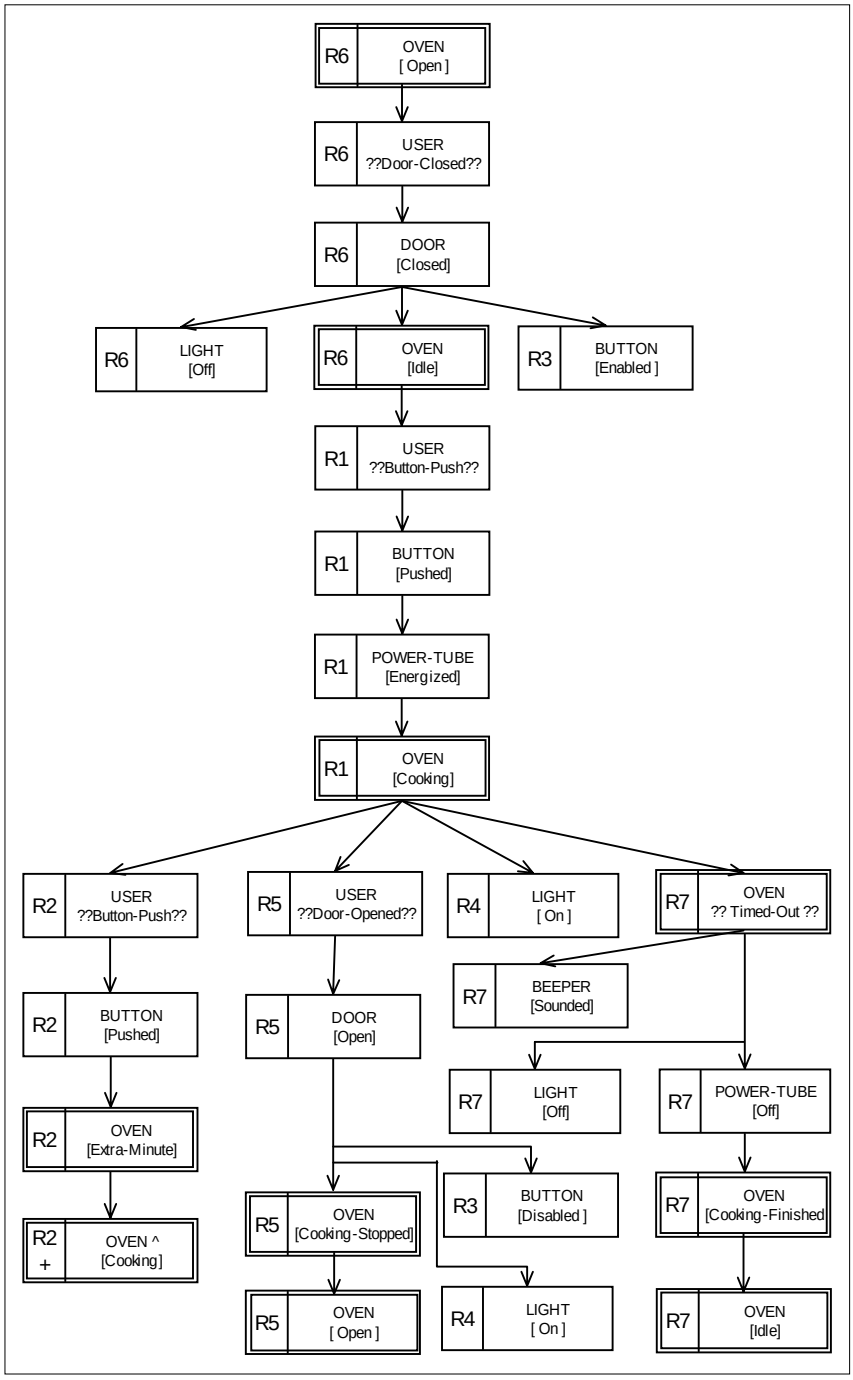


Figure 23. The design behavior tree (DBT) of the Microwave Oven System

3.4.3 CIN and Other Component Diagrams

From the DBT in Figure 23, the component interaction network (CIN) is projected out in Figure 24 using the process described in section 3.3.3.

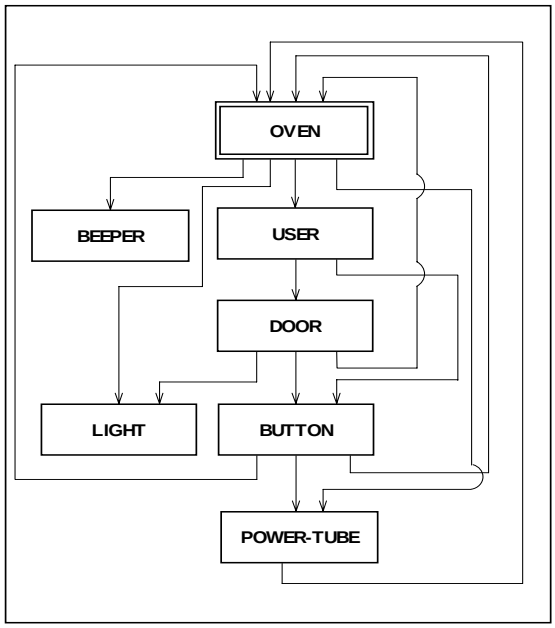


Figure 24. Component Interface Network (CIN) of the Microwave Oven System

From Figure 24, it can be found that the component USER depends[9] on the component DOOR and component BUTTON, because the "User" needs to push the "Button" and close or open the "Door". The component OVEN, which is drawn in doubled border, is dependents on the component USER because the status of the "Oven" determines what the "User" can do. Generally speaking, the CIN as the architecture of the system at the component level reflects the functional

[9] In this thesis, the term of "depend" is used to express the abstract relationship between two components, which means a component needs the existence of another component so the first component's functionality can be integrated into a system. In the above example, the term of "depend" can be replaced by a more specific term such as "control".

requirements.

The component behavior tree (CBT) of the component OVEN is projected out from the DBT and shown in Figure 25.

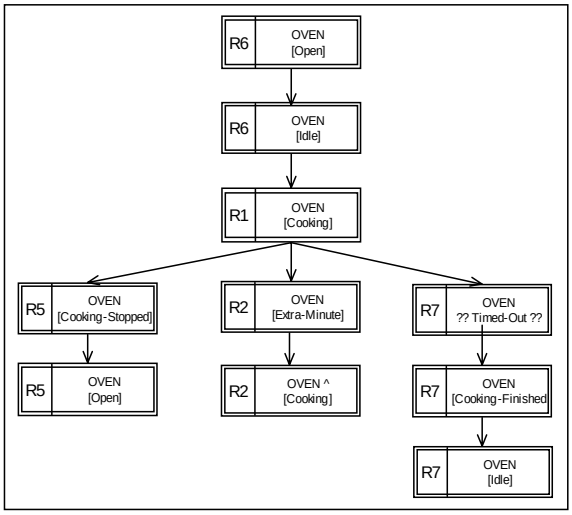


Figure 25. The component behavior tree (CBT) of the component OVEN

From Figure 25, we can see that the component OVEN starts from the "Open" state and then can change to the "Idle" state. Then from the "Idle" state, it can change to the "Cooking" state. From the "Cooking" state, it can change to "Cooking-Stopped" state etc. From this diagram, the behavior and the logical relationship of the different states of the component OVEN is clearly and formally visualized. Also, from this diagram, we can identify some missing requirements. For example, when OVEN changes from "Idle" state to "Open" state, based on common knowledge, we know it should be able to change back to "Idle" state directly. However, this state change path is not in Figure 25, so we know some functional requirements must be missed in the original requirements. Requirement defect detection is not the focus of this thesis, but to add the missing requirement is used to illustrate our proposed traceability model in later chapters.

The component interface diagram (CID) of the component OVEN, which is projected out form the DBT in Figure 23, is show in Figure 26.

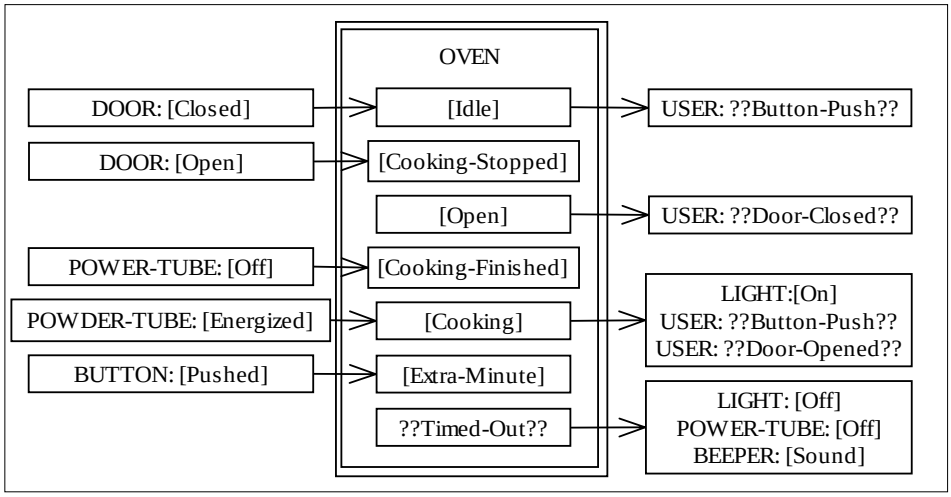


Figure 26. The component interface diagram (CID) of the component OVEN.

Figure 26 shows all the interfaces of the component OVEN and for each interface, what other components (in what state) will "call" it and what components will be called in this interface. Usually, in the software design phase, the designer needs to consider several components and their integration at the same time. However, during the development phase, it will be better for the programmer to only deal with one component at one time. Therefore, the question is how to isolate all the features of one component from the inter-related design diagrams so it can be well developed and also can be perfectly fitted back into the whole system is always a big problem, especially for large systems where designers and developers may belong to different groups. The CID and DBT that are directly projected out the DBT provide a very good solution.

Chapter 4 From Requirement Changes to Design Changes

The ideal we seek when responding to a change in the functional requirements for a system is that we can quickly determine (1) *where* to make the change (2) *how* the change affects the architecture of the existing system (3) *which* components of the system are affected by the change (4) and, *what* behavioral changes will need to be made to the components (and their interfaces) that are affected by the change. Because a system is likely to undergo many sets of changes over its service time, there is also a need to record, manage and optimize the system's evolution driven by the change sequence.

The change problem is complicated because requirements changes are specified in the problem domain, whereas the design response and the implementation changes that need to be made are in the solution domain. Requirements and design representations vary significantly in the support they provide for accommodating requirements changes. An important way of cutting down the memory overload and difficulties associated with making changes is to use the same representation for requirements and the initial design response to the change.

Based on the concepts of behavior trees and GSE, a traceability model, which uses behavior trees as a formal notation to represent functional requirements, is proposed to reveal change impacts on different types of design constructs (documents) caused by the changes of the requirements. The proposed model introduces the concept of edited design documents and evolutionary design documents that record the change history of the designs. From these documents, any version of a design document as well as the difference between any two versions can be retrieved. An important advantage of this model is that the major part of the procedure to generate these evolutionary design documents can be supported by automated tools (Wen 2007a, Wen 2007c).

4.1 Introduction

Large software systems are subjected to changes and one major type of change is the change of user requirements. To map the frequent changes of user requirements (problem domain) to the design (solution domain) and to keep all the design documents consistent can be a difficult, tedious, and costly job. Traditional traceability analysis solutions have applied hypertext systems (Garg 1990, Conklin 1987, Trigg 1986 and Bigelow 1988) and relational database (Horowitz 1986 and Lock 1999) to build an environment in which all the software documents are linked into a web. In this web, if one document is changed, what other documents might be affected can be easily identified and browsed. Based on this approach and UML presentation, there are also many commercial systems that provide traceability such as IBM Rational Rose (IBM 2007), Telelogic DOORS (Eriksson 2005) etc. However, this kind of solution usually does not provide facilities to automatically update the

affected documents and usually requires manually defining the relationships between documents and manually keeping the whole set of documents consistent.

Here we use behavior trees to represent functional requirements. A behavior tree is a tree-like graph that can be used to describe individual functional requirements. After all the requirements are translated into their corresponding behavior trees, they can be integrated into a larger behavior tree, which is called a design behavior tree. The design behavior tree captures all the functional requirements and shows the relationships between those requirements. One advantage of using behavior trees to denote functional requirements is that it clarifies ambiguities, which are common in natural language described requirements. It also helps to identify conflicts and missing pieces in the original requirements (Zheng 2003). Another advantage of behavior trees is other design diagrams such as component architecture, component interface and component behavior trees can be projected out from it through mathematic rules (Dromey 2003). This process is called genetic software engineering.

Inspired by the unique features of behavior trees and method of genetic software engineering, we propose a new process to map requirement changes to design changes. The general concept is to create the new design behavior tree based on the changed functional requirements; then compare the new design behavior tree with the old design behavior tree by merging them together to create an edit behavior tree (EBT). The interesting part of the edit behavior tree is it includes both the information of the old requirements and of the new requirements and it also clearly marks which parts only exist in the old requirements, which parts are brought in by the new requirements and which parts are unchanged. From the edit behavior tree,

we can project out other edit design diagrams (architecture diagram, component behavior diagrams and component interface diagrams). Similar to the edit behavior tree, the other design diagrams also include edit information by marking the new, old and unchanged pieces in different colors or different printing styles. The edit information not only describes the change impact, caused by the change in functional requirements, in a visual and easy to understand way, but also helps the developer to adjust the implementation to match the new design.

One advantage of the proposed method is the rules used to compare two behavior trees and project out other diagrams from the edit behavior tree are defined at the syntactic level so much of the process can be automated.

4.2 The Traceability in GSE

GSE provides clear bi-directional traceability between the work-products of the design process (see Figure 27).

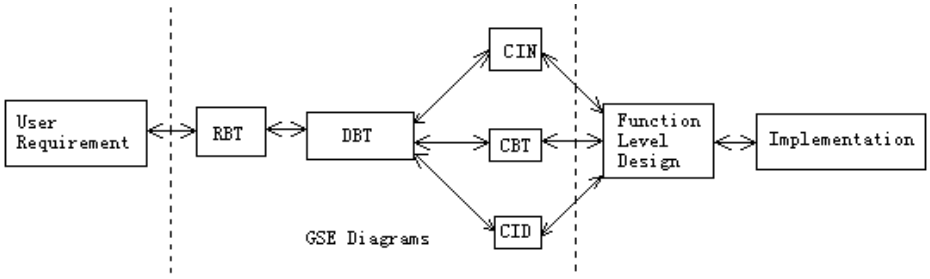


Figure 27. The traceability between the work-products of GSE

This traceability works as a bridge to connect functional requirements, the design documentation and the implementation. In traditional software engineering, most design documents are generated manually by the design team based on the

designers' understanding of the system and personal experience. In contrast with GSE, the first step, translating individual functional requirements into RBTs, needs an *understanding* of the system while the other steps have the potential to be either fully or at least partially automated. This traceability and the potential for automation of key steps provide important assistance for designing and implementing processes to support change of a set of functional requirements.

There are two great potential advantages for a fully automated bi-directional traceability link between the functional requirements and the design artifacts.

The first is to identify the defects in and/or optimize the original functional requirements. Once the design artifacts are created based on the original requirements, we can investigate the component architecture and other design diagrams. If there is any inconsistency or incompleteness in those diagrams, we can trace back to the defected functional requirements, which may be hard to detect by only studying the requirements. Zheng has done some research on this topic (Zheng 2003); and in the later chapter, we have proposed a method to normalize a software system's architecture.

The second advantage for the automated traceability is we can map the evolution of the design to the evolution of the functional requirements. Through the study and comparison of the evolution path between the design artifacts and evolution path of the functional requirements, we can identify which parts of the functional requirements have the greatest impact on the design artifacts and these results may help to select more reasonable evolution of the functional requirements.

4.3 Traceability Model

Consider a software system that has been designed based on a set of functional requirements. Once the requirements are changed, the problem is how to change the design to match the new requirements. Existing design methods, including GSE, do not provide a clear process, and supporting representations, for adjusting the design to accommodate the change in the functional requirements.

The present proposal addresses the problem of formalizing the impact of change. The output of the method is a set of edit design diagrams which show the impact of the changed requirements on the design. More specifically, the edit design diagrams not only show the new design, but also mark which parts are new in the design, which parts existed in the old design but have been removed and which parts are unchanged. Currently, the method is only suitable for projects originally designed by the GSE method, because GSE provides a systemic process to translate and integrate functional requirements into the design. However a similar concept may be applicable to projects designed using other methods.

4.3.1 The Procedure of the Traceability Model

The first traceability model is described in Figure 28:

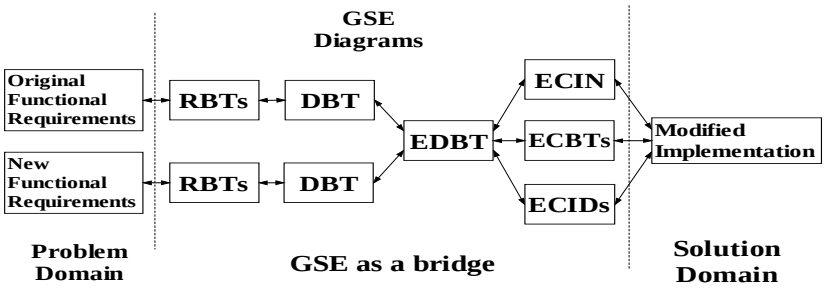


Figure 28. The first traceability model

To understand the formalization of change, suppose we have a design originally constructed using GSE. To map subsequent changes to the functional requirements onto the existing design (captured by the DBT), we use the following major steps:

1. From the changed requirements, we translate any new/additional requirements to behavior trees.
2. We then use requirements integration and editing of the old DBT to produce a new DBT that accurately reflects the changed requirements.
3. *The new DBT and the old DBT are merged to produce an Edit Design Behavior Tree (EDBT).*
4. Project out other diagrams such as ECIN (Edited Component Integration Network), ECBT (Edited Component Behavior Tree) and ECID (Edited Component Interface Diagram) from EDBT by *modified* GSE rules.

The procedure is similar to the original GSE procedure, but it introduces a very important step: that of comparing the old DBT and the new DBT and merging them into an EBT (the detail of the merging algorithm is described in the next section). The key point is that the EBT contains all the behaviors of the original DBT and new DBT and it also contains the edit information, which marks the change impact of the changes in the functional requirements.

The last step is to project from the EBT the other edit design diagrams: the ECIN (edit component integration network, which shows the change impact on the architecture), the ECBTs (edit component behavior trees) and ECIDs (edit component interface diagrams). The method of projection is similar to that used in GSE except it also maintains the edit information. Details of the projection rules are discussed in the following sections.

4.3.2 Algorithm to Compare and Merge Behavior Trees

The purpose of comparing the new DBT and the old DBT is to identify the changes, that is, to find out the new behaviors that are introduced into the new tree, the behaviors in the old tree but not in the new tree and the behaviors unchanged in the two trees. This information is stored in the EBT. As an example, suppose that T_1 and T_2[10], shown in Figure 29, are the old DBT and the new DBT respectively.

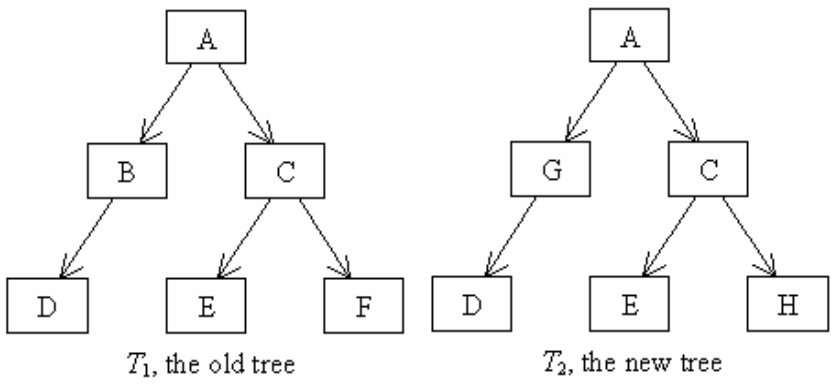


Figure 29. The old tree T_1 and the new tree T_2

[10] T_1 and T_2 are behavior trees, so each node is actually a component plus an associated state. However, to simply the example, we abstract them into a box with a single letter.

To compare T_1 and T_2 and generate the Edit Behavior tree, we use the following algorithm, which is based on a typical tree traversal algorithm (Knuth 1997a):

1. Start with the comparison[11] of the root nodes (in this example, node A). Because the root node exists in both trees, it is created in the edit behavior tree as an unchanged node.
2. Find the comparing node's child-node set in both trees. (In this example, the child-node set in the old tree is {B, C} and the child-node set in the new tree is {G, C}.
3. If a node exists in the old tree's child node set but not in the new tree's child node set, this node will be marked in the edit behavior tree as an old node. (In this example, B is such a node)
4. In the old tree, the sub trees under the old node will be generated in the EBT as old. (In this example, the node D under node B in T_1 is such a case)
5. If a node exists in the new tree's child-node set but not in the old tree's child node set, this node will be created in the EBT as a new node. (In the example, G is such a node)
6. In the new tree, the sub trees under the new node will be generated in the EBT as new. (In this example, the node D under node G in T_2 is such a case)
7. If a node exists in the child node sets of both trees, it will be generated in the EBT as an unchanged node. (In the example, the node C is such a case)
8. An unchanged node will be a new comparison node and the algorithm will go back recursively to step 2.

---

[11] In this algorithm, we assume the two trees have an identical root node. If the two trees have different root nodes, one possible solution is to add an artificial root in both trees or adopt more sophisticated algorithms.

The edit behavior tree T_e produced from T_1 and T_2 is shown in Figure 30. The new part in the tree is drawn with bold lines and the old part in the tree is drawn with dotted lines and the unchanged part is drawn in the normal style.

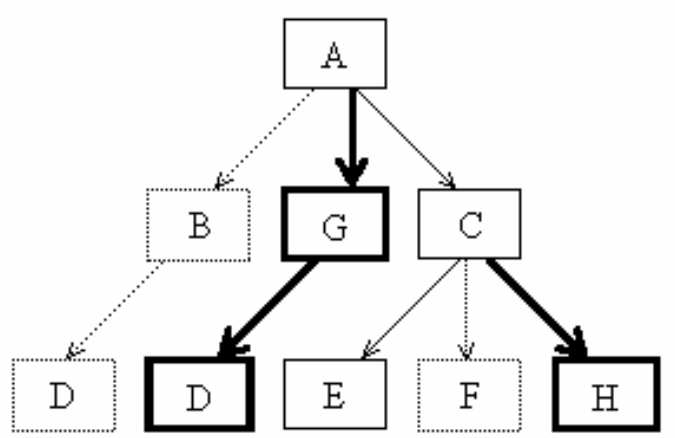


Figure 30. The edit tree T_e merged from T_1 and T_2

One interesting thing in Figure 30 is node D. It is both old and new, which means it should be an unchanged node. However, the algorithm cannot resolve this problem at this stage. In the next stage, when projecting other diagrams from the EBT, the true status of node D will be determined.

4.3.3 The Projection and Transformation Rules

The rules to project the edit design diagrams from an EBT are similar to the rules to project design diagrams from a DBT that have been introduced in previous chapters. The only difference is that the rules used for an EBT have to carry out the edit information.

As we have discussed before, during the process of projecting diagrams from a DBT, the DBT is decomposed into many atomic elements, while each element is

either a node (a state, a condition or an event) or a link, and each element maps to a corresponding part in the target diagram. When a design diagram (a CIN, a CBT or a CID) is projected (or in the case of a CIN, obtained by transformation) from a DBT, any atomic part in the design diagram can be traced back to a link (or several links) or a node (or several nodes) in the DBT. If the projection and transformation source is not the original DBT but the EBT, each atomic part in the design diagram will inherit the edit information from its counterparts in the EBT.

For example, with the EBT in Figure 30, because node *H* is marked as "new", in a design diagram, if a particular part is projected or transformed from node *H*, that part will also be marked as "new". The same rule applies to entities of "old" and "unchanged" status.

In addition to the straightforward mapping rule, there is one exception. The transformation from an EBT to the CIN or a CID can be a many-to-one projection. This means several nodes (or links) in the EBT may project and transform to one single part in the design diagram, just as a particular state of a component have more than one node in an EBT, but when the EBT is transformed to a CIN, these nodes will merge to a single state within a component projected behavior tree. Therefore, a single atomic part in an edit design diagram may have more than a single edit source in the EBT.

The rules to merge this different edit information turn out to be straightforward. Referring to Figure 30 again, there are two node D's, one is marked as "new", which means the node D exists in the new requirement and another is marked as "old", which means node D also exists in the old requirement. Because node D exists in both the original requirements and the modified requirements it must be treated as

unchanged in the edit diagram. From this simple analysis, we know that whenever an entity of "old" status merges with one of "new" status, it becomes "unchanged". Similarly, when "old" merges with "unchanged" it will be treated as "unchanged". For the case of "new" merging with "unchanged" it is also resolved as "unchanged". We may therefore summarize all the projection and transformation rules for dealing with edit information as follows:

1. "New" to "new".
2. "Old" to "old".
3. "Unchanged" to "unchanged".
4. "New" merged with "new" equals "new".
5. "Old" merged with "old" equals "old".
6. "New" merged with "old" equals "unchanged".
7. "New" or "old" or "unchanged" merged with "unchanged" equals "unchanged".

4.3.4 An Example

In Chapter 3, we used a simple example to explain the general concepts of GSE. If the functional requirements are changed, the following example will show how the change impact is captured and reflected in different design diagrams through the traceability analysis model.

Suppose a new component TIMER is introduced. The main functionality of TIMER is to timing the cooking state of OVEN. With the new component, the

original requirement 1, 2 and 6 can be changed as below: (the modifications to the three requirements are underlined).

Modified requirement 1: There is a single control button available for the user of the oven. If the oven is idle with the door closed and you push the button, the timer will be set to one minute and the oven will start cooking (that is, energize the power-tube)

Modified requirement 2: If the button is pushed while the oven is cooking it will cause the timer to add one extra minute

Modified requirement 7: If the timer times-out, the light and power-tube are turned off and then a beeper emits a sound to indicate that the cooking is finished.

Figure 33, Figure 32 and Figure 33 show the new requirements behavior trees of the modified requirement 1, 2 and 7 and the edit behavior tree (EBT) is shown in Figure 34. It was constructed using a tool that employs the rules described in previous sections. The "@" in Figure 34 indicates it is an integration node.

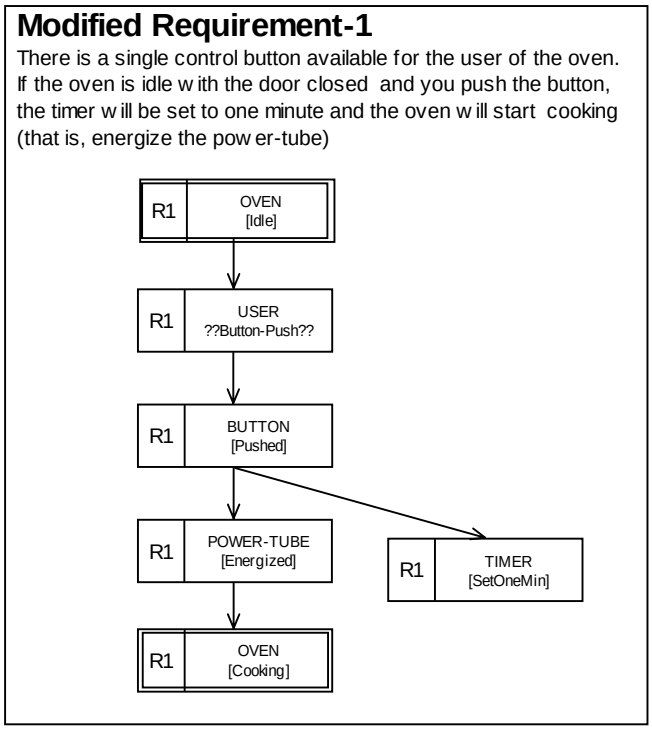


Figure 31. The RBT for modified requirement R1.

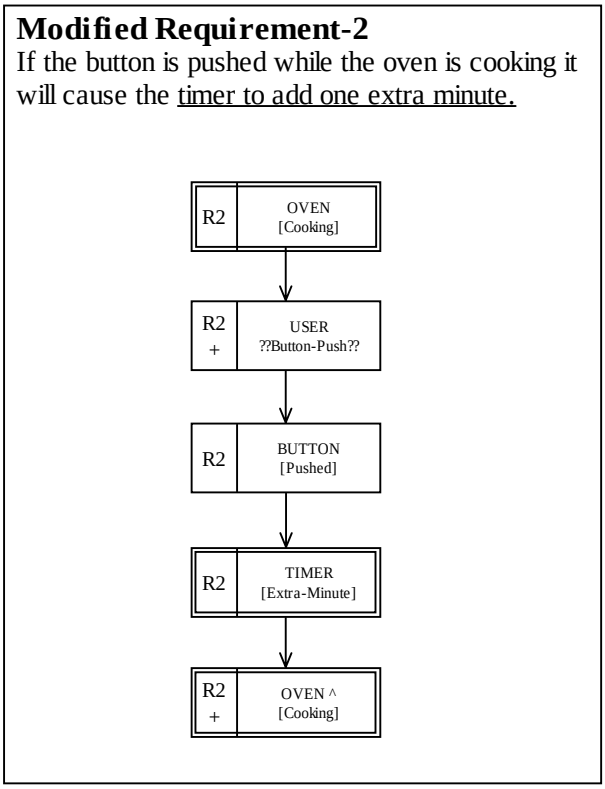


Figure 32. The RBT for modified requirement R2

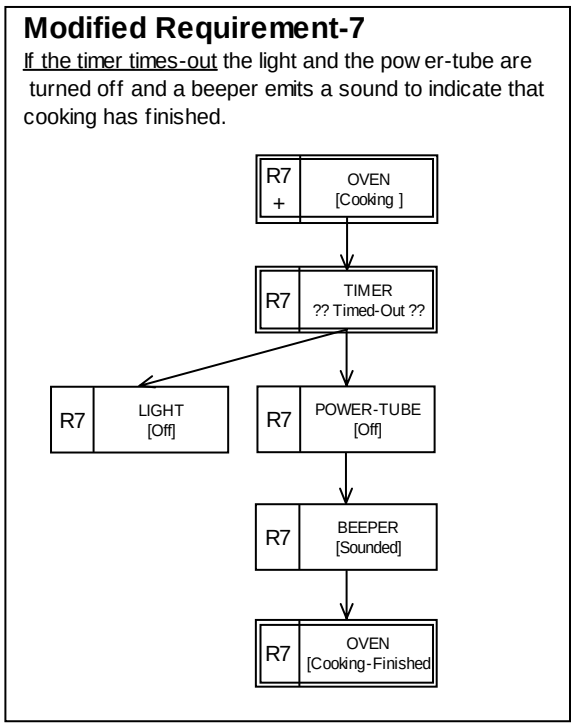


Figure 33. The RBT for modified requirement R7

In Figure 34, the new fragments of behavior are drawn in bold lines and filled with dark gray, the old fragments of behavior, which are not in the modified system, are drawn in light gray lines and the unchanged parts are drawn in the normal style. This diagram shows clearly the change impact of the modified requirements on the behavior tree. From the EBT, other diagrams (the ECIN in Figure 35, the ECID of OVEN in Figure 36 and the ECBT of OVEN in Figure 37) are projected. The edit component diagrams of component OVEN are shown below.

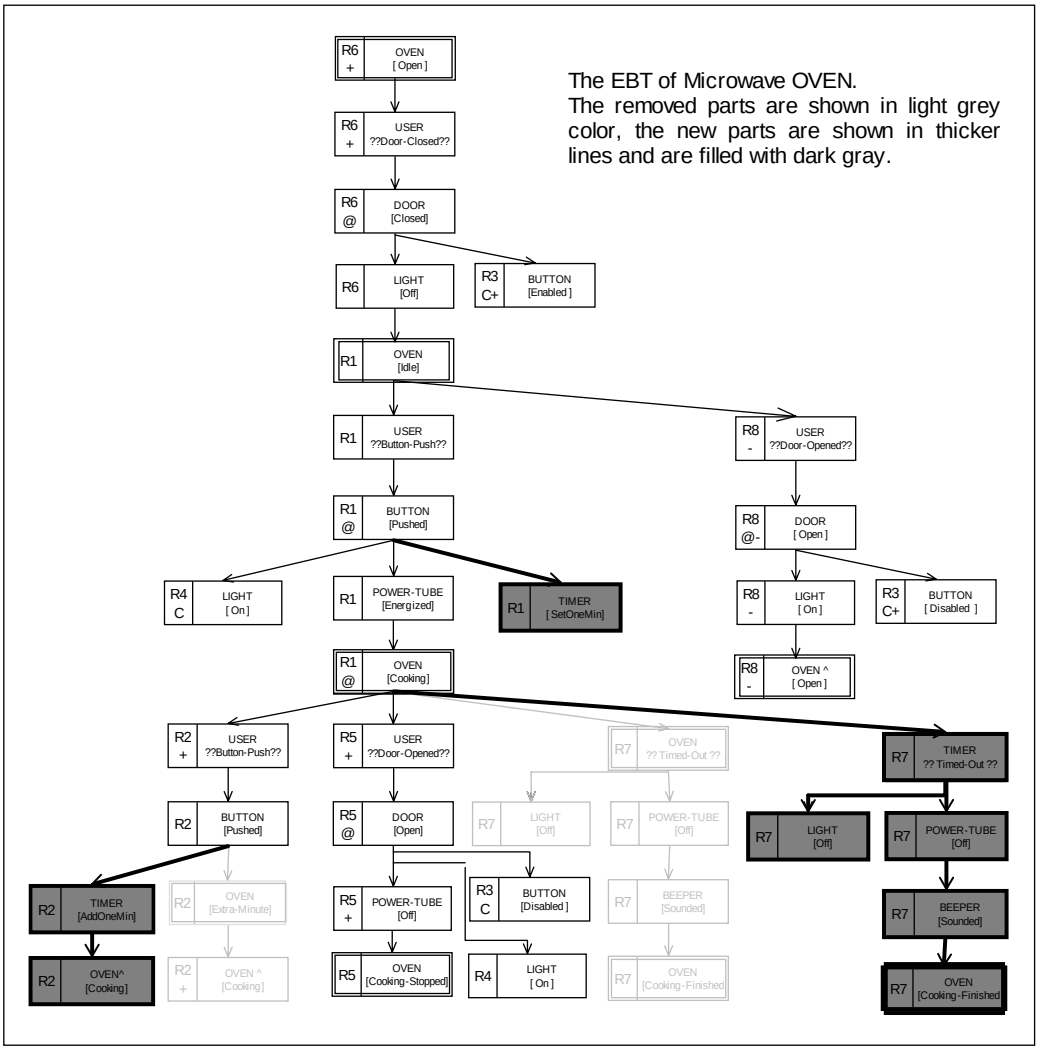


Figure 34. The edit behavior tree of the Microwave Oven System

From the ECIN (Figure 35), the change impact on the software architecture is clearly marked. Figure 35 shows that that several interaction relationships between the component OVEN and other components are removed and a new component TIMER is added as well as several component interaction relationships with TIMER.

Figure 36 is the ECID (Edit Component Interface Diagram) of the component OVEN. In this diagram, the new text is bolded and filled with dark gray and the old part is drawn in light gray. It shows that the interface ??TimeOut?? and

[Extra-Minute] are removed from OVEN component and the new component TIMER, which is called from the [Cooking] interface is added.

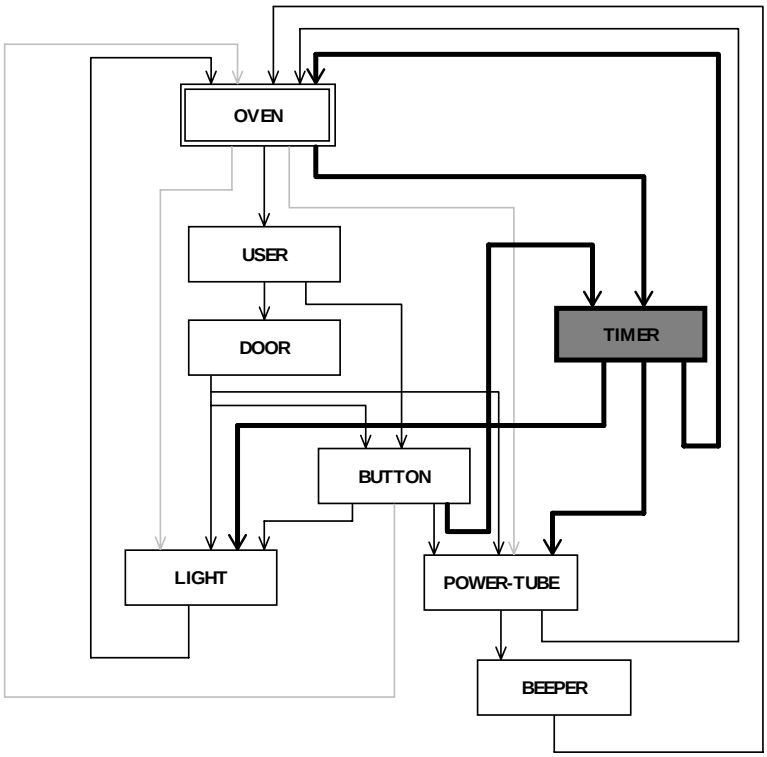


Figure 35. The ECIN of the new Microwave Over System

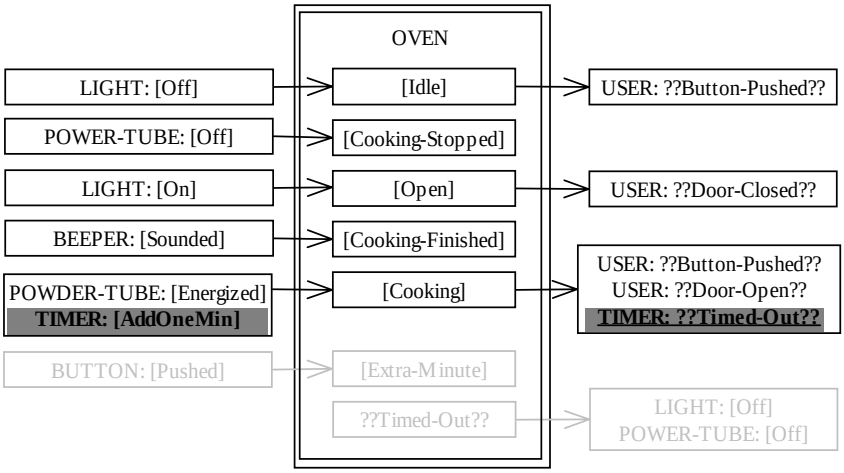


Figure 36. The ECID of the OVEN component

Figure 37 is the ECBT (Edit Component Behavior Tree) of the component OVEN.

This figure shows the change impact on its internal behavior.

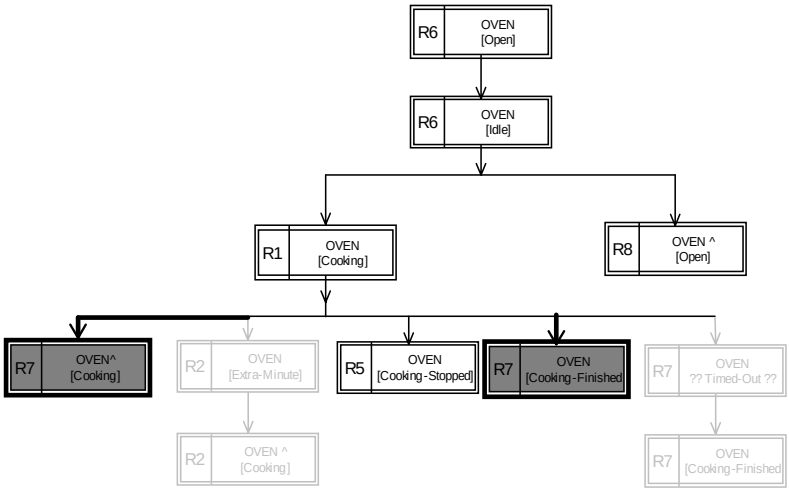


Figure 37. The ECBT of component OVEN

4.4 The Extended Traceability Model

In the previous sections, we have proposed a traceability model to map the changes from functional requirements to the design documents. In this section, the model is extended to handle multiple sessions of changes and to show the evolutionary procedure for handling the impact on design documents.

4.4.1 The Procedure of the Extended Traceability Model

The extended traceability model is shown in Figure 38. It is similar to the model in the previous section. However, the major difference is that the extended model can handle multiple sessions of changes; it can merge by comparing more than two different DBTs (each DBT has a unique version tag) and create an Evolutionary Design Behavior Tree (EDBT). And then from the EDBT, other evolutionary

design documents can be projected out. From those evolutionary design documents, we can project out the design documents of any version as well as the difference between any two versions.

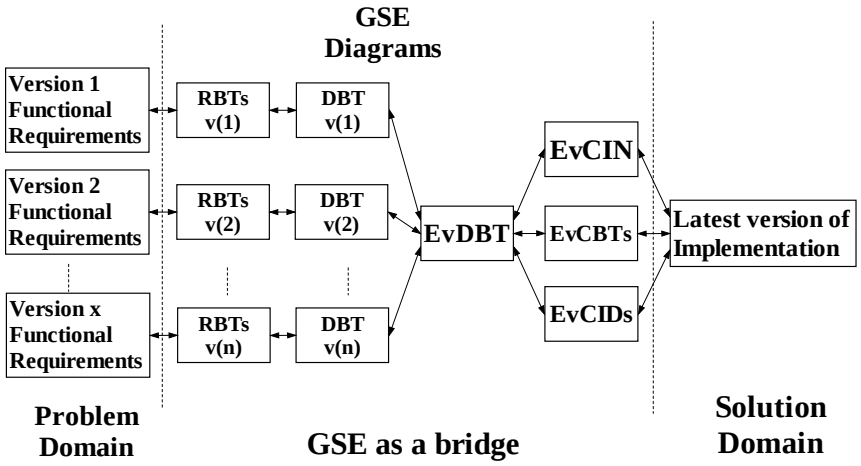


Figure 38. The extended traceability model

4.4.2 The Extended Tree Merging Algorithm

In section 4.3.2, we have introduced an algorithm to compare two DBT and merge them into a new tree called the EDBT. Now we modify that algorithm so it can compare multiple trees.

A tree is a collection of atomic items that are arranged according to a certain relative positions. An atomic item can be either a node or a link between two nodes. In a DBT, each atomic item is associated with a number of tags. A tag can be a requirement tag (such as R1, R2, …, R7 in the previous example) or a version tag (in this section, we mainly focus on the version tags). Let us reconsider the example shown in Figure 29. Suppose that the "old tree" T_1 is the DBT of version 1 and "new tree" T_2 is the DBT of version 2. Each atomic item in T_1 has attached a tag v_1

and each atomic item in T_2 is attached a tag v_2. When T_1 and T_2 are merged as T_e, the atomic items marked as old are only attached with tag v1, the atomic items marked as new are only attached with tag v2 and the atomic items marked as unchanged are attached with tag v1 and v2. The tree, which is called an evolutionary behavior tree, is shown in Figure 39.

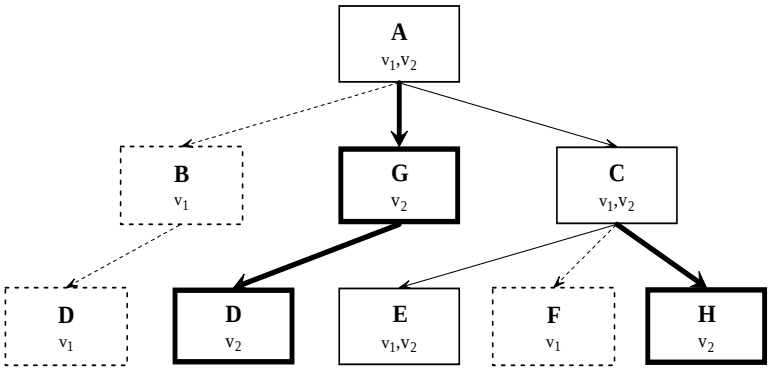


Figure 39. The EvDBT merged from T_1 and T_2

Consider a third version of DBT T_3 shown in Figure 40, and suppose we merge it with the EvDBT in Figure 39. The merging procedure is very similar to the procedure described in section 4.3.2. The only thing that needs to be mentioned is that when two nodes in two different trees are identical and are supposed to be represented as one node in the merged tree, the set of version tags associated with the node in the new tree is the union of the two version tag sets of the two nodes in their original trees. Finally, we generate a new EvDBT from shown in Figure 41.

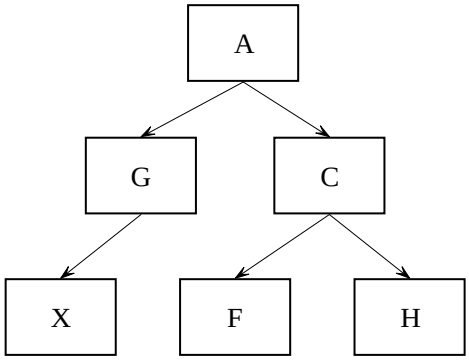


Figure 40. The third version T_3

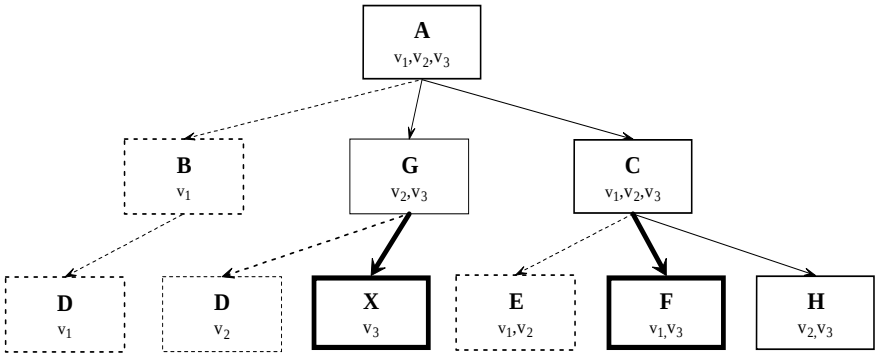


Figure 41. The EvDBT merged from T_1, T_2 and T_3

In Figure 41, the atomic items with both the latest version tag and the second latest version tag are printed in the normal style, the atomic items with the latest version but not the second latest version tag are printed in bold, which means that new parts are added in the latest version, and the atomic items without the latest version tag are printed in dotted line, which means that old parts are removed from the latest version. We will use this notation in the examples discussed later in the section.

In a tree, any link has two connected nodes, the parent node and the child node. For an EvDBT, the associated version tag set for a link is identical to that of its child node.

4.4.3 The Rules to Project out Evolutionary Design Documents

The rules to project out the edit design diagrams or evolutionary design documents from an EDBT or an EvDBT are similar to the rules to project out design diagrams from a DBT that have been introduced in previous chapters. The only difference is that the rules used for an EDBT or an EvDBT have to carry through the edit information or version information.

As we have discussed before, during the process of projecting diagrams from a DBT, the DBT is decomposed into many atomic elements, while each element is either a node (a state, a condition or an event) or a link, and each element maps to a corresponding part in the target diagram. When a design diagram (a CIN, a CBT or a CID) is projected out from a DBT, any atomic part in the design diagram can be traced back to a link (or several links) or a node (or several nodes) in the DBT. If the projection/transformation source is not in a normal DBT but in an EDBT or an EvDBT, each atomic part in the design diagram will inherit the edit information or version information from its counterparts in the design behavior tree.

During the procedure of projecting out design documents from an EDBT or an EvDBT, the edit information or version information is carried through. However, some of the projection is not simply a one-to-one mapping but many-to-one mapping. This means several nodes (or links) in the EDBT or EvDBT may project/transform to one single part in the design diagram, just as a particular component may have more than one node in a DBT, but when the DBT is transformed to a CIN, these nodes will merge to a single node to represent the

component. Therefore, a single atomic part in a design diagram may have more than one single source in the DBT. It is the same for an EDBT or an EvDBT.

The rules to merge the different edit information from an EDBT have been introduced in Section 4.3.3. For an EvDBT, the rule is even simpler. If multiple atomic items from an EvDBT are merged into one single part in an evolutionary design document, the version set associated with the merged part is the union of the version sets of the source atomic items.

4.4.4 An Example

We use the same microwave oven example that has been discussed in previous chapters and the previous section in this chapter. Minor changes have been made to make the example more focused on the traceability model rather than details of the specifications.

In this example, we will have three different versions of the functional requirements and each version has an associated DBT.

The first version of DBT is drawn based on the original 7 functional requirements (without the Requirement 8, which is added to make the system closer to completed). The version 1 DBT is shown in Figure 42.

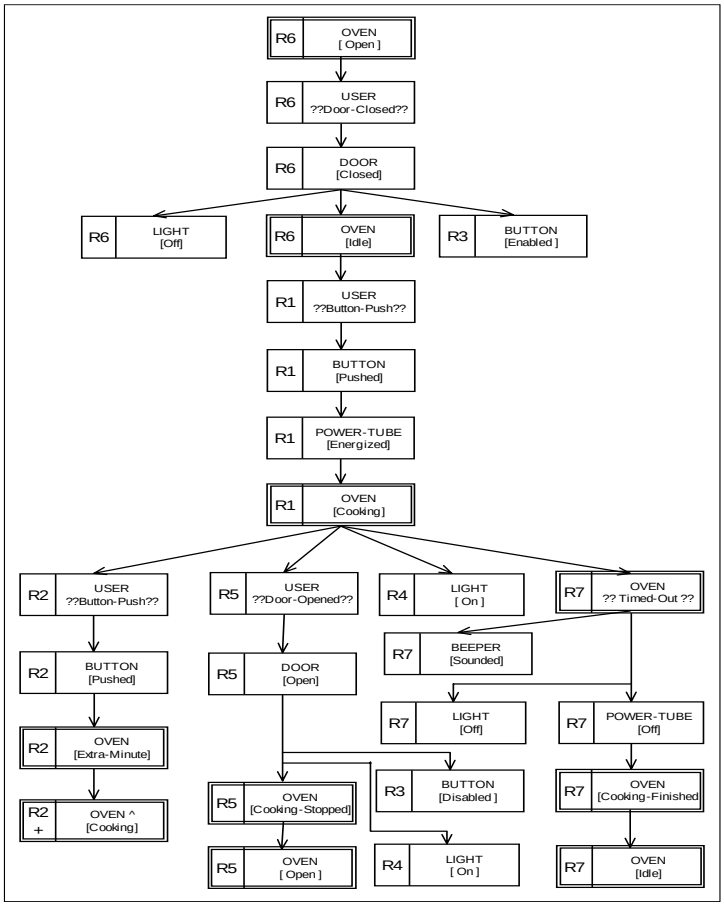


Figure 42. The first version of the DBT of the Microwave Oven System

Checking the DBT shown above, we find that there is one important requirement missing from the original requirements set. That is, when the OVEN is idle, what will happen if the USER opens the DOOR. Based on the common knowledge about the behavior of a typical microwave oven, it is not difficult to write the missing requirement:

Missed requirement 8: When the oven is idle, if the user opens the door, the door will be open, and the oven have the status open.

Then in that DBT we also notice that the two states OVEN:[Cooking-Stop] and

OVEN:[Cooking-Finished] that are not really necessary, so we can remove those two states from the new DBT to simplify the design. This simplification will not affect the functional requirements. When we integrate requirement 8 into the DBT and remove the two unnecessary states, we will have a new DBT. Merging this DBT with the original DBT in Figure 42 and we get the EDBT shown in [12].

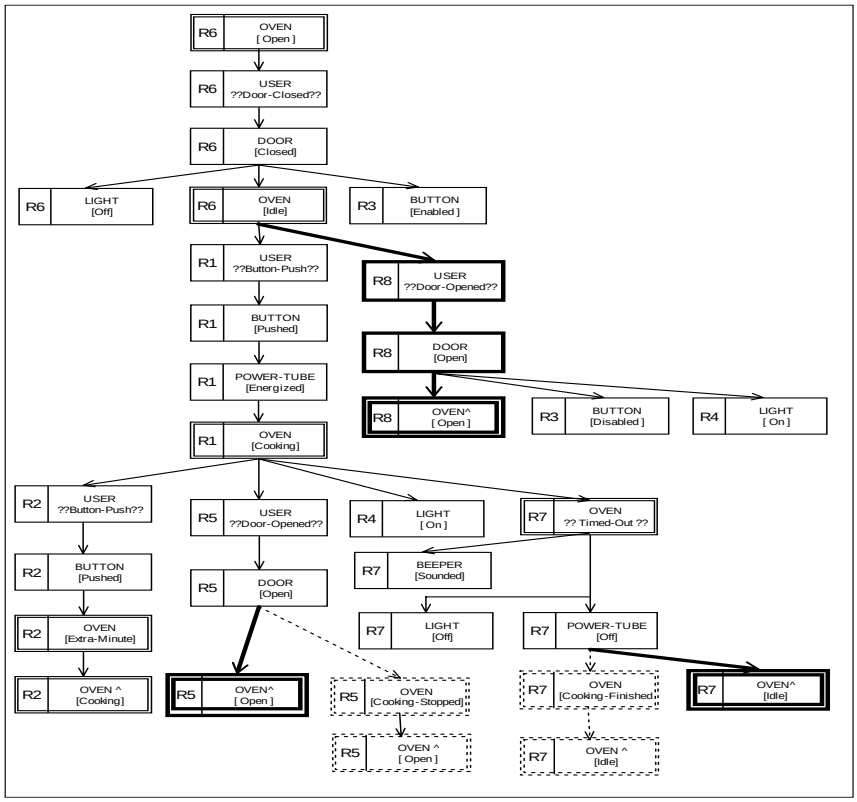


Figure 43. The EDBT merged from the first version of the DBT and the second version of the DBT.

[12] According to the constraints in requirement 3 and requirement 4 in Table 1, when the door is open, the light will be on the button will be disabled. The corresponding RBTs are integrated with R5 in the DBT in Figure 42. In the present example, and subsequent figures we have dropped these two constraints from R5 in the DBT because of space limitations with the diagrams. This does not impact the architecture or the interface because they have been integrated into the missing requirement R8.

The next step is to add the new component TIMER as in previous section. Then we have the R1, R2 and R7 modified as well.

Modified R1: There is a single control button available for the user of the oven. If the oven is idle state and you push the button, the timer will be set to one minute and the oven will cook (that is, energize the power-tube)

Modified R2: If the button is pushed while the oven is cooking it will cause the timer to add one extra minute

Modified R7: If the timer times-out, the light and power-tube are turned off and then a beeper emits a sound to indicate that the cooking is finished.

Based on the modified functional requirements, we construct the third version of the DBT and merge it with the previous EDBT and generate a new EvDBT shown in Figure 44.

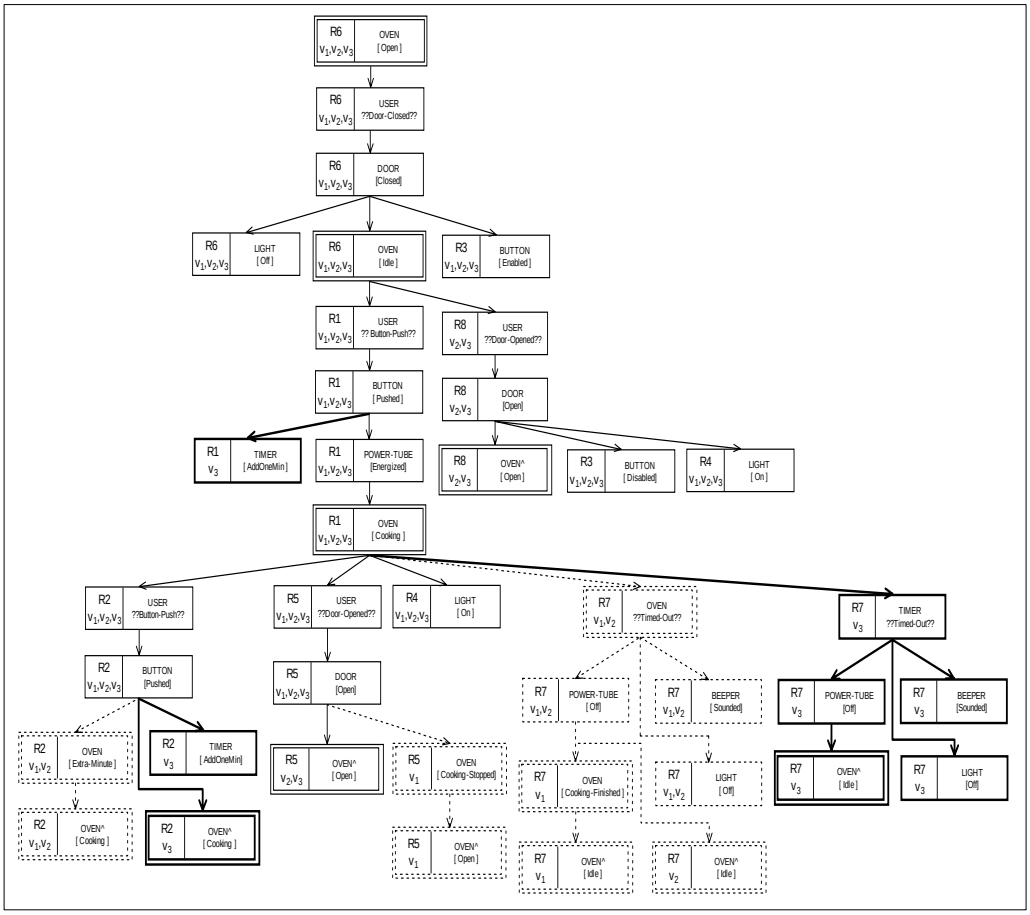


Figure 44. The EvDBT (Evolutionary Design Behavior Tree) of the Microwave Oven System

In Figure 44, the version tags are printed in the left part of the method boxes; the new fragments of behaviors (behaviors that only exist in version 3) are drawn in bold lines, the old fragments of behavior, which are not in the modified system (behaviors not in version 3), are drawn in dotted lines and the unchanged parts (behaviors in both version 2 and version 3) are drawn in the normal style. This diagram shows clearly the change impact of the modified requirements on the behavior tree and the relationships between the behaviors and versions.

From the EvDBT, other diagrams (the EvCIN in Figure 45, the EvCID of OVEN in Figure 46 and the EvCBT of OVEN in Figure 47) are projected. Because of

space limitations, only the evolutionary component diagrams of component OVEN are shown.

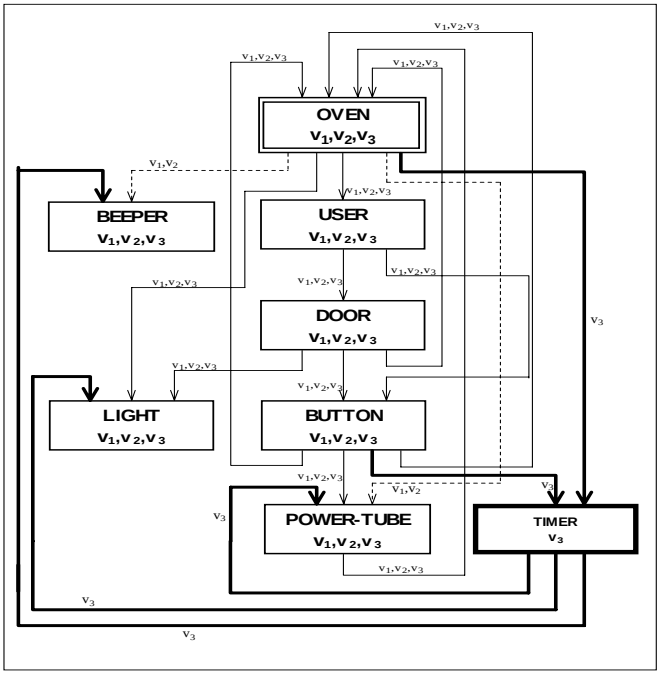


Figure 45. The EvCIN of the Microwave Oven System

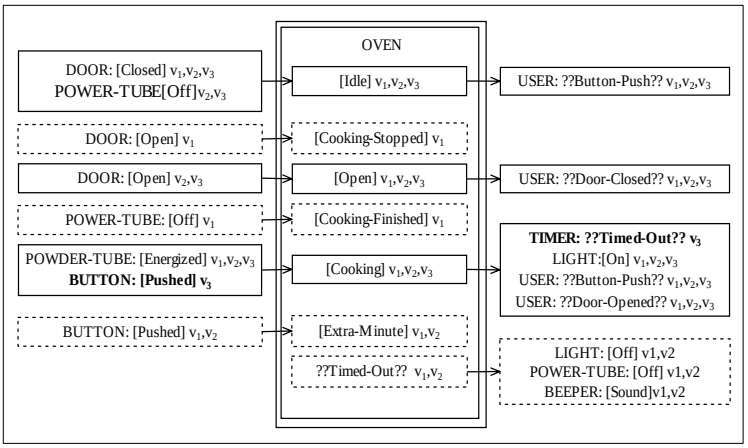


Figure 46. The EvCID of the OVEN Component

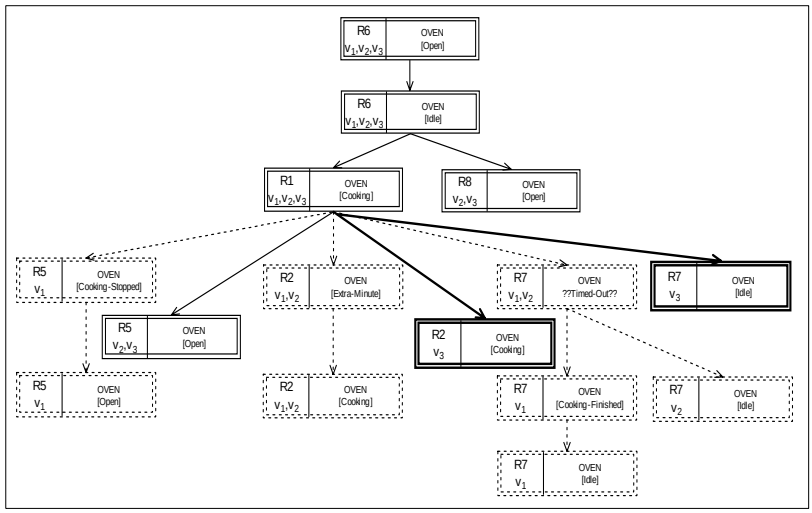


Figure 47. The EvCBD of the OVEN component

In the EvCIN, the change impact and the evolutionary information for the architecture is clearly marked by associating each component and each dependency relationship with a set of version tags. The diagram shows that a new component TIMER is added in version 3 while the other components exist in all the three versions. Similarly, several interaction relationships between the component OVEN and other components are removed and in version 3 and several component interaction relationships with TIMER are introduced in the latest version.

Figure 46 is the EvCID (Evolutionary Component Interface Diagram) of component OVEN. In this diagram, each interface of OVEN, the callers of each interface and what other called interfaces are all marked with a set of version tags. From this diagram, we know that the interface [Cooking-Stopped] is introduced in version 1 but removed in versions 2 and 3 and the interface [Extra-Minute] exists in versions 1 and 2 but is removed in version 3. Generally, an EvCID clearly records the evolutionary history of a component's interfaces.

Figure 47 is the EvCBT (Evolutionary Component Behavior Tree) of component OVEN. Similar to other evolutionary diagrams, each atomic item in this diagram is attached with a set of versions. From these version tags, the evolutionary of a component's internal behavior is recorded and can be easily traced.

This example demonstrates how the proposed model can be used to identify the change impacts on different artifacts in a software system, not only at the architecture level, but also at the component internal structure and interface level as well. This information can be used to direct and trace the changes in the software's implementation, make the system match the changes of the requirements, and eventually reduce the cost for the maintenance of the system.

4.5 Comparison and Conclusion

Other research on software change differs from the method proposed in this thesis. The goal here has been to find a systematic process to map the changes in functional requirements to the changes in the design and the implementation and to record the changes in different types of evolutionary diagrams. From these evolutionary diagrams, with the help of tools, design documents of every version and the comparisons of design documents between any two versions can be easily retrieved. Other approaches, a number of them have been introduced in Chapter 2, include DIF (Garg 1990), SODOS (Horowitz 1986), the traceability model based on B (Bouguet 2005), architectural slices (Zhao 2002) and the difference and union of models (Alanen 2003). All this earlier work has some degree of connection with our traceability model, but our proposed model has some unique merits.

The early software management systems such as DIF and SODOS have provided an environment to store different types of software lifetime objects (SLOs) and also to allow the users to make connections (some of the connections could be automatically generated based on design templates) between those documents. Based on those connections, once a document has been changed, it is possible for people to browse and identify other documents that might have been affected by the change and need to be updated. However, those environments are not based on a well defined design approach such as GSE, which supplies fully bidirectional traceability between several different types of design documents. Therefore those systems will usually not be able to automatically update the impact on documents. Our proposed traceability model can update several different types of design documents automatically or semi-automatically once the new sets of RBTs are created. The software tool GSET, which is introduced in Chapter 7, can demonstrate this feature.

The B notation traceability model proposed by Bouguet (2005) has good tool support. However, compared with the B notation, GSE has a graphical presentation -- the behavior tree, which is easier to understand than textual language such as B. This feature makes the proposed traceability model easier[13] to understand and validate by non-technical stakeholders.

The architectural slices approach proposed by Zhao (2002) is based on the

[13] The traceability model is easier because in GSE, the designers use the same vocabulary as the original requirements; the model provides an overall behavior model and keeps the traceability tags. The cost for the traceability is extra initial processing to insert the traceability tags.

assumption that the software architecture is consistent during the change, so an architectural slice can be used to determine the set of components that may be affected by a potential change. In our model, we focus on the change impact on individual components as well as on the architecture. Therefore, the questions can be answered by our approach include:

1. How the functional requirement changes (behavior changes) can be mapped to the change on components and the architecture.
2. After a proposed change is applied, what is the new component architecture and what is the difference from the old architecture.
3. After a proposed change, which components will be changed and what are the changes on those components' behavior and interfaces.

Of course, similar architectural slicing and chopping techniques can also be adapted into our model to enhance its capability to manage software changes.

Finally, compared with the difference and union models proposed by Alanen (2003), both their approach and our traceability model provide a way to merge multiple models into a new model, but they have the following differences:

- Our method is based on GSE while theirs is for MOF and UML.
- Our method is using tree-graphs, and their solution is based on sequences of operations.
- Their solution of merge may cause conflict during merge, so human adjustment has to be introduced. The reason to cause conflict is that for a sequence of operations, if the order of the operations is changed, the final result of the operations will be changed as well. However, in our approach, because the merge process only involves the operation of set union, it will not lead to merging conflicts caused by applying the change operations in a different order.
- In our approach, we provide a way to present the merged model that shows

information of all the previous models as well as the final model. Furthermore, it also highlights the changed parts. This feature is not common in most other approaches.

- Our model merge operation can be supported by automation tools (Wen 2007a, Wen 2007c).
- From our merged model, we can retrieve other types of models with all the merged information.

In our approach, except for the first step of translating functional requirements into behavior trees, all the other steps are based on well-defined rules and processes. This means they can be implemented by automated or at least semi-automated tools. A further advantage of this automated support is that functional requirements can be integrated into the edit behavior tree one by one. As these changes are made the corresponding design diagrams can be automatically re-generated on the fly to reflect each change as it is made. Therefore, the impact of each individual requirement on the design can be traced. This unique feature gives the method a powerful and systematic means for controlling the impact of change on a design.

The representations we have presented here show considerable promise as the basis for a fundamental theory that could underpin the creation of powerful software design and software maintenance tools. The prototype tool we have developed confirms the feasibility of this approach. It was used to generate the edit diagrams used in this thesis.

There has always been a wide gap between a set of functional requirements and a software design. GSE provides a bridge to link requirements to a corresponding design that will satisfy those requirements. The original GSE method did not answer the question "if one side of the bridge changes, how should the other side change

to make the two parts correspond?" The method introduced in this research directly addresses this question. The proposed method treats a software system as a result of an evolutionary procedure; it not only presents a system but also shows how it comes to be that way. A clear advantage of using a representation that allows us to build a system out of its functional requirements and trace of its change history is that the accompanying change process is relatively easy to formalize and therefore support with automated tools. This representation also helps us answer the question, as to where to make a change, and what impact the change has on the architecture, the component designs and the component interfaces. It also helps with questions about different versions of a design and how to optimize the design of a system.

The proposed model, as presented, is only suitable for software projects that use behavior trees and the GSE design methodology. The concepts employed in this method might however also be adapted for other software design methods, such as the traditional OO design approach based on UML (Fowler 2000). However, the lack of strict dependency relationships among different types of diagrams limits the possibility of automatically updating other design diagrams if one diagram is changed. In contrast, with GSE, the principal diagram is the DBT, which describes the integrated behavior of the targeted system, contains all the information needed to construct the other design diagrams.

Chapter 5 Software Architecture Normalization

Being able to systematically change the original architecture of a component-based system to a desired target architecture without changing either its behavior or the set of functional requirements the system satisfies is a useful capability. It opens up the possibility of making the architecture of any system conform to a particular form or shape of our choosing. The Behavior Tree notation makes it possible to realize this capability. Once this constructive relationship is established between the functional requirements and the architecture it is then possible to transition the architecture of a system from its current form to some target form by appropriately inserting action-inert bridging nodes in the DBT and regenerating the architecture and the component behaviors. For example, we can convert typical network component architectures for a system into normalized tree-like architectures which have significant advantages. We can also use this "architecture change" capability to keep the architecture of a system stable when changes are made in the set of functional requirements for a system provided the requirements changes do not introduce new components into the behavior of the system. The work in this chapter is built on the work of formalizing the impact of requirements change on the design of a system and the results have been published in FACS05 and ENTCS (Wen 2005).

5.1 Introduction

Software architecture is one of the critical issues in software engineering. The term has been given a number of different interpretations (Bass 1998, Stafford 2001, Le 1998, Shaw 1997 and Garlan 1994), which means it needs qualification and clarification when used in a particular problem context. According to Bass (1998), software architecture is defined as "the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them." In this chapter, we will use the concept of component interaction network (CIN) as our chosen architectural construct. A CIN is a graph that shows a software system's components and the dependencies or interactions among them.

The component architecture influences the quality of a component-based software system. If the CIN is too complex it may affect the performance of the system, and make the system difficult to understand and maintain. For example, with a complex CIN, each component may have many dependent connections with other components, which means once the functionality of one component has been changed, because of its high dependency, the change may cause a ripple effect that propagates widely across the system making the impact of the change hard to comprehend and trace.

The structure of a CIN is determined or at least strongly influenced by the functionalities of the system (Dromey 2003). A complex system may inevitably produce a complex component architecture. However, our research shows that the

topological structure of a CIN can be made independent of the functional requirements that the system satisfies. This opens up the possibility of using a relatively simple component architecture to realize a complex system.

To prove this point, we use the genetic software engineering (GSE) design process. As we know, GSE provides a formal process for designing component-based software systems. The underlying strategy of GSE is to build a design out of its requirements. Each individual functional requirement is translated (manually) into a requirement behavior tree (RBT). The resulting set of RBTs are then integrated one at a time to produce a design behavior tree (DBT). The DBT captures all the functional requirements and shows their logical and behavioral relationships. The *component architecture*, the *component behaviors* and *component interfaces* of each of the components in the design are emergent properties of the DBT. The procedures to integrate individual RBTs into the DBT and then derive the CIN from the DBT are precisely defined, so once all the RBTs are fixed, the structure of the CIN is also fixed. Therefore, if we need to change the structure of CIN, we must adjust the RBTs or the DBT.

The question is how we can have different sets of RBTs for the same set of functional requirements. (1) The first method is to adjust the order of nodes in RBTs. For a functional requirement, if the sequence of certain behaviors is not significant and has not been specified, we can draw slightly different RBTs by adjusting the order of some nodes. The difference in the RBTs will not affect the functional requirements but may lead to different CINs. (2) The second method is more systematic. For a given RBT, we can insert *bridge component-states* (or *bridge states* for short), which are similar to hidden events in CSP (Hoare 1985); these states can

be hidden or ignored if we observe the behavior tree from a functional level. Therefore they do not change the functional requirements and hence the behavior of the original RBT. However, these bridge states influence the structure of the CIN. In this chapter, we prove that by inserting suitable bridge states in a DBT, we can manipulate the CIN to whatever pre-selected component architecture we choose. In other words, the component architecture can be independent to the functional requirements.

Generally, a lower coupled system is more portable and easier to maintain. In this thesis, we propose a tree-like hierarchical structure as an optimized component architecture because of the scalability and simplicity of trees. A tree is a connected graph with the least amount of coupling. Also, a hierarchy is a natural form for managing large and complex systems in different disciplines (Ahl 1996). We call a software system with a tree-structured CIN a normalized system and the procedure for transforming a non-normalized system into a normalized system is called *architecture normalization*.

GSE not only provides a systematic approach to construct component-based software design, it also provides a formal method to implement software design changes (Chapter 4). When a software system designed by GSE has been changed due to the changes in the functional requirements, a traceability model has been proposed to show the change impacts on the component architecture as well as on the components behaviors and the component interfaces. Usually, when a software system's functional requirements are changed, these changes affect the component architecture. Repeated changes of a system may eventually ruin a system's architecture. However, based on the result of the present work, it is possible for the

designers to preserve the architecture even though the functional requirements have been changed. Of course, if the changes of the functional requirements cause the system to add new components or remove old components, the original component architecture will have to be changed. Even so the designers can always adjust the new DBT to keep the change impact on the component architecture to a minimum. If the component architecture of a large system can be kept stable during a system's lifetime, it will undoubtedly reduce the maintenance costs of that system.

This chapter is organized as following: Section 5.2 provides the proof of the main theorem - the independence of a CIN from the functional requirements. Section 5.3 give a brief introduction of hierarchy theory and explain the concept of software architecture normalization. In Section 5.4, the microwave oven case study is normalized to illustrate the simplicity of a normalized system. Finally, the last section of this chapter gives a brief conclusion.

5.2 Architecture Transformation Theory

In this section, we will introduce the architecture transformation theory, which shows that, by adding *bridge component-states*, we can modify a DBT to produce a CIN with a pre-selected topological structure. To achieve this target, we have defined some basic concepts in the first sub-section and used a simple example to briefly introduce the main ideas of the architecture transformation theory in the second sub-section, and finally we give the main theorem and the proof in the last sub-section.

5.2.1 Basic Concepts

A CIN includes components and the links among them; a link can be a one-way link or a two-way link. In the original definition of a CIN, if there are two links L_a and L_b that connect a pair of components C_i to C_j in different directions, L_a and L_b are treated as two separated links. Here, in order to simplify the discussion, we merge L_a and L_b into one single link, without explication, any link is supposed to be bi-directional, and a one-way link is only a special case of a two-way link (this difference is unobservable if we abstract a CIN as a bidirectional graph). From this simplification, for any two components in a CIN, there exists at most one link between them.

Definition: A **network** is a graph that includes links and components, each component only appears once in the network and between two different components, there exists at most one link. A **link** is drawn as a line between two components; it can be identified by the two components. Here, we denote a link L as (C_i, C_j), where C_i and C_j are two components in the network (Note that (C_i, C_j) equals (C_j, C_i) in this chapter). The definition of a network is similar to the definition of an undirected graph (Sedgewick 1988, Diestel 1999).

Definition: In a network N, if there exists a link between two components, we say that these two components are **directly connected.** Suppose $C_1, C_2, \cdots, C_m$ are m different components in N, if for all $1 \le i \le (m-1)$, C_i and C_{i+1} are directly connected, we say $C_1, C_2, \cdots, C_m$ form a **path** and the length of this path is *m-1*.

Definition: A network is called **a connected network,** if for all pairs of

components C_i and C_j, which belong to this network, there exists a path starting from C_i and ending at C_j in this network.

Definition: In a network, the **distance** between two components is defined as the length of the shortest path between these two components. Thus, the distance between two directly connected components is one.

Definition: From a DBT T, we can project out a CIN N through a formal process defined in 3.3.1. The CIN is called this DBT's **associated CIN** and the project process is denoted as M. Then we have $N = M(T)$. Here **M** is the project process, T is the DBT and N is the CIN.

Proposition 5.1: A CIN is a connected network.

Proof:

Let T T be a DBT and N be the associated CIN, we have $N = M(T)$. For any two components C_i and C_j belonging to N, because N is T's associated CIN, there exists a state (in the following discussion, we will use the term node to refer to a state) of N_{C_i} in T that is *associated with* component C_i ("associated with" means N_{C_i} is for C_i to realize a state, check a condition or trigger an event, etc.) Let the parent node[14] of N_{C_i} be $N_{C_{i,1}}$ and the parent node for $N_{C_{i,1}}$ be $N_{C_{i,2}}$ …. Then we will have a list of nodes $N_{C_i}, N_{C_{i,1}}, N_{C_{i,2}}, \cdots, N_{C_{i,s}}$ and $N_{C_{i,s}}$ is the root node of T. From this list we can project out a series of component $C_i, C_{i,1}, \cdots, C_{i,s}$.

[14] Because T is a behavior tree, in a behavior tree a state is also referred to as a node and each node except the root node has a parent node.

Then according to the projection rules of GSE, we know $C_i, C_{i,1}, \cdots, C_{i,s}$ is a path or covers a path in N that connects component C_i and the component that is associated with the root node. Similarly, for C_j, there also exists a path linking it to the component of the root node. Merging these two paths together, we have a path linking C_i to C_j, so N is a connected network.

From Proposition 5.1, we know that a CIN must be a connected network; that means all the components in a software system are joined together and any two components are connected directly or through a list of other components. This result is important for proving the main theorem, which shows that the structure of a CIN can be independent of the associated system's functional requirements. We will prove this theorem after considering a simple example, which illustrates the basic ideas.

5.2.2 A Simple Example

To demonstrate the procedure for transforming a behavior tree's associated CIN into another topological structure by inserting bridge component-states, we consider the DBT in Figure 48. The tree T, has 4 components and 4 states. It is easy to find out that the associated CIN N of T is the same structure (Figure 49). We have removed the arrows in N to simplify the discussion.

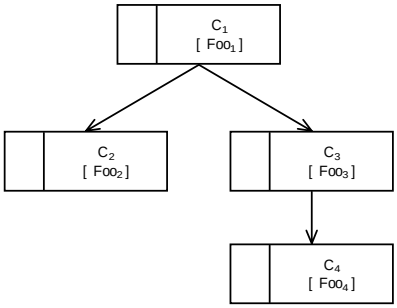


Figure 48. A simple DBT T with 4 components and 4 states

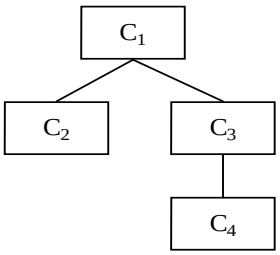


Figure 49. The CIN N of T shown in Figure 48

Now suppose we think the CIN $\tilde{\mathrm{N}}$ shown in Figure 50 is more desirable. The problem is how could we insert bridge component-states in T to make the new tree's associated CIN become $\tilde{\mathrm{N}}$.

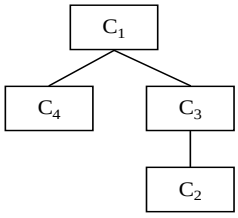


Figure 50. The desired CIN $\tilde{\mathrm{N}}$

The link set of N is $L_N = \{(C_1, C_2), (C_1, C_3), \ (C_3, C_4)\}$, and the link set of $\tilde{\mathrm{N}}$ is $L_{\tilde{N}} = \{(C_1, C_4), \ (C_1, C_3), (C_3, C_2)\}$. Because the links of (C_1, C_4) and (C_3, C_2) exist in $L_{\tilde{N}}$ but not in L_N, we can add two nodes in T to create a new tree $\mathrm{T'}$ shown in

Figure 51

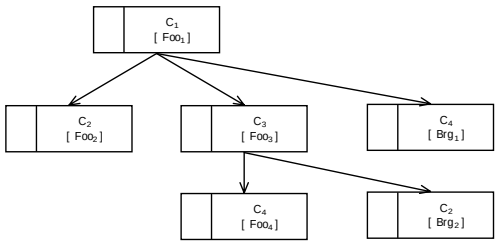


Figure 51. Two bridge component-states are added into the tree T to generate a new tree T'.

Let N' be the associated CIN of T', then it is obvious that the link set for N' is: $L_{N'} = \{(C_1, C_2), (C_1, C_3), \ (C_3, C_4), (C_1, C_4), (C_3, C_2)\}$. Comparing $L_{\tilde{N}}$ with $L_{N'}$ it is found that the links $(C_1, C_2), (C_3, C_4)$ exist in $L_{N'}$ but not in $L_{\tilde{N}}$. To get rid of the extra links, we need to insert bridge component-states between the unwanted direct connections. In Figure 48, there is a direct connection from C_1[Foo$_1$] to C_2[Foo$_2$]. Because C_1 and C_2 are not supposed to be directly connected, we need to insert bridge state(s) between the two nodes. Checking $\tilde{\mathrm{N}}$, we find the path to link C_1 and C_2 is C_1, C_3, C_2, so we should insert a bridge component-state of C_3 between C_1[Foo$_1$] and C_2[Foo$_2$]; by similar analysis, we know that a bridge component-state of C_1 should be inserted between C_3[Foo$_4$] and C_4[Foo$_4$]. The result is we have the new tree shown in Figure 52. Inspecting this tree and we find that if we remove C_4[Brg$_1$] and C_2[Brg$_2$], the associated CIN will not be affected. We therefore remove these two nodes to get the final $\tilde{\mathrm{T}}$ shown in Figure 53.

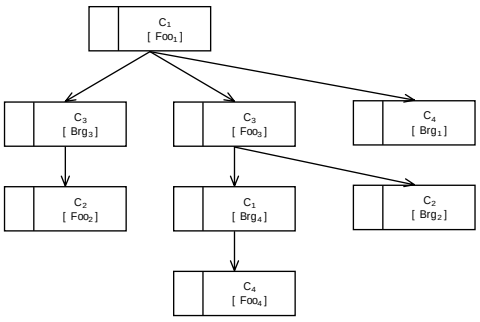


Figure 52. Two more bridge component-states are inserted to get rid of the unwanted direct connections

It is easy to prove that $\tilde{N} = M(\tilde{T})$. If we ignore the bridge component-states in $\tilde{T}$, the behavior of $\tilde{T}$ is exactly the same as the behavior of T. This simple example clearly illustrates how we can transform a component architecture into a new form by inserting bridge component-states into the DBT.

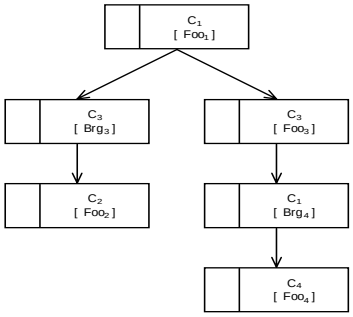


Figure 53. Prune the unnecessary bridge component-states and get the final T

5.2.3 Behavior Invariance Theorem

Definition: A ***bridge component-state***, (also called ***bridge state***), is a special state in a behavior tree. It is visible when the tree is observed from the solution domain, but it becomes invisible when we observe the tree in the problem domain. It is similar to the concept of a hidden event in CSP (Hoare 1985). When we observe a system from a higher level, some low level details become unobservable.

Generally, a design behavior tree (DBT) is a bridge to connect the two domains of a

system: the problem domain and the solution domain. In the problem domain, a DBT should capture all the functional requirements and in the solution domain, many design decisions are properties that directly emerge from a DBT.

Proposition 5.2: When we insert bridge states in a DBT, the bridge states will not change the functional requirements captured by the behavior tree.

Proof:

Bridge states are only visible in the solution domain. When we check the functional requirements captured by a DBT, we are looking at it from the problem domain, so the bridge states are invisible and the DBT has not been changed with regard to the functional requirements.

Theorem 5.1: Let T be a DBT and N be its associated CIN, where $\mathrm{N} = \mathrm{M}(\mathrm{T})$. Suppose there are a total of s components $C_1, C_2, \cdots, C_s$ in N and $\tilde{\mathrm{N}}$ is an arbitrary connected network that includes and only includes those s components. Then, by adding extra nodes to T, we can produce a new DBT $\tilde{\mathrm{T}}$ with $\tilde{\mathrm{N}}$ as the associated CIN of $\tilde{\mathrm{T}}$, where $\tilde{\mathrm{N}} = \mathrm{M}(\tilde{\mathrm{T}})$.

Proof:

A network can be represented by a set of components and a set of links, and each link can be represented by a component pair. Therefore, for two networks, if they have the same component set and the same link set, they are identical. Now we compare N and $\tilde{\mathrm{N}}$, because they have the same component set, if they are different, they must have different link sets. In this situation, there are only two possible scenarios, the first is that there is a link (C_i, C_j) that belongs to $\tilde{\mathrm{N}}$ but

does not belong to N, or there is a link (C_l, C_k) which belongs to N but not to $\tilde{\mathrm{N}}$. Let us discuss the two scenarios separately.

If the link (C_i, C_j) belongs to $\tilde{\mathrm{N}}$ but does not belong to N, we can simply add node of C_j under node of C_i in tree T. The associated CIN will then include the link (C_i, C_j).

If the link (C_l, C_k) belongs to N but not to $\tilde{\mathrm{N}}$, then in tree T, there must be a node of C_l that is directly connected to a node of C_k (note this direct connection between C_l and C_k may have multiple occurrences, but based on our method, multiple occurrences can be handled by repeating the insertion operations multiple times). As for $\tilde{\mathrm{N}}$, because it is a connected network and C_l, C_k are not directly connected, there must exist a path between C_l and C_k. Excluding C_l and C_k, supposing the path includes components $C_{n_1}, C_{n_2}, \cdots, C_{n_t}$, then at the each occurrence of component C_l and component C_k directly connected in T, we add a series of states of $C_{n_1}, C_{n_2}, \cdots, C_{n_t}$. Then the modified behavior tree's associated CIN will not have the direct link of (C_l, C_k). Because the inserted nodes are ordered according to an existing path in $\tilde{\mathrm{T}}$, the insertion of the new states will not introduce extra links that are not in $\tilde{\mathrm{T}}$, and the effect of the insertion operation will remove the link of (C_l, C_k) from the associated CIN.

Theorem 5.2: Let T be a DBT and N be its associated CIN, where $\mathrm{N} = \mathrm{M}(\mathrm{T})$. Suppose there are a total of s components $C_1, C_2, \cdots, C_s$ in N and $\tilde{\mathrm{N}}$ is an arbitrary connected network that includes and only includes those s components.

Then, we can create a new DBT $\tilde{T}$ that captures the same set of functional requirements as T, and has $\tilde{N} = M(\tilde{T})$.

Proof:

According to Theorem 5.1, we can add extra nodes into T to generate a new DBT $\tilde{T}$ with $\tilde{N}$ as the new tree's CIN. If we make sure all the inserted new nodes are bridge component-states, according to Proposition 5.2, the inserted nodes will not change the functional requirements of the original behavior tree T. Then Theorem 5.2 is proved.

Theorem 5.2 is interesting because it states that the component architecture is somehow independent to the functional requirements. It means we can pre-determine the desired architecture of a software system. To extend the idea further, a more significant conjecture is that there may exist universally optimized architectures that can be implemented as standard architectures for different software systems.

What kind of topological structure is optimized? This question may have different answers under different criteria. In the following section, we propose a tree-structured hierarchical architecture as an optimized form due to some of the unique features of trees.

5.3. Software Architecture Normalization

Nearly everything a man confronts in his daily life could be classified as a problem of a complex system, from balancing his physical body, thinking, to operating a car.

All of these common tasks involve systems with thousands or even millions of components. If a person needs to be aware and to control all the details of all the components to perform those tasks, none of them can be possibly performed on a daily basis. The reason that we can handle complex systems with much less effort is hierarchy. In a hierarchy, details of low level components are hidden and controlled by higher level components. Through the limited interfaces provided by the top-level component, one can easily manage a complex system with thousands or even millions of components.

As software systems become larger and more complex, it is natural to implement the concept of hierarchy to design those systems. Actually, a software system can be treated as different hierarchies from different views. For example, the most straight forward way to examine a software system as a hierarchy is going through the inclusive approach: A software system developed in Java may include several software packages, a package may include many classes, a class includes methods and attributes, and finally a method includes multiple Java statements. However, in this chapter, we mainly focus on the hierarchy for integration of relationships among components.

A large software system can have hundreds or even thousands of software components. Those components are integrated with each and form a complex network called CIN (component integration network). Usually the network is a scale-free network (see the next chapters) and not a tree form. It can be very hard to understand and maintain the system by checking this network due to its complexity. Software architecture normalization allows us to transform this network into a tree-formed hierarchical structure. After normalization, the topological form of

CIN is greatly simplified. It will be much easier to isolate and update individual components (since the integration relationships with other components are clear and simple) as well as to understand the system as a whole. In general, normalization makes the system much easier to understand and maintain.

In this section, hierarchy theory and properties of trees are briefly introduced, and then the concept of software architecture normalization is presented; finally, we present a case study and some discussions.

5.3.1 Hierarchy Theory

Different Types of Hierarchies

Hierarchy is a natural phenomenon on that exists in diverse situations around the world. From the physical objects of the universe, management systems in companies to genealogy trees of families, we can always find different types of hierarchies.

Two of the most prominent types of hierarchy are composition hierarchy and controlling hierarchy (Ahl 1996). Composition hierarchy can be easily observed in any physical systems of different scales. E.g. a table may be composed of a top and 4 legs; the solar system is composed of the sun, 9 large planets and many other space objects; a molecule is composed of a number of atoms. Compared to a composition hierarchy, a controlling hierarchy is more abstract. A good example of a controlling hierarchy is the management structure of a typical company. The

leading role is the general manager; under him there are directors of individual departments; under each director, there are a number of staff.

Besides the two major types of hierarchies, we may find many other types of hierarchies such as sub-class hierarchy (Dromey 2003b), e.g. the class of vehicle includes sub-class car and truck; the sub-class car may include sports_car and sedan (Figure 54), and inheritance hierarchy which is usually drawn like a family's genealogy tree.

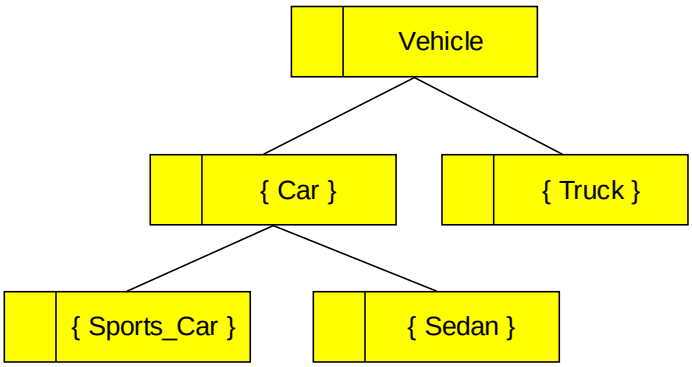


Figure 54. Sub-class hierarchy

The similarity among different types of hierarchies is that they all can be visualized as trees. The difference between them is the specified relationship between a parent node and its child nodes. In this chapter, since we are studying the component integration network (or component dependency network), the focused relationship is the dependent relationship between software components.

What, How and Why – Information Hiding

One of the most important properties of object-orient programming is information hiding, which is to separate the interfaces and the implementations. Usually, the interfaces are open but the implementations are hidden from other objects.

Tree-structured component architecture not only supports information hiding of the implementation, but also raises a new type of information hiding – *purpose hiding*.

When we design a component, we need to answer three questions: **What** is the functionality of this component, **how** does this component realize this functionality and **why** do we need a component with this functionality? The three questions represent three aspects of a component: the interface, the implementation and the purpose. The relationship between the interface and the implementation has been discussed in many object-oriented programming books already, but the purpose of a component is rarely stressed as a kind of information hiding.

A tree-structured component architecture is a hierarchy. In this system, the purpose of each component can be abstractly summarized as to help its parent component to realize the parent's functionality. So the functionality of its parent is its purpose. In this scene, a component itself does not have a purpose or, in other words, it has no reason to exist by itself. Therefore, the purpose of a component, in the hierarchy model, is hidden by its parent component; the component only "knows" what its functionality is and in order to realize this functionality, it may need several child components. This component needs to specify the functionality of its each child component but how those child components realize those functionalities is also hidden from this component.

To explain the concepts of implementation hiding and purpose hiding in a simple way, let us consider an example of a low rank officer X in an army. As an officer, X

receives orders from his super officer. When he receives an order, he may decompose the order into a few tasks and assign those tasks to individual soldiers under him. In this hierarchy, we have three levels, the upper level officer, the low level officer and the soldiers. For X, when he assigns tasks to the soldiers, what is important to him is that these tasks need to be finished by the soldiers but how these tasks are finished by the soldiers is of no interest to him and can be hidden to him. We call this implementation hiding. Apart from the implementation hiding, the upper level officer also hides something from X that is the purpose of an order. When X receives an order, he is not in the position to ask why he is given such an order or the purpose of such order. His duty is only to obey the orders. In this situation, we call it the purpose hiding. People may argue that if X knows not only the order but also the purpose of the order, he may be able to fulfill the purpose in a better way rather than simply following the orders. It could be true in many isolated case studies. However, without the purpose hiding, the duty of X is increased and the authority of the upper level officer will be decreased. Generally, the purpose hiding in an army simplifies the management and makes the performance of the army more predictive.

Similar to the army officer example, the concept of two types of information hiding, the parent component hides the purpose and the child components hide the implementation, can be used in software engineering and it simplifies the design and development of each individual component since it only needs to focus on one question: what is its functionality.

5.3.2 Trees and Normalized DBTs

There are a number of equivalent definitions of trees and a number of mathematical properties that imply this equivalence (Knuth 1997, Sedgewick 1988, Diestel 1999 and Kingston 1998). Since most of the properties are obvious and have been discussed by many books on data structures, we will not repeat the proof for some of the obvious propositions.

Proposition 5.3: A connected graph is a tree when and only when for each pair of nodes in the graph; there is only one unique path between them. (A path is a sequence of connected links and no node can be included twice in a path). This property is sometimes used as the definition of a tree (Kingston 1998).

Proposition 5.4: A connected graph is a tree when and only when there is no circular path (If there is a circular path in a graph, then there exist two different paths between any two nodes on the circular path).

Proposition 5.5: A connected graph with n nodes has at least $(n-1)$ links. It is a tree when and only when there are $(n-1)$ links. In other words, a tree is a connected graph with the least possible number of links (Sedgewick 1988 and Kingston 1998).

Definition: A DBT T is called a **normalized DBT** if the associated CIN N ($N = M(T)$) is a tree. A software system with a normalized DBT is called **normalized software system**.

Theorem 5.3: Any DBT can be normalized (transformed into a normalized DBT) by adding bridge component- states. (Direct result from Theorem 5.2)

Proposition 5.6: For a CIN N with *n* components, the number of the links must be greater than or equal to $(n-1)$. The number of links equals to $(n-1)$ if and only if the CIN is a tree.

If we use the number of links among components as a measure of the complexity of the architecture of software systems, proposition 5.6 indicates that a normalized software system has the simplest architecture.

Proposition 5.7: Let T be a DBT and N be its associated CIN. T is normalized when and only when for all pairs of components C_i and C_j in N, there exists only one path between the two components in N provided no node in the DBT is included twice in a path.

Proof:

According to the definition of a normalized DBT, its associated CIN is a tree. Each component is associated with one node in the tree. Then according to proposition 5.3, for each pair of nodes, there is only one path between them[15].

Proposition 5.7 indicates a very important feature of a normalized software system. For large software systems, we frequently face the problem of passing references, messages or attributes between different components. We cannot make each pair of

[15] For a pair of components, we may have multiple types of information exchanged between them, for example, data flows or controls. However, in this paper, we assume that we can apply one type of abstract connection that can pass all the different types of information. Therefore, we can have at most one connection between two components in all the possible situations.

components directly connected to each other because that will make the whole system too complex. In this situation, we can use some components as bridges to pass messages or references. If there are multiple paths between two components, we may accidentally use different path to pass different types of messages and eventually make the system too complex to maintain. If there is only one path between any pair of components, this problem will be solved easily.

Proposition 5.8: If there is no mutual component in two tree-structured CINs, when the two CINs are connected by a link, the new CIN is also tree-structured.

Proof:

Suppose the number of components in the first CIN and the second CIN are *n* and *m*; then the number of links in the first CIN and the second CIN are (*n-1*) and (*m-1*). Because there is no mutual component in the two CINs, the merged CIN will have (*m*+*n*) components and $(n-1)+(m-1)+1=m+n-1$ links. According to proposition 5.6, the merged CIN is also tree-structured.

Proposition 5.9: Consider two tree-structured CINs N_1, N_2. If there is only one mutual component C in both CINs, the two CINs can be merged through the mutual component C; then the merged CIN is also tree-structured.

Proof:

Suppose the number of components in the first CIN and the second CIN are n and m; then the number of links in the first CIN and the second CIN are (*n-1*) and (*m-1*). Because there is only one mutual component C in the two trees, the merged CIN will have $m+n-1$ components and $(n-1)+(m-1)=m+n-2$ links. According

to proposition 5.6, the merged CIN is also tree-structured.

Theorem 5.4: If a normalized DBT T is broken into two DBTs T_1 and T_2 by cutting off a link; then T_1 and T_2 are also normalized DBTs.

Proof:

If T_1 is not normalized, let N_1 be the associated CIN of T_1. N_1 is not tree-structured. According to proposition 5.7, there exists at least a pair of components C_i, C_j in N_1 and there are two separated paths between them. When T_1 and T_2 are merged into the original T, because no link in the T_1 is lost in T, the associated CIN of T has all the links in T_1's associated CIN. So the two separate paths linking C_1 and C_2 are also in T's associated CIN, but this is contrary to the condition that T is normalized. Therefore, we know T_1 is normalized, and similarly T_2 must be normalized.

Proposition 5.8, proposition 5.9 and theorem 5.4 are very interesting. They specify a unique feature in trees. That is, if a tree is broken into two parts, each part is still a tree; if two trees are integrated into one graph, the graph is also a tree if the integration is based on specified rules. This feature simplifies the procedure of integrating and decomposing a normalized system because the property of normalization will hold if we decompose a normalized system and also when we integrate two normalized systems, if there is only one join point, the integrated system will also be normalized. This feature is very important for building large systems. No matter what is the size of the final system, it can be broken down into a few smaller sub-systems and each sub-system can also be broken down into sub sub-systems etc. Then we can manage each small part and make it normalized, and then integrate them hierarchically to finish the final system. According to theorem

5.4, the final system is also normalized and the architecture is still in a well organized tree-structure. The cost of using these normalization procedures is that control and data will sometimes have to flow through longer paths than perhaps is necessary – in return we can achieve architectural control and stability.

5.3.3 Comparison to Common Architectural Styles

Architectural styles provide a standardized vocabulary to help stakeholders communicate about the high-level structure of a software system (Stafford 2001). Some common architecture styles include Pipe and Filter, Shared Repository, Layered Abstract Machine, Buss and Client-Server (Perry 1992, Shaw 1997). Below is a brief description of the 5 common software architectural styles.

- **Pipe and Filter**: It is like a stream; each component has one input and one output, and the output of the previous component will be the input for following component
- **Shared Repository**: There is a central data repository that can be directly accessed by a number of different components.
- **Layered Abstract Machine**: The system is stratified and each layer includes a number of components. The data processed in one layer is only available to the components in the above layers.
- **Buss**: There is a shared communication medium that is directly connected to a few components. Data is broadcast over the medium and is available to each component. A single component can select to process the data or ignore it.
- **Client-Server**: There is a server component and a number of client components. A request is sent from a client component to the server

component, and then the server component processes the request and sends back a respond to the client component.

The topological structures of the 5 architectural styles are illustrated in Figure 55, exhibit one interesting property: If we abstract components as nodes and the interaction relationships between components as links so each type of style is presented as a graph, then we will find that most of the graphs are special types of trees. For example, in the Pipe and Filter structural style, it is a tree where each node has one or zero child nodes; in the Shared Repository style and the Client-Server style, the graphs are trees of one parent node with a number of child nodes. The Buss style and Layered Abstract Machine styles are exceptions. However, in the Buss style, if we treat the central line as a special node then it becomes a tree; in the Layered Abstract Machine, if we group components in the same layer as single large component, then the associated graph becomes a tree as well.

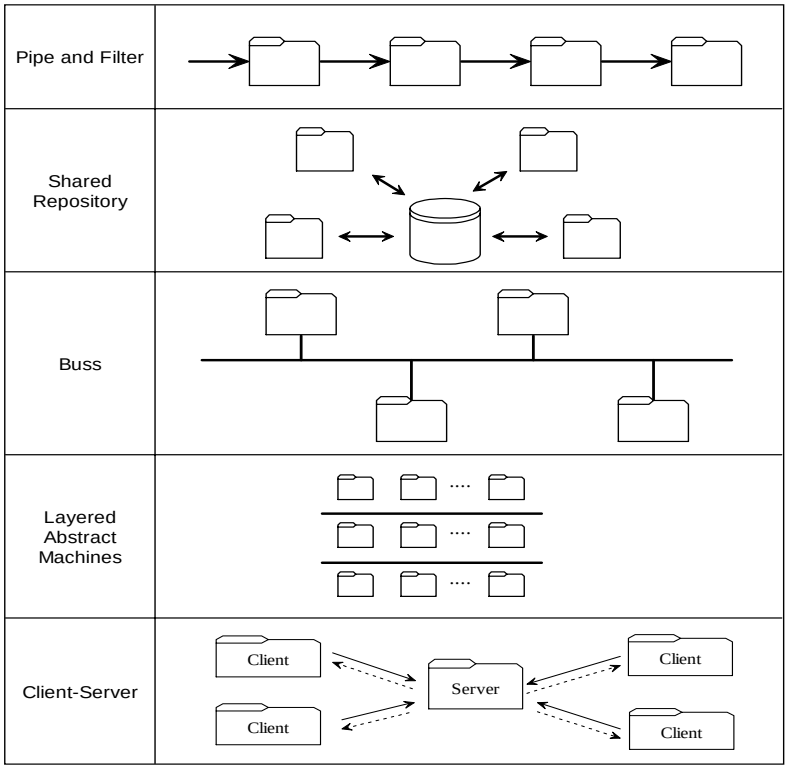


Figure 55. The topological structures of common architectural styles used in software systems

From the analysis above, we find the tree-formed topological structure, due to its simplicity, is very common among general architectural styles. However, for the five listed common architectural styles, because they only assume special forms of trees, they have generally lost one of the most important features of trees: the scalability. For large software systems, a common architectural style listed above is usually implemented in a limited range at a certain level. It is hard to apply a single style in the whole system and in different levels. However, for the tree-formed architectural style, due to its scalability, it is possible to keep the style in the whole system at all levels.

Another difference between a normalized software system with a tree-formed software architecture and the common structural styles is that the tree-formed structure is the key feature in a normalized system but for the common architectural styles, the tree-formed feature usually exists only in the simplified illustrated version. When the system becomes larger, the tree-formed feature can be easily broken. For example, in the client-server architecture style, a server component can be treated as a parent node and the client components can be treated as child nodes. Of course, a client component can also work as a server component for other components. If we restrict things so that each component can have at most one server component, then these components and the associated client-server relationships will form a tree. However, it is very common that one client has more than one server for different services, so the simplicity of a tree is destroyed.

A layered-system also has some similarities to a tree-structured CIN. In a

layered-system, each layer provides service to the layer above it and serves as client to the layer below it (Shaw 1997). One of the main differences between layered systems and tree-structured systems is: In a layered-system, components within the same layer can communicate with each other directly, and any two components from two neighboring layers can communicate. But in tree-structured systems, a component is only directly connected to its parent component and child components. Another difference is in a tree-formed architecture, a component on a lower level can have one parent node in the upper level, but in a layered-system, a component in a lower level can be directly accessed by many components in the directly connected upper level.

Generally, for most common architectural styles, when they are illustrated in a highly abstract and simplified form, they are usually tree-structured. However, due to the limitations of those styles, a single style normally is not sufficient to cover a complex software system in different levels, so that they usually lack scalability, the unique feature of trees. The result is, in a large software system, when we examine the component or class level, the architecture will become a complex network.

5.3.4 Case Study

In Chapter 3, we have used a Microwave Oven case study to explain the fundamental concepts of GSE. Here we will normalize it to demonstrate how the component architecture can be simplified through the normalization. Figure 56 shows a normalized DBT of the Microwave Oven case study. The normalized process is a mixture of inserting bridge states and adjusting the order of some states. The bridge component-states are colored with grey. The associated CIN of the

DBT is shown in Figure 57.

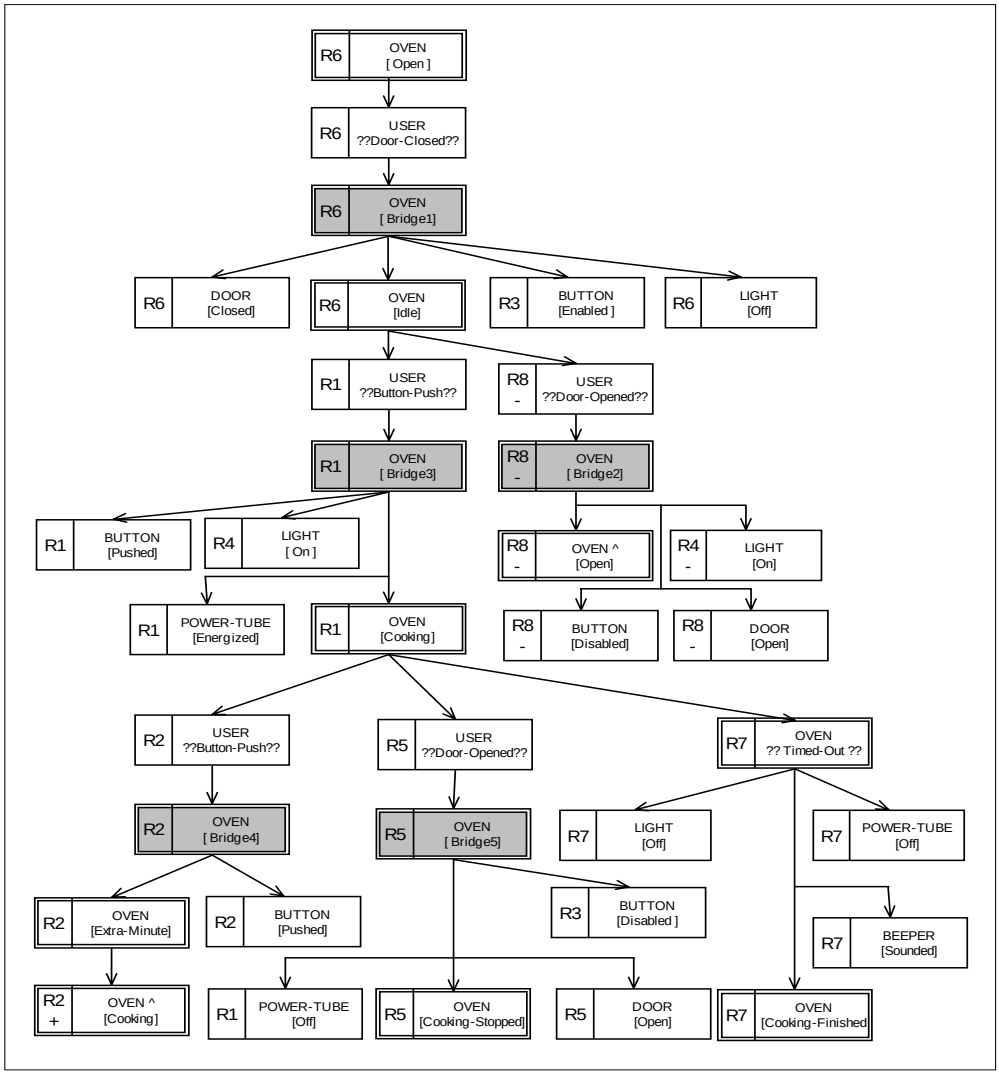


Figure 56. A normalized DBT for the Microwave Oven System

Comparing the normalized DBT with the original DBT in Figure 23, we have found that differences between the two behavior trees are trivial and both DBTs capture all the functional requirements in Table 1. However the differences between the two CINs are significant. The CIN shown in Figure 57 is much simpler than the original CIN in Figure 24. Even though the Microwave Oven case study is a small system with only 7 components, the architecture normalization has dramatically simplified

the component architecture. If the same process is applied in large systems, we expect that the impact of simplification on the component architecture will be more significant.

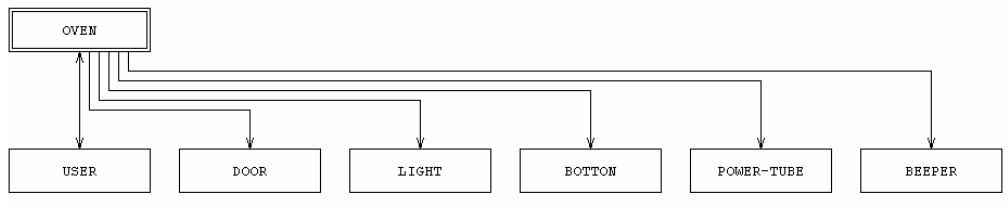


Figure 57. The tree-structured CIN associated with the DBT in Figure 56.

Figure 58 shows another normalized DBT of the Microwave Oven case study and Figure 59 is the corresponding CIN. Comparing the CIN in Figure 59 with that in Figure 57, the new tree-structured CIN is more general because it has three levels.

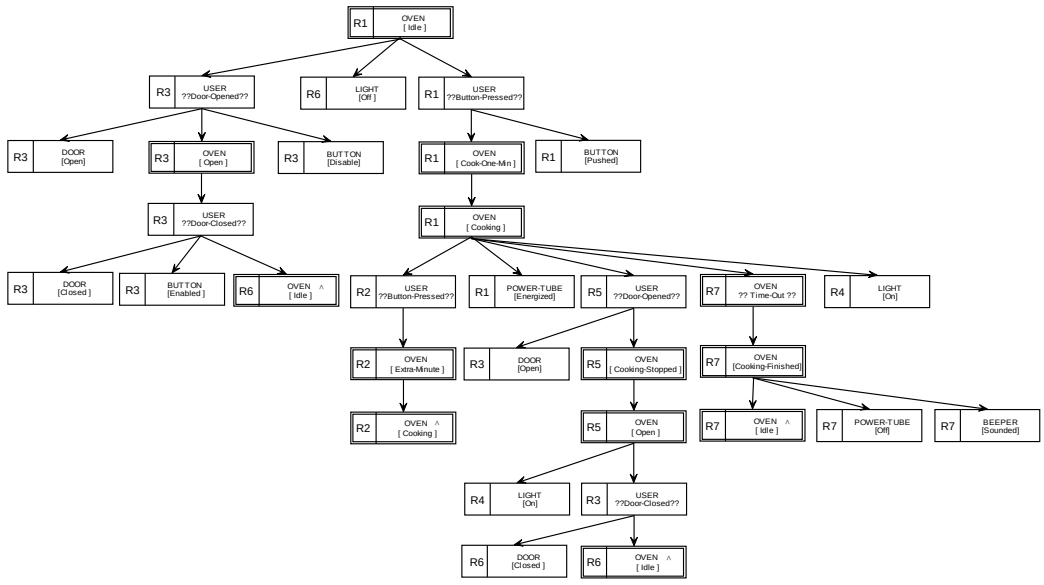


Figure 58. Another normalized DBT of the Microwave Oven case study.

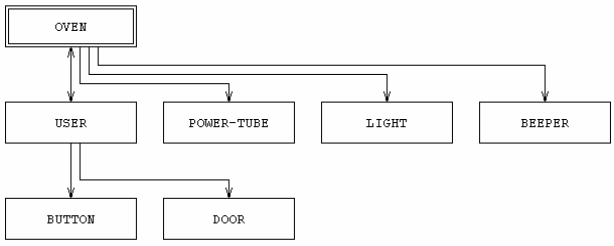


Figure 59. The CIN projected out from the DBT in Figure 58.

5.4 Conclusion

This Chapter has addressed two things: the relationship between the functional requirements and the component architecture of a system, and the control of change of the architecture of a system. A consequence of this work has been results that show the advantages of using tree-like architectures as simple optimized forms.

The component architecture of a system must support the implementation of all the integrated behaviors of a system. The latter are in turn implied by the set of functional requirements for the system. Current software engineering practice suggests that, for a given problem, there exist many different approaches to designing a solution to the problem (Glass 2004) each of which may lead to a system with a different component architecture. What we have sought to do is establish the relationship between a set of functional requirements and the component architecture of a system and then shown how systematic change of the architecture can be achieved without affecting the set of functional requirements that the system satisfies. The formal result we have obtained shows how we can decouple the component architecture of a system from its functional requirements once we have initially established the relationship between the requirements and the architecture.

Once we have the means to systematically change the component architecture of a system we can equally effectively use this power to resist the consequences of changes on the architecture of a system. It is a well known observation of software engineering practice that repeated change to the functional requirements a software system tends to gradually degrade the original component architecture and increase the cost of the maintenance. The results in this paper prove that we can usually

keep the component architecture constant when a system is changed. This has significant implications for reducing the cost of software maintenance.

According to Medvidovic (2002), "current software engineering practice is the continued preponderance of ad-hoc development approaches … rather than well-understood scientific principles". As a result, many existing architecture styles like those introduced in the section 4.2, together with styles like the C2 style (Medvidovic 2002) and MDA (ORMSC 2001) have been developed based on certain assumptions about the implementation environment or current functional requirements. For example, MDA, particularly focuses on development environments such as CORBA, JAVA, and .NET. In this thesis, we have sought to keep the concept of component and the relationship between two components at the highest abstract level, in order to obtain results that are completely independent of any implementation considerations. This has led to a proposal for the use of tree-formed architectures as simple optimized forms.

People can argue that software architecture may not be determined by the functional requirements but may be determined by non-functional requirements such as the performance and security requirements. We suggest that even though non-functional requirements can improve some limitations on the software architecture, (for example a certain component must be directly or not directly connected to another specific component,) the majority of the software architecture will not be fully determined by the non-functional requirements. Therefore, after some of the critical decisions of the software architecture have been determined by the non-functional requirements, the rest part of the architecture can also be designed based on the normalization procedures proposed in this chapter.

Chapter 6 Software Systems and Scale-Free Networks

In the previous two chapters, we have introduced a new traceability model and software architecture normalization. Even though these approaches are based on GSE and novel, they are still following the traditional way to investigate software changes. The traditional way focuses on individual changes and the change process is described as a minicycle (Rajlich 1999).

In this chapter, we will use a broader view to examine software changes. In this view, we will not be concerned about the reasons for and change impacts of individual changes; instead, we focus on the topological properties of a component architecture after a sequence of software changes or software evolutions. To achieve this, the major tool we have selected is network theory. Some results in this chapter are published recently (Wen 2007b).

6.1 Introduction

Anything called a system can be treated as a network. In a system, individual components cooperate with each other to allow the system to realize its higher level

achieve functionalities. From a network's point of view, a component is abstracted as a node while the cooperation between two components is abstracted as a link between the two nodes. Using this idea of abstraction, a bicycle is a network of mechanical parts linked by physical connections, a brain is a network of nerve cells connected by axons and society is also a network of people linked by different kinds of relationships. Other examples include economic systems, ecosystems and power supply systems. They are all complex networks.

Despite the pervasiveness of networks, one important similarity among many different types of complex networks was discovered only recently (Barabási 2003). This discovery reveals that in spite of the huge number of nodes, which can be millions (the population in a country) or even billions (web pages of the www), there will be only a relatively very small number of eminent nodes that seem to rule the whole network by attracting a significant number of links. The property that there is no upper bound for the number of links on a node is called the "scale-free" property (Barabási 2003). A network with this property is called a scale-free network.

A scale-free network's most prominent property is that the tail of the distribution of link numbers follows a "power law" pattern. Let P_k be the probability of a node with k links; the power law indicates $P_k \sim k^{-\gamma}$ when k is large (where γ is a constant, usually less than 3 and greater than 2).

Recent research reveals that many complex networks are scale-free. Examples include: the World Wide Web, the social network, the movie star network, the power

supply network, the scientist co-author network, the chemical network of a cell, the web of human sexual contact etc (Barabási, A., 2002).

In this research, we have explored the topological structures of the component dependency networks (CDN)[16] of seven Java packages. In a Java package, a component is defined as a public class or an interface. In our study, we have discovered that all the CDNs of the tested Java packages are scale-free networks. This result indicates that even though the component architectures of different software systems are different in detail, they are all controlled by the same laws. We presume that component dependency networks of most large software systems will be scale-free unless a system has been specially manipulated so the CDN can be in some other forms.

Another discovery is the relationship between scale-free networks and optimized sorting algorithms. Most sorting algorithms require comparisons of the key values of target records. If we consider a record as a node and the comparison between two records as a link, the process to sort a sequence of records generates a network, which is called a sorting comparison network (SCN). Through the study of the topological structures of 5 different sorting algorithms, we have discovered that for a sorting algorithm, if the number of comparisons is close to the theoretical lower bound ($\lceil \log(n!) \rceil$), the SCN tends to be a scale-free network. The result suggests that the scale-free property is an indicator of the efficiency[17] of a sorting algorithm.

---

[16] The concept of component dependency network (CDN) is same as component integration network (CIN) that shows the dependency relationships between components in a software system.

[17] In this thesis, when we investigate a sorting algorithm, we only consider the operation of comparison, other operations such as inserting and swap are ignored.

Based on the second discovery, we conjecture that the scale-free property can be used as a measure of the optimization of the topological structure of a network. If this conjecture is true, based on the first discovery, we conjecture that the CDN of those software systems as well as many other large complex networks are optimized in certain aspects.

This chapter is structured as follows: Traditional graph theory and scale-free networks are reviewed in Section 6.2. Section 6.3 introduces the methodology we used to explore the properties of Java packages' dependency networks and the testing results are also given in that section. Section 6.4 presents our discovery of the relationship between scale-free networks and optimized sorting algorithms. Finally, in the last section, some discussions are presented.

6.2 Scale-Free Networks

6.2.1 Graphs and Networks

A graph is a pair of sets $G = \{P, E\}$, where P is a set of points (vertices) $P = \{p_1, p_2, ..., p_n\}$ and E is a set of edges (lines) $E = \{e_1, e_2, ..., e_k\}$. Each edge in E connects two points in P.

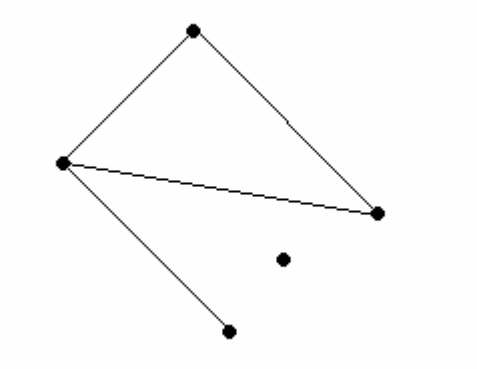


Figure 60 A simple graph with 5 points and 4 edges.

Graph theory had its origins in eighteenth century in the work of Leonhard Euler. In the early stages, the graph theory mainly focused on small graphs with a high degree of regularity (Albert, R. and Barabási, A., 2002). An example is the problem of the Königsberg Bridges (Figure 61).

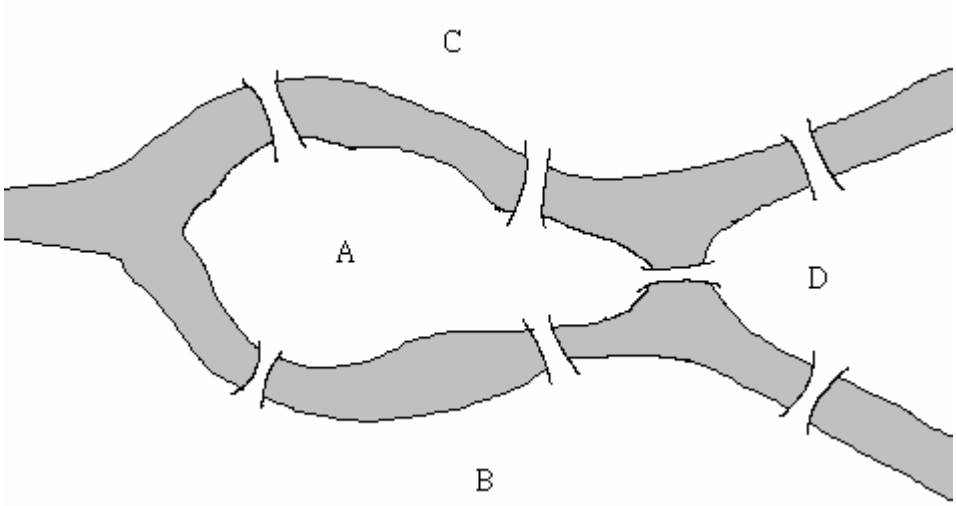


Figure 61 **Königsberg Bridges**. In Königsberg, there were seven bridges connected between one island A and three land areas B, C and D. The problem is to find a path that goes through all the 7 bridges once and only once.

In 1736, Euler solved this problem by introducing graph theory, in which the 4 land areas are represented by four points (A to D) and each bridge is represented by an edge (Figure 62). Using his new theory, Euler proved that on this graph, a path crossing each edge only once does not exist.

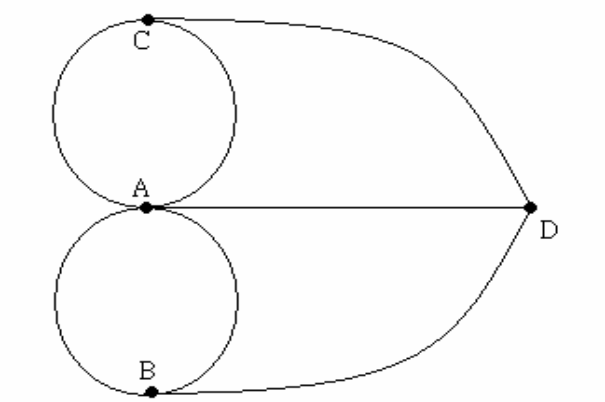


Figure 62 The graph derived from the problem of Königsberg Bridges. Now the problem becomes starting from one of the 4 points, to find a path that goes through all the 7 edges only once. The proof of Euler is simple: If there is a path going through all the 7 edges only once, it must cross all the 4 points. Only the starting and the ending points can have odd numbers of edges. For the middle points, if there is an edge to lead in, there must be another edge to lead out, so it must have an even number of edges. However, all the 4 points in the graph have odd number of edges, so the path crossing all the seven edges only once does not exist.

Euler's solution is elegant. However, it is not the problem or the proof that makes Euler's contribution important but rather the intermediate step that he took to solve the problem. The step to transfer the layout of Königsberg Bridges into a graph, which is a collection of points and edges, symbolizes the birth to the graph theory, a new branch of mathematics.

A network is a system that can be visualized as a graph. In a network, a point is usually called a node while an edge is usually called a link. A link can be directional or bi-directional. Different from traditional graphs, modern network theory mainly deals with random (irregular) graphs with huge number of nodes and links.

One important point in the problem of the Königsberg Bridges is that when the

layout of bridges is transferred into a graph, some internal properties are still kept. Similarly, when a complex system is described as a network and visualized as a graph, some properties of the system are recorded in the topological structure of the network. Studying a network's structure may reveal deep properties of the system.

6.2.2 Random Network Model

In 1960s, Erdős and Rényi (1960) introduced the random-graph theory that has dominated the graph theory for more than 40 years (Albert, R. and Barabási, A., 2002). A random graph can be defined as a graph with N labeled nodes and the probability to have an edge between any two nodes is a constant p. Based on this model, defining the number of edges as a variable K, then the expectation of number of edges is:

$$E(K) = p \times \frac{N \times (N-1)}{2}. \quad (1)$$

A typical question addressed by random-graph theory is the relationship between the probability p and graph properties when the number of nodes $N \to \infty$. For example, is a typical graph connected or does it contain a certain shape of graph?

The greatest discovery of Erdős and Rényi was that many complex graph properties appear suddenly when the probability p exceeds a critical threshold. That means if the probability is smaller than the threshold, nearly none of the graph has this property but when the probability is bigger than the threshold, nearly every graph has this property. To describe this concept mathematically, we define the critical

threshold as $p_c(N)$ where N is the number of nodes in a random graph, and then the probability of a random graph with N nodes and $p = p(N)$ connection probability has property Q satisfies:

$$\lim_{N\to\infty} P_{N,p}(Q) = \begin{cases} 0 & \dfrac{p(N)}{p_c(N)} \to 0 \\ 1 & \dfrac{p(N)}{p_c(N)} \to \infty \end{cases} \tag{2}$$

For example, when the probability reaches N^{-1}, triangle subgraphs will appear. According to Bollobás (Bollobás, 1985, Albert, R. and Barabási, A., 2002), the critical probability thresholds for the emergence of some subgraphs can be described as the form:

$$p_c(N) = N^{-z} \tag{3}$$

Figure 63 lists some basic shapes and the corresponding z.

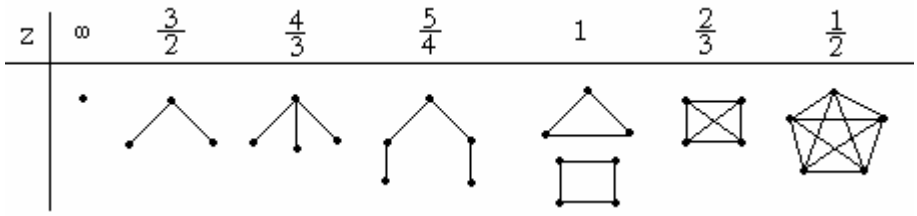


Figure 63. The critical probability threshold for the emergence of some basic subgraphs. For example, when $p \sim N^{-1}$, subgraphs of triangle will appear in random graphs.

One important note is for most graph properties, the critical probability threshold $p_c(N)$ is dependent on the size N. An alternative is to use the average number of edges connected to a point, which is also called the average degree of graph. For a random graph, the average degree $\langle k \rangle$ satisfies:

$$\langle k \rangle = \frac{2K}{N} = p(N-1) \approx pN \tag{4}$$

It is easy to show that with a fixed probability *p*, $\langle k \rangle \to \infty$ when $N \to \infty$

A significant property of random graph is the bell curve distribution of the number of edges on individual points (See Figure 64). The number of edges on a node is also called the degree and the distribution is also called *degree distribution*. According to Bollobás (1981), in a random graph $G(N, p)$, the degree k_i of a node i follows a binomial distribution:

$$P(k_i = k) = C_{N-1}^{k} p^k (1-p)^{N-1-k} \tag{5}$$

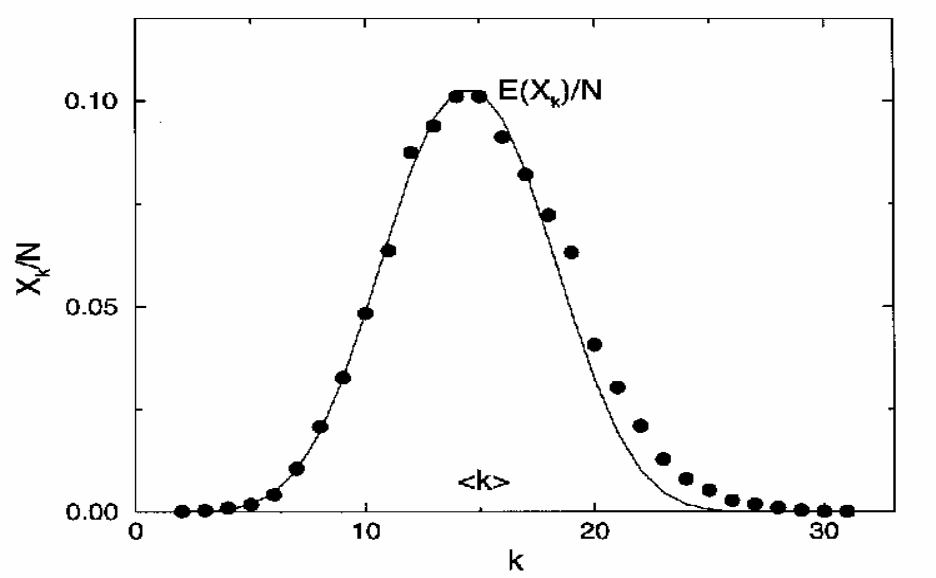


Figure 64. A typical bell curve distribution of node linkages in random graphs. The dots represent the distribution of a generated random graph with $N = 10000$ and $p = 0.0015$ (Albert, R. and Barabási, A., 2002). We can see that the deviation is small.

Other common properties about random graphs include the *diameter* and the *clustering coefficient*. The diameter of a graph is defined as the maximum distance between a pair of its nodes (Here we assume that the graph is connected and it contain no isolated subgraphs). A cluster of a graph is defined as a subgraph of a

graph where nearly every pair of nodes in the subgraph has an edge between them. To quantify this property, the clustering coefficient is introduced. For a node i in a graph, the clustering coefficient C_i is defined as:

$$C_i = \frac{2E_i}{k_i(k_i - 1)} \tag{6}$$

where k_i is the number of edges connected to node i, and E_i is the number of edges among the k_i nodes that are directly connected to the node i. The clustering coefficient C of the whole graph is the average of all individual C_i's as:

$$C = \frac{1}{N}\sum_{i=1}^{N} C_i \tag{7}$$

According to Barabási (2002) the diameter of a random network d is concentrated around

$$d = \frac{\ln(N)}{\ln(pN)} = \frac{\ln(N)}{\ln(\langle k \rangle)} \tag{8}$$

For a random graph, because the probability to have an edge between any pair of points equals p, it is not difficult to find out the clustering coefficient for any point will have an expectation of p and finally, the clustering coefficient for the whole graph also equals p (Albert and Barabási , 2002).

$$C_{rand} = p = \frac{\langle k \rangle}{N} \tag{9}$$

6.2.3 Scale-Free Network Model

Despite the mathematic beauty of the random network model, research in recent years has revealed that many real networks of large-scale cannot fit this model (Barabási, 2002). For example, the network of the World Wide Web in which the

nodes are individual html pages and the links are the hyper-links between the pages is a scale-free network (Barabási, A., Albert, R., Jeong, H., 2000).

In the random network model, the probability of a link between any two nodes is a constant, so the number of links on any node will be in a small range and the degree distribution is binomial with a bell curve as in Figure 64. With $\langle k \rangle = Np$ as the average number of links on a node, there is no node with significantly more links. However, after studying the maps of the WWW drawn by different spiders or Robots (Barabási, 2002), it was found that some pages attract many more links. Actually, the degree distribution follows a power law (see Figure 65):

$$P(k) \sim k^{-\gamma} \tag{10}$$

where γ is usually between 2 and 3 when k is large enough (Barabási, A., Albert, R., Jeong, H., 2000).

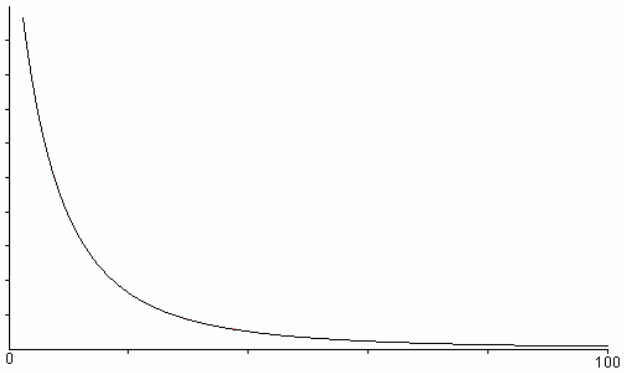


Figure 65 The power law distribution. For a scale-free network, the tail of its degree distribution follows a power law distribution similar to the curve in this figure.

In this model, nodes with an extreme number of links exist and they are called the hubs. Due to the fact that there is no obvious scale limit for the maximum number of links on a node relating to the generally limited average degree $\langle k \rangle$, people define

this kind of network as a scale-free network.

In recent years, it has been found that as well as the WWW, many other complex networks from different domains display the scale-free property. The long example list includes: the network of movie actors in which the nodes are actors and the link between two actors is the movie in which they are co-stared (Albert, R. and Barabási, A., 2000), the citation network of scientists (Redner, R. 1998) and the network of the Internet routers (Faloutsos 1999). Other examples include the biological networks in cells, where the nodes are substrates and enzymes and the links represent chemical interactions, the social network where the nodes are individuals or organizations connected by different social interactions, power supply networks, economical networks etc (Barabási, A., 2002).

There are different mathematical models that can generate scale-free networks. One is a deterministic model (Barabási, A., Ravasz, E., Vicsek, T., 2001). In this model, the network is generated by repeated steps. At the beginning, the network starts from only one node; after each step, it becomes three times larger. Mathematically it is proved that the degree distribution of this network follows power law. This deterministic model provides a set of regular, well defined scale-free networks, but it cannot be used to explain the evolution of real scale-free networks. Other models are stochastic. One is called the *extended model* (Albert, R. and Barabási, A., 2000). In this model, the network starts from m isolated nodes, then with probability p add n links, with probability q rewire n links and with probability $1-p-q$ add a new node. When creating a new link or rewiring a new link, once the first node is selected (randomly), the other node is selected based on the number of existing links on the node. That means a node with more links has a higher chance to be

connected by the new or rewired link. This phenomenon is called preferential attachment or can be simply summarized as "the rich get richer". After a few steps, the network will show the scale-free property and the degree distribution will follow the power law. There are more discussions about this model in (Albert, R. and Barabási, A., 2002).

6.2.4 Properties of Scale-Free Networks

Due to the universality of scale-free networks, the study of the properties of the scale-free networks will benefit different areas. Most of the research results are summarized in Barabási and Albert's work (Barabási, 1999, 2002). Here, we will only review a few highlighted properties of scale-free networks.

- **Small world**: The small world property means even though in a very large network, the average distance between any two nodes is relatively much smaller. This property has been independently discovered and represented in different forms. One is called the six degree of separation (Milgram, 1967). The six degrees of separation means for any two people on the earth, it requires only about six steps of acquaintances for these two people to be connected. Or in other words, the diameter of the social network is about 6. Another example is the WWW. According to Barabási (Albert, R., Jeong, H., and Barabási, A., 1999), in 1999, the estimated size of WWW is about 8×10^8; the diameter of WWW is about 18.59. The small world property does not only exist in scale-free networks, it also exists in random networks when the connectivity is high enough. According to Bollobás (1985), the diameter of a random network is proportional to the logarithm of the network's size (see pg. 144).

- **Hubs**: Studying a scale-free network, the most prominent feature one may spot is the existence of hubs; nodes with much larger numbers of links. The existence of hubs can be directly deduced from a scale-free network's power law degree distribution. However, the significance of hubs to a scale-free network is beyond the pure statistical distribution. From a hierarchy point of view, a hub can be the top level of a hierarchy with controlling power to the whole network; from a communication point of view, a hub can be the exchange center that significantly determines the communication efficiency of the whole network.

- **Clustering**: In a random network, because the probability to have a link between any two nodes is the same, it is very rare to have clusters. However, in a scale-free network, because the probability to have a link between two nodes is determined by many other factors, the emergence of clusters becomes very common. For example, in a social network, the chance of a person's two close friends to be also friends with each other is much higher than the chance of two randomly selected persons to be friends. According to Albert and Barabási (2002), many scale-free networks' clustering coefficient is much higher than that of random networks of the same size and connectivity. Those networks include WWW, movie actor network, MADLINE co-authorship, Silwood Park food web, synonyms words and power grid network etc.

- **Efficiency of Spread**: Because of the existence of hubs, information spread through a scale-free network can be much more efficient than through a random network. Here, information is only an abstract form of matters

exchanged through a network. In fact, viruses and fads are also spread through different networks. According to Barabási (2002), many virus and fads are spread in a speed as fast as light. One example is AIDS, in early 1980', little was known about this fatal disease. By now, it has killed almost 20 million people. Barabási claimed that the scale-free property of the gay sex network is one of the major reasons for the disease's amazing rate of spread. Another everyday example of a scale-free network which we often take for granted is that important information can reach a significant proportion of population in a surprising short period of time, e.g. the news regarding the September 11 terrorist attack. The efficiency of the information spread is also credited to the scale-free topological structure of the news network.

- **High error tolerance**: No matter whether it is the WWW or the social network, with billions of nodes, it would not be surprising that in each second thousands of nodes or links are not able to function properly. However, most times, we will not notice any failure of the WWW or the social network. Also every day, millions of cells in our body mutate but most people will not suffer cancer in their life time. All these are examples to show how robust a scale-free network works against local failures. According to Barabási (2002), for a random network, if you randomly remove nodes, when the number of removed nodes reaches a critical threshold, the network is broken into small pieces (Callaway, S., Newman, J., Strogatz, H. Watts, J. 2000). However, this critical threshold disappears on a scale-free network. That means a scale-free network can almost never be broken apart by randomly removing nodes.

- **Vulnerable to well organized attacks**: Despite a scale-free network being

robust against random node failures, it can have special weakest points. Failures in those points will affect major hubs and the cascade effect may collapse a considerable part of the network. Examples include the 1997 Asian financial crisis and 2003 blackout in New York and North American. Both events were triggered by relatively small incidents but the effects were spread and amplified by the hubs and finally causing crises to the whole system.

- **Winner-takes-all network**: A typical scale-free network includes hubs of different sizes. It may have one or two large hubs, a few medium hubs and more small hubs. The distribution follows the power law. The number of connections to a hub is determined by the hub's fitness to the network. In an evolving network, hubs are competing for the number of links; the fittest hub will gradually attract more number of links and this trend is called "fit-get-rich" (Barabási, A., 2002). Ginestra discovers that the model for calculating the degree distribution of a complex network can be mapped to the model used to calculate the energy levels of Bose gas model (Bianconi, G., Barabási, A, 2001). One interesting part about the Bose gas model is when the temperature is low enough it will reach a status called Bose-Einstein condensate, in which a significant fraction of the gas particles will settle to the lowest energy level while other particles scattered in other levels. The equivalent part of Bose-Einstein condensate in complex network is called winner-takes-all network (Barabási, A., 2002). In a winner-takes-all network, one hub becomes so fit that it will subdue all other hubs, dominate the whole network and finally have links to any other nodes in the network. According to Barabási (2002), Microsoft in the operations systems market is a possible example to fit in this winner-takes-all network.

We have mentioned above only a few interesting properties for general scale-free networks. For particular types of networks, it is too early to say all those properties are applicable or whether new properties may be revealed.

6.3 Dependency Networks of Java Packages

Java is a popular object-oriented computer language. Due to its cross-platform feature and its open source nature, most experiments on standard Java releases can be easily repeated by other Java programmers. This is one of the major incentives for us to select standard Java packages as the samples to investigate the topological structure of software systems' dependency networks.

In our research, we have studied the dependency networks of several publicly released Java packages as well as two Java packages developed by the author. After calculating the statistical parameters of these dependency networks, we have found that all the degree distributions of those networks follow the power law; or in other words, they are all scale-free networks. Because of the omnipresence of scale-free networks and our positive testing results, we conjecture that most dependency networks of complex software systems are scale-free.

Traditional software architecture research has paid little attention to the topological structure of complex software's dependency network. However, our experiments reveal that dependency networks of different software systems follow the same fundamental law and this result implies a new approach to the design and study the complex software systems. More specifically, we are aiming to optimize software

architectures. From the results of the previous chapter, we know the topology of a software system's dependency network can be independent to the software system's functional requirements; in this chapter we found that dependency networks of different software systems may have similar topological structure. These results suggest the existence of universal optimized architectures for general software systems regardless of the systems' functional requirements. Even though the existing dependency networks for the investigated Java packages may not be in the most optimized structures, they have been well designed, implemented and successfully used in countless software systems. Therefore, studying their architecture may help us to identify some *good features* for a well-designed software system.

Other aspects regarding the exploration of software's dependency network are related to the maintenance and reusability issues of a software system.

6.3.1 The Class Domain and the Source Code domain

Java is a relatively more purified object-oriented computer language compared to the first well-accepted OO language C++. In Java, except a few primary data types such as integer, char and double, the smallest entities are classes and their instances, called objects. Generally, an object is a run-time concept; when we look at the static structure or the blueprint of a Java system, we are looking at the classes.

In Java, classes are grouped into packages and one package may include several sub-packages and sub-packages may have sub-sub-packages. The whole system is organized in a hierarchical structure.

An interesting aspect about Java is the hierarchical structure in the class domain can be mapped directly from the domain of source code. In the source code domain, each public class (or interface) is represented as a Java source code file and each package is mapped as a directory in the local file system. Also, the hierarchical relationships between the directories match the hierarchical relationships between packages. To clarify the mapping between the two domains, let us think about a simple Java package X that includes two classes A and B and also a sub-package Y. Under package Y, there is one class C. Then we have the mapping between the source code domain and the class domain shown in Figure 66.

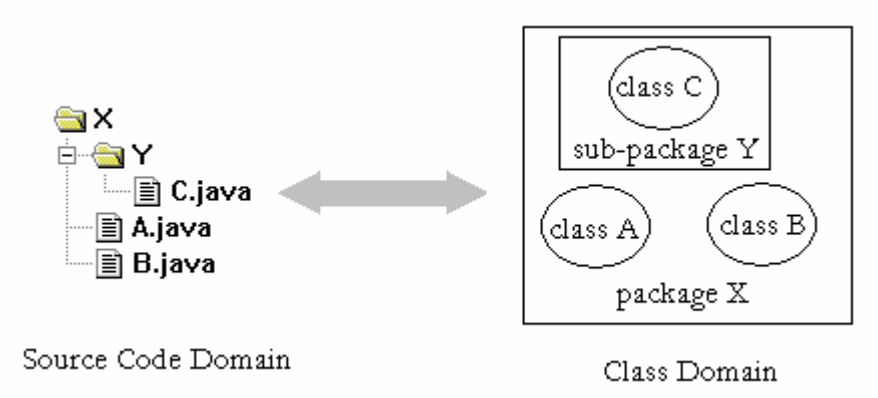


Figure 66. A simple diagram to show the bi-directional mapping relation between the source code domain and the class domain of Java systems.

Because of the bi-directional mapping relationship between the class domain and the source code domain, the topological relationships as well as the dependent relationships of a Java system[18] can be retrieved from Java source code.

[18] A Java system is equivalent to a Java package if we ignore the effect of Java's CLASSPATH setting by moving all the required Java resources under one root directory.

6.3.2 The Dependent Relationship and the Dependency Network

Nearly every single Java class needs other classes to work properly; or it depends on other classes. The dependency can be direct or indirect. Here, we only focus on the directly dependent relations. Then from individual dependent relationships, we can draw the dependency network of the whole system.

Normally, there are three different types of directly dependent relationships. The first is inheritance, which comes as one class inherits another class (or implements an interface). The second is inclusion, which means one class includes members of another class. The last type is reference, which means another class might be referred to in the methods of the first class, either by being passed as a parameter or by being used as local variables in the methods.

```
public class class1 inherits class2 {
    ....
    ....

a Inheritance relationship

public class class1 {
    class2 x;
    ....
    ....

b Inclusion relationship

public class class1 {
    ....
    ......
    public void foo(class2 var,...) {
      .....
      .....

c Reference relationship
```

Three different types of Java directly dependent relationships. a. Inheritance relationship, b Inclusion relationship and c. Reference relationship.

The examples above show the three different types of directly dependent

relationships. In either type of relationship, it is clear that **class1** depends on **class2** to work or to be able to compile. In other words, without **class2**, **class1** cannot function at all, so it is a direct and explicit dependency.

A Java dependency network is like a normal network and it includes nodes and links. A node is a public class (or a public interface) and a link between two nodes represents the directly dependent relationship between the two classes. The link can be one directional or bi-directional that means the two classes depend on each other. Because we are discussing the topological features of the dependency network, sometimes we will be concerned by the direction of a link but sometimes we will not. Further discussions regarding the topological features of the dependency network will be presented in the following sections.

6.3.3 The Testing Results

We have developed a tool (introduced in Chapter 7) to explore the topological structure of Java packages. There are 5 open-sourced Java packages, delivered by Sun in the J2sdk1.4.1_02 release and 2 small packages, *classnet* and *netp*, which are created by the author, have been tested. The 5 open-sourced packages are *java*, *java.awt*, *javax*, *org* and *com*. The general statistical features of the 7 packages are shown in Table 2.

Package	n	l/l_d	k	*std*	p	C	$\bar{d}$	$d_{\max}$	γ_{in}	γ_{out}	γ_{tot}
java.awt	345	1721/151	4.99	12.72	0.03	0.63	2.99	7	3.74	3.04	2.29
java	1172	9453/374	8.7	31.45	0.01	0.57	2.58	6	3.15	2.86	2.08
javax	909	4683/124	5.15	15.89	0.01	0.61	4.03	13	3.87	3.79	2.27
com	642	2535/132	3.95	13.43	0.01	0.61	2.83	7	4.13	4.05	2.66
org	1083	7286/172	6.73	31.01	0.01	0.61	2.48	6	4.29	2.83	2.15
netp	66	120/6	1.81	3,88	0.06	0.70	3.03	8	5.45	5.45	4.78
classnet	15	22/3	1.47	2.30	0.21	0.81	2.52	5	8.72	3.37	3.25

Table 2 The statistical figures of the 7 tested Java packages. From these figure, we can see the dependency networks are scale-free networks rather than random networks

In Table 2, n is the number of nodes; l is the number of links; l_d is the number of bi-directional links; k is the average number of links on each node; *std* is the standard deviation of the number of links per node; p is the probability to have a link between two arbitrary nodes if the targeted network is a random network with the same n and l (the value P also equals the clustering coefficient on a random network). For each network, we also calculate the clustering coefficient C, the average distance between any two nodes $\bar{d}$, the maximum distance $d_{\max}$ and the power law parameter γ. Because the dependent relationship has direction[19], we use *in* and *out* to represent the depended and depending links and *tot* as the total number of links.

From Table 2, we find the link number's standard deviation *std* is about 3 times the

[19] In Table 1, except l_d, γ_{in} and γ_{out}, other data are calculated without considering the directions of the links

average k. It is significantly different from that of a random network with a normal distribution (in a normal distribution, the standard deviation equals the mean). We also noticed the clustering coefficient C is about 20 times greater than the probability p. Because in the random network model, the clustering coefficient equals the probability p (see pg. 144), we believe the target networks cannot be explained by the random network model.

Now we focus on the dependency network of *java.awt*. The network is shown in Figure 67. In this network, each public class or interface in package *java.awt* or its sub packages is represented as a node; a line between two nodes represents the dependent relationship between the two corresponding classes. The size of a node is determined by the number of links on the node. We notice that there is a few large nodes compared with many small and some medium size nodes. This is the most significant indicator for a scale-free network. Figure 68 is an improved version of Figure 67. Figure 69 and Figure 70 show the degree distributions and the node distance distribution of the network.

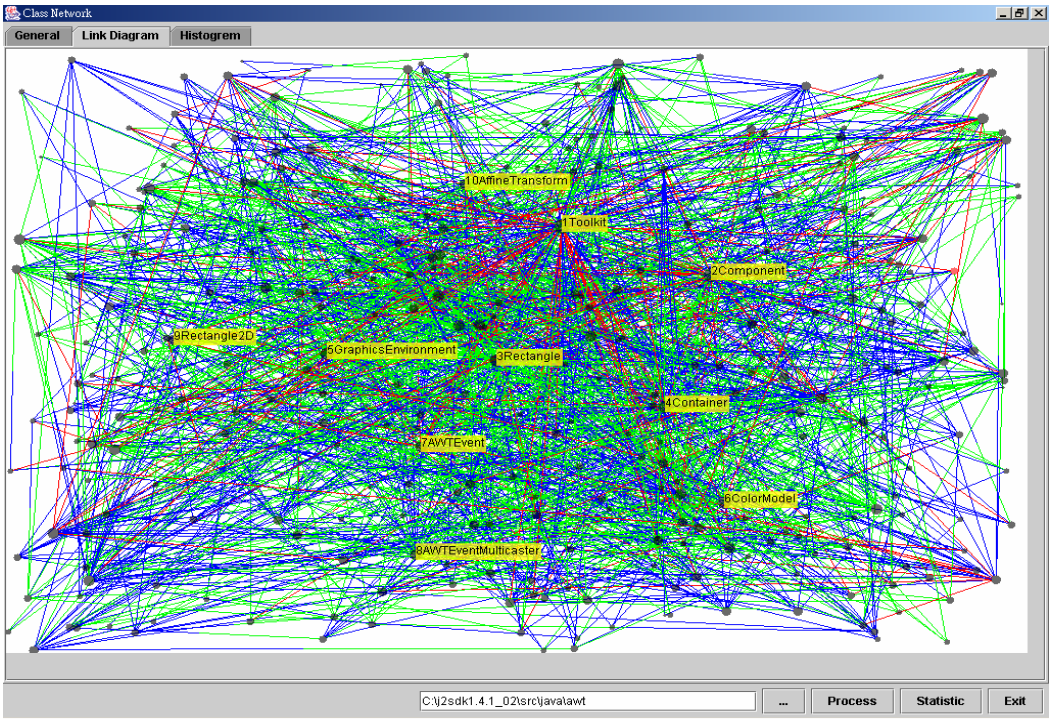


Figure 67. The dependency network of package *java.awt*. The labels show the top 10 nodes with the most number of links.

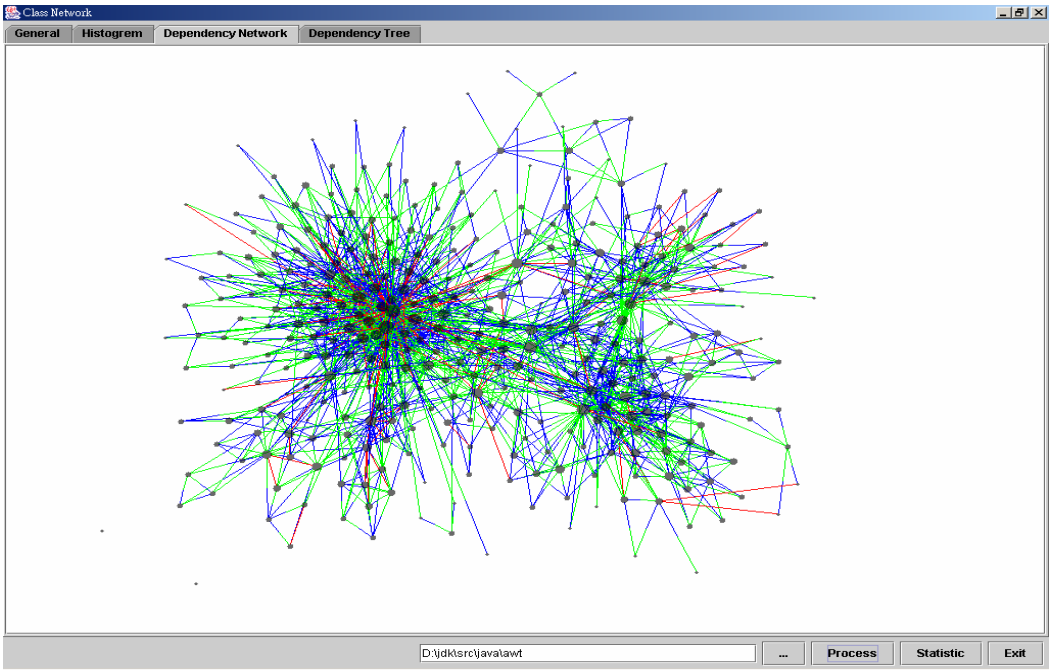


Figure 68. The dependency network of package *java.awt*. In this diagram, the position of nodes are rearranged by force-directed algorithm.

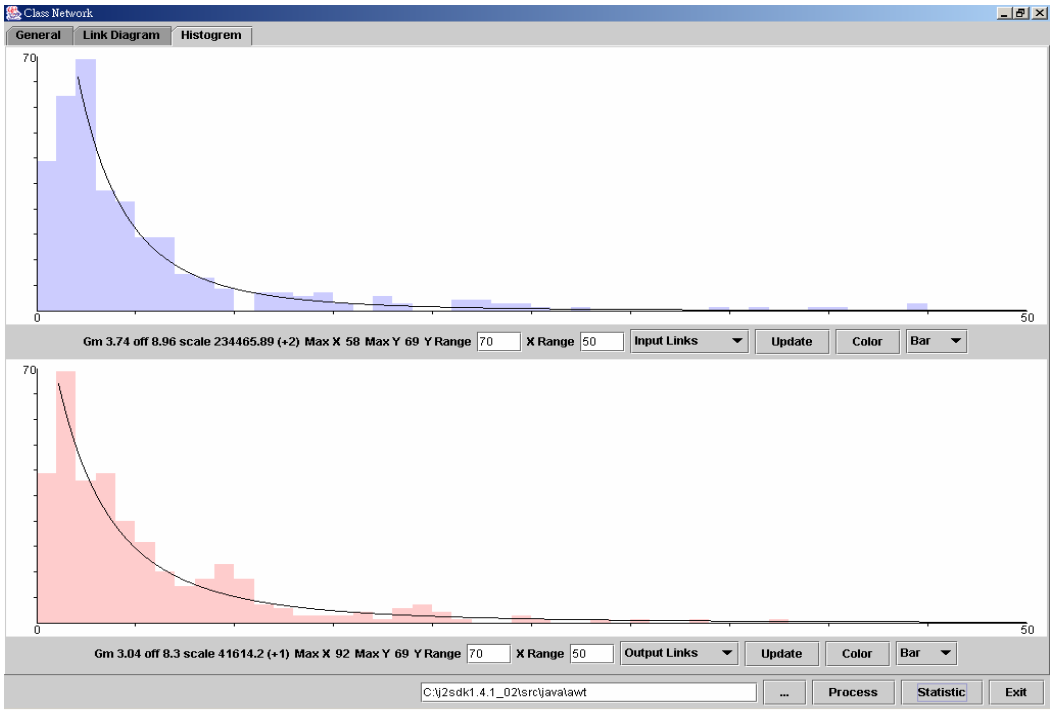


Figure 69. The histograms of income and outcome link degree distributions of package *java.awt*. The curves are drawn based on power law distribution; the parameters are estimated by using maximum likelihood algorithm.

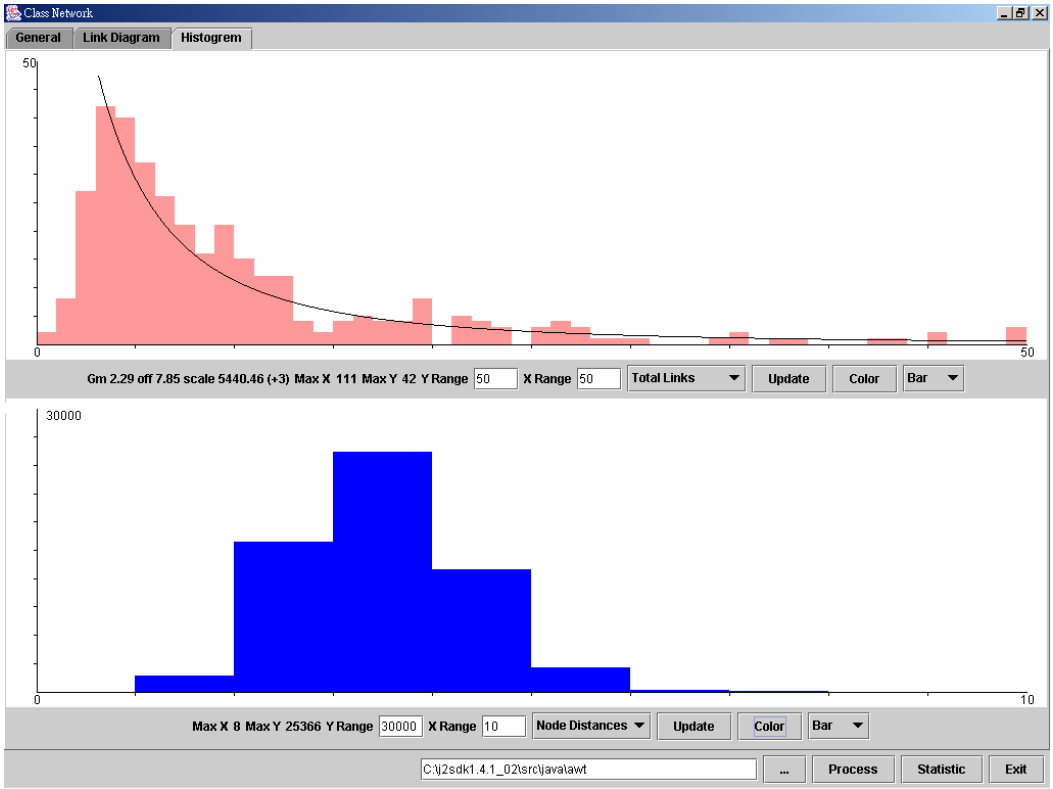


Figure 70. The top diagram is the histogram of the degree distribution of the total number of links on individual nodes. The bottom diagram is the distribution of the distance between nodes.

In Figure 67, the positions of the nodes are randomly selected; in Figure 68, we have implemented the force-directed algorithm (Cruz and Tamassia, 1998) to rearrange the positions of the nodes and the new diagram looks less complex and more organized. Also from Figure 68, we can observe a few large clusters in the network.

The force-directed algorithm we have implemented in our tool includes three different kinds of forces. The first force is created by the links; each link will pull the two linked nodes together. The second force is a kind of electronic force; each node is a body charged with the same electricity and they repel each other. The repelling force between two nodes is determined by their geometrical distance on the diagram. The third force comes from the edges of the diagram and it tries to drive the nodes into the middle part of the diagram. Driven by the three different types of forces, each node moves to a new position with lower potential energy level and gradually, the network will evolve into a balanced and more organized form.

Figure 69 shows the degree distribution of the income links and the outcome links. In a Java dependency network, if **class a** depends on **class b**, there is a link between them; to **class a**, it is an outcome link and to **class b**, it is an income link. From Figure 69, we can see both distributions follow the trend of power law. The curves are drawn by the function:

$$y = A(x - x_0)^{-\gamma} \quad x \geq x_{off} \tag{11}$$

The parameters are inferred by the criteria of Minimum Mean Square Error (McClave, 1997). For the income links, $A = 234465$, $x_0 = 6.96$, $\gamma = 3.74$ and

$x_{off} = 2$. The degree distribution of the outcome links has similar results. The degree distribution of the total number of links is shown in the top part of Figure 70. The total number of links on a node includes the income links and the outcome links. However, if two classes, **class a** and **class b**, depend on each other, there is only one bi-directional link between them rather than two links and the link number on each node is one. The parameters of the three different distributions on all the tested dependency networks are list in Table 3.

Packages	Income link number			Outcome link number			Total link number		
	A	x_0	γ	A	x_0	γ	A	x_0	γ
java.awt	234465	6.96	3.74	41614	7.3	3.04	5440	4.85	2.29
java	108587	5.37	3.15	97263	6.69	2.86	11910	5.64	2.08
javax	1.3E6	9.23	3.87	6.8E5	7.88	3.79	13740	6.09	2.27
org	3.3E6	8.51	4.29	74586	6.05	2.83	12462	5.75	2.15
com	2.0E6	9.64	4.13	1.7E6	9.81	4.05	35766	8.25	2.66
netp	7.9E6	10.31	5.45	1.1E7	11.2	5.45	5.0E5	6.53	4.78
classnet	2.8E8	8.57	8.72	1620	3.51	3.73	7872	9.17	3.25

Table 3. The parameters of the power law distribution of the tested Java packages' dependency network

From Table 3, we can see γ is larger than 2 and when the number of links increases, it decreases. After examining the degree distributions for each dependency network, we find that all of them show clear signs of power law, so all the dependency networks are scale-free. The diagrams of the dependency networks and the corresponding degree distributions for other 6 tested packages can be found in

appendix E.

6.4 Scale-Free Networks and Sorting Algorithms

In the first part of this chapter, we have presented our discovery that the component dependency networks of all the examined Java packages are scale-free. This discovery shows the connection between software evolution and the scale-free networks. However, for a large software system, the evolution process may take many years and some of the evolution details may not be documented, so it is usually very difficult to trace the evolution process backwards. Also, there are so many unpredictable factors affecting the software change that it is impossible to repeat the same evolution process and generate the same network. The unrepeatability and the uncompleted information make the research of the evolution of CINs very difficult. In the second part of this chapter, we use sorting algorithm to investigate network evolution and discover the relationship of scale-free network and optimized sorting algorithms. This result might imply a new approach to investigate network evolution and software change.

6.4.1 Introduction

The study of sorting algorithms has been one of the most important research topics in computer science. Many sorting algorithms have been invented. Most require comparison of the key values of records (Knuth, 1997c). In fact, for general sorting algorithms, the comparison of key values is inevitable. If we draw each record in the original sequence as a node, and each comparison of two records as a link between the two nodes then the direction of the link indicates the comparison

result. The sorting process yields a directed network.

For example, considering a sequence of 5 distinct integers: $n_1 n_2 n_3 n_4 n_5$[20], if we compare 7 pairs and get the results: $n_1 < n_2$, $n_1 < n_5$, $n_2 < n_3$, $n_4 < n_1$, $n_4 < n_5$, $n_5 < n_2$ and $n_5 < n_3$, then the integers and the comparison results can be drawn as a network shown in Figure 71.

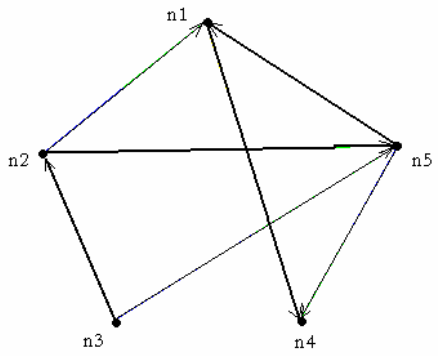


Figure 71 The sorting network of $n_1 n_2 n_3 n_4 n_5$. From this diagram, a unique, directional path $n_3 n_2 n_5 n_1 n_4$ that goes through each node once can be found.

The pairs of nodes used in the comparisons are selected based on the particular sorting algorithm. From Figure 71, we may find a unique path $n_3 n_2 n_5 n_1 n_4$ that travels in a single direction and visits each node once. The resulting sequence is sorted[21].

---

[20] General sorting algorithms are used to rearrange each record R_i based on its key K_i. However, in this thesis, when it does not make any confusion, we will not distinguish the record and the key, and without losing the generality, they might be represented as an integer n_i or r_i.

[21] From the comparison set, we can simply integrate each pair into a tree. This yields a tree that can be pruned along the way to generate the ordered sequence

The example above, suggests that the process of sorting a sequence can be mapped to the evolution of a network. We call this network a sorting comparison network (SCN). The topological properties of the SCNs of a number of different sorting algorithms have been studied. What we have found is that SCNs of "optimized" sorting algorithms, (where the numbers of comparisons is close to the theoretical lower bound $\lceil \log(n!) \rceil$), are scale-free networks.

Weaving a network is a universal methodology for constructing a system that links individual components to achieve a high-level function. Sorting a sequence of numbers is a general problem of this kind that requires the involvement of each component (record) to achieve a higher-level function (sorting). The discovery of the association of "optimized" sorting algorithms and scale-free sorting comparison networks backs up the conjecture that the scale-free network is a universal optimized topological structure for complex networks.

6.4.2 Sorting and Sorting Algorithms

Sorting is the process to arrange a sequence of records in a certain order. *In normal human usage of data the order of data is very important since order suggests critical relationships* (Lorin, 1975). In computer science, the importance of sorting cannot be over estimated because the ordered data can make many other software processes and some hardware work more efficiently.

Generally, sorting can be classified into *internal sorting*, in which the records are kept entirely in the computer's high-speed memory, and *external sorting* where not all the data can be access as quickly as in internal sorting. In our research, because we only

consider comparisons of key values from a theoretical point of view, we will limit our discussions to internal sorting.

For internal sorting, a sorting method can usually be classified into one of the following 5 groups: *sorting by insertion*, where part of the sequence is sorted and the unsorted records will be inserted into the sorted part one by one in the proper position; *sorting by exchange*, where pairs or records are compared and exchange two records if they are not in the right order; *sorting by selection*, where the record with the maximum or the minimum value is selected to put at the beginning point of the sorted sequence and the record with the maximum or the minimum value in the remaining records is selected to put after the previous selected records; *sorting by merging*, where the original sequence is separated into sub-sequences and after each sub sequences are sorted, they will be merged into one complete sorted sequence and *sorting by distribution*, where distribution of the records' index is calculated to determine the suitable position of the records (Knuth 1997c).

In this thesis, we have studied 5 common sorting algorithms. They are bubble sort (sorting by exchange), heapsort (sorting by selection), quicksort (sorting by exchange), binary insertion sort (sorting by insertion) and merge insertion sort (sorting by merging). Even though the 5 sorting algorithms belong to different sorting categories, all of them use records' key values. In this thesis, we will pay less attention to other operations such as exchange, insertion and merging. Details of the 5 sorting algorithms can be found in Knuth's book (1997c). A brief introduction is attached in Appendix A.

6.4.3 Sorting Comparison Networks

A sorting process can be mapped to a network or graph is not a new concept. In Knuth's book (Knuth 1997c), a similar idea has been introduced to illustrate the process of the merge insertion sorting algorithm. However, the focus of a sorting comparison network (SCN) is different. Traditionally, the graph or the network associated with a sorting process is used to illustrate the sorting algorithm. In this thesis, the sorting comparison network only records all the comparisons of the items of the source sequence.

Assume $r_1 r_2 \ldots r_n$ is a sequence of n numbers and we have $r_i \neq r_j$ when $i \neq j$; $1 \leq i, j \leq n$. The sorting process is to rearrange the sequence to generate a new sequence $r_{S_1} r_{S_2} \ldots r_{S_n}$, so we have $r_{S_i} > r_{S_{i+1}}$, while $1 \leq i < r$.

No matter what sorting algorithm it is used, the SCN can be created by the following steps:

1. Draw n separate nodes on a plane to represent the n numbers.
2. When 2 numbers are compared, we draw a link between the two corresponding nodes. The direction of the link represents the comparison result.
3. During the sorting process, more links will be added to match the new comparisons. The sorting comparison network ignores other sorting operations such as swapping and insertion etc.
4. When the sorting process is finished, the corresponding SCN is complete.

A complete SCN has enough information to arrange the whole set of nodes in either ascending or descending order; that means from a complete SCN, we can find

a unique path that goes through every node once and all the links on this path follow the same direction. We call this path the *sorting path* and each link on this path is called a *contiguous link* because it connects two contiguous records. Other links are called *non-contiguous links*. Suppose an SCN has n nodes and no two nodes have the same key value; then the contiguous links and non-contiguous links have the following properties:

1. There are totally $n-1$ contiguous links and $(n/2-1)(n-1)$ non-contiguous links.
2. If the SCN has enough comparison information to totally sort the corresponding sequence, all the $n-1$ contiguous links must be included in the SCN, because if there is a contiguous link that is not included in the SCN, we will not be able to determine the order of the two nodes that are connected by this link. The reason is that the comparison results from any other nodes to the two contiguous nodes are identical.
3. The minimum number of links of a SCN to provide the information to completely sort the sequence is $n-1$.
4. Any sorting algorithm, before it finishes the sorting process, must discover all the $n-1$ contiguous links.

From the properties of the contiguous links, the sorting problem is equivalent to discovering the $n-1$ contiguous links from a n node network by adding links. Different sorting algorithms use different stratagems to build links. The theoretical lower bound of the link number for any algorithm under the worst situation is $\lceil \log(n!) \rceil \approx n\log(n)$ (Knuth 1997c). If we use a random network model, which means we select links randomly, the total number of links needed before we identify the sorting path depends on chance. In the best case, the first $n-1$ links selected are the $n-1$ contiguous links, but in the worst case, we may select the right link in the last selection. The average number of links required by the random model

is $(n-1)(n^2-n+2)/2n$. Table 4 summarizes the number of links in 4 different special situations.

#	Situation	Number of Links
1	Theoretical lower bound	$\lceil \log(n!) \rceil \approx n\log(n)$
2	Random (best case)	$n-1$
3	Random (worst case)	$n(n-1)/2$
4	Random (average)	$(n-1)(n^2-n+2)/2n$

Table 4 Number of links required to find the sorting path.

In Table 4, #2 and #3 are obvious. The proof for #1 can be found in Knuth's book (Knuth 1997c). The proof for #4 is in Appendix B. According to #4, by using the random network model, even though under the best case situation, only $n-1$ links are required to solve the sorting problem, the average number of links is $(n-1)(n^2-n+2)/2n$. This is close to the worst situation and much larger than the theoretical lower bound. This result clearly illustrates the inefficiency of the random network model.

6.4.4 The Comparison of the 5 Sorting Algorithms

The number of links of the 5 different sorting algorithms' SCN has been simulated and the result is presented in Figure 72.

The x-axis is the number of nodes (from 16 to 1024). The y-axis is the logarithm of the number of links. In order to make the curves smooth, for each number of nodes (from 16 to 1024), the average of the links of SCNs of 10 independent

random sequences were used for the simulation. Figure 72 also includes 4 reference lines. They are $y = n$, $y = \lceil \log(n!) \rceil$, $y = n\log(n)$ and $y = n \times (n-1)/2$.

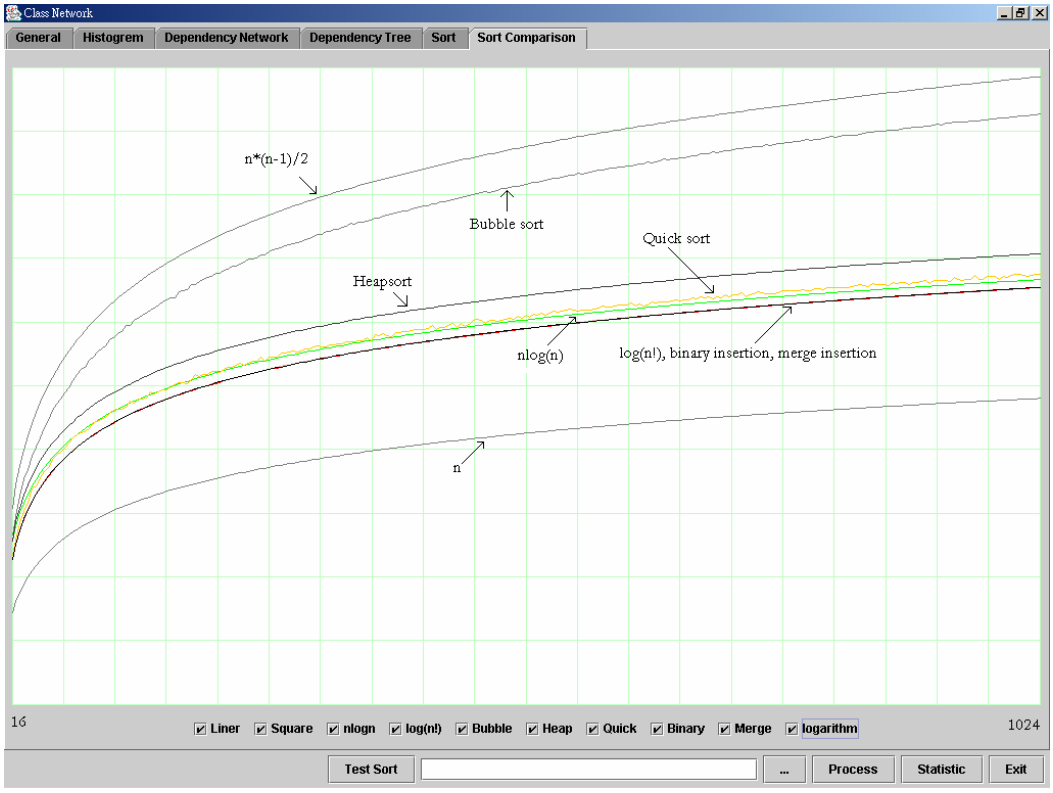


Figure 72. The number of links of the SCNs of different sorting algorithms.

From this diagram, it is seen that the binary insertion and merge insertion are very close to the theoretical lower bound $y = \lceil \log(n!) \rceil$. Bubblesort is much worse than the other sorting algorithms. Actually it is close to the worst case $y = n \times (n-1)/2$. Heapsort and quicksort are also close to the curve of $y = \lceil \log(n!) \rceil$, but they are worse than the lower bound's approximation $y = n\log(n)$. In Figure 72, quicksort is better than heapsort regarding to the number of comparisons, but quicksort is not as stable as other sorting algorithms. The real SCNs generated from the 5 sorting algorithm are attached in Appendix C.

6.4.5 The Degree Distribution of the 5 Sorting Algorithms' SCN

Figure 73 -- Figure 77 are histograms of the degree distributions of the SCNs generated by the 5 sorting algorithms. The x-axis is the number of links on each node, and the y-axis is the number of nodes. These degree distributions are based on 1000 independent random sequence of 256 records.

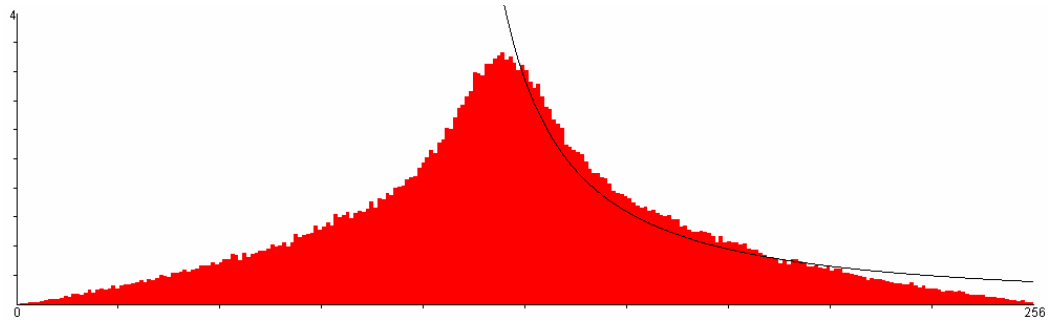


Figure 73. The average degree distribution of SCNs of bubble sort (sequence length 256, sample size 1000).

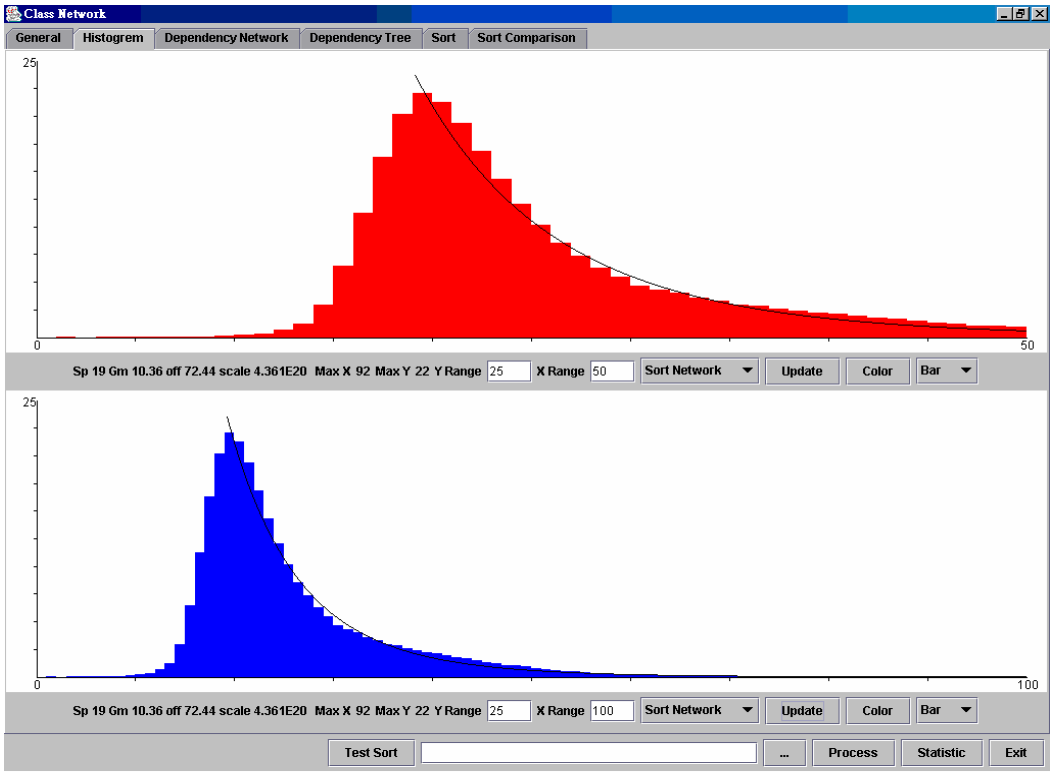


Figure 74. The average degree distribution of SCNs of heapsort (sequence length 256, sample size 1000; the top diagram shows the distribution in the lower range and the bottom diagram shows the

distribution of the whole range).

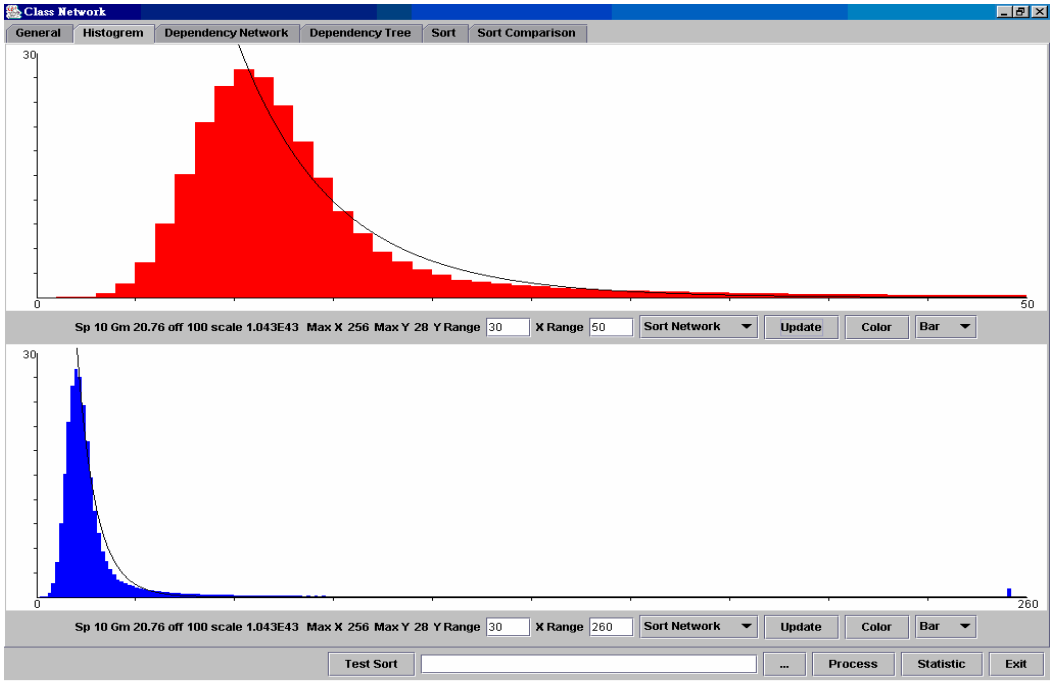


Figure 75. The average degree distribution of SCNs of quicksort (sequence length 256, sample size 1000; the top diagram shows the distribution in the lower range and the bottom diagram shows the distribution of the whole range).

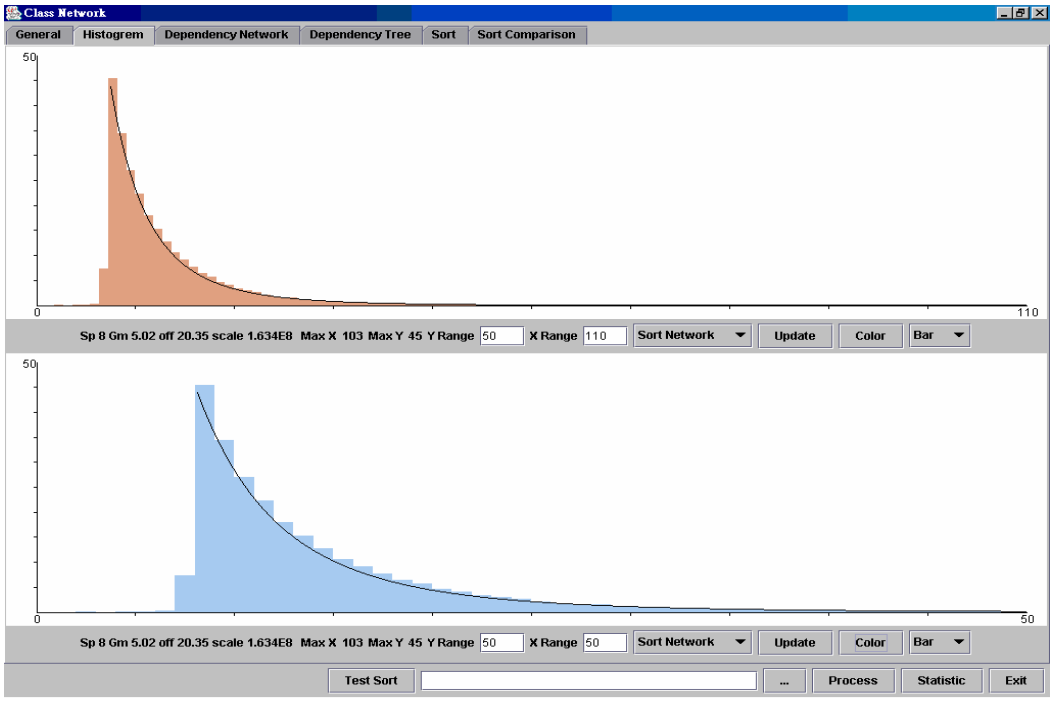


Figure 76. The average degree distribution of SCNs of quicksort (sequence length 256, sample size 1000; the top diagram shows the distribution in the lower range and the bottom diagram shows the distribution of the whole range).

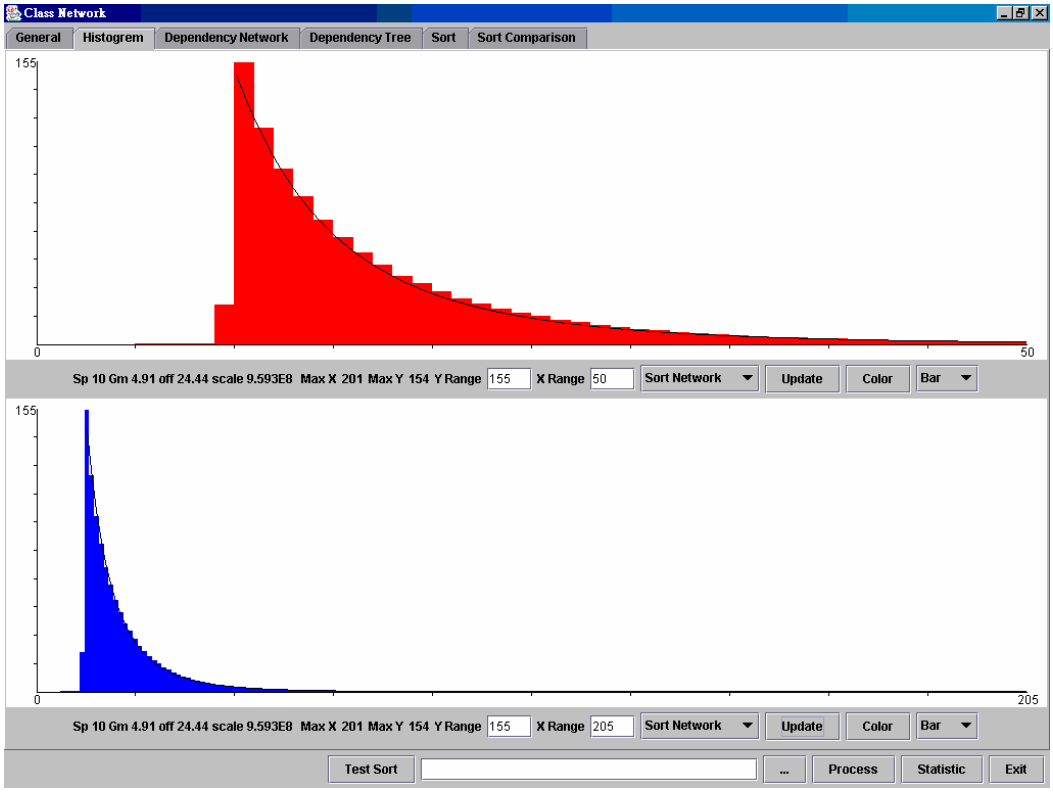


Figure 77. The average degree distribution of SCNs of binary insertion (sequence length 1024, sample size 1000; the top diagram shows the distribution in the lower range and the bottom diagram shows the distribution of the whole range).

From these distributions, it can be found that except for the bubble sort, all the other histograms follow the power-law distribution at the tails. In other words, they are scale-free networks. But the distribution of bubble sort is close to a normal distribution. Also, the power-law property in Figure 76 and Figure 77 (which are created by binary insertion and merge insertion) is more obvious than that in Figure 74 and Figure 75 (which are from heapsort and quicksort). Thus, we form a conjecture that when a sorting algorithm requires less comparisons or it is more optimized with regard to number of comparisons, the associated sorting comparison network is more likely to be scale-free. More testing results and the degree distributions can be found in Appendix D.

6.4.6 Summary

In the previous section, we have examined the sorting comparison networks of several different sorting algorithms and found that the optimized algorithms tend to generate scale-free sorting comparison networks. This result indicates that the scale-free property applies to SCNs of sorting algorithms that are close to optimal in terms of comparisons. Because sorting is a general computational strategy, the link with scale-free networks, implies that the scale-free form may represent a universal optimized structure for general networks.

Several mathematical models have been proposed for generating scale-free networks(Albert 2002, Barabási 2001 and Manna 2003a, 2003b), among them, the most widely used one is the preference model, which suggests new nodes are more likely to link to nodes that already have more links. Under these rules a network will evolve into a scale-free network. The scale-free property of many real complex networks can be explained by this model. However, some scale-free networks can NOT be explained by this rule. For example, a SCN generated by binary insertion algorithm, when we try to find the right position of a new number among the already sorted numbers, we compare the new number with the number in the middle point of the sorted sequence. Even though the number at the middle point may have more links, the reason to select is not its link number but of its location. Another interesting fact is that even though that many real life complex networks are scale-free, some of them are not. One good example is railway networks. A large airline network is high likely to be scale-free, the likelihood of a large railway network being scale-free is very low (Barabási 2003). At this stage, the weakest

precondition for a network to evolve into a scale-free network is unknown. The deep reason why so many large networks are scale-free is still a mystery. The study of sorting comparison network may shed some light on this mystery.

The last point we will address is the final form of SCNs. An SCN is evolved during the process of sorting. In the evolution process, new comparison results may cause old comparison results to become redundant. This means we can remove some old links without affecting either the determination of new links or the final sorted results. If we do not remove the redundant links, the optimized SCN is still a scale-free network. However, if we remove the redundant links, the final form will only contain the $n-1$ contiguous links. It is a linked list or in general a tree. With regard to the possible minimum number of links, a tree is the optimized form of a connected network. In the real world, many optimized systems use tree-structured networks, e.g., hierarchical management systems and normalized software dependency networks that will be addressed in the next chapter. Therefore, we suspect that under some preconditions, the optimized networks may take the form of a tree.

6.5 Discussion

6.5.1 The Origin of Scale-Free Property

Why is it that many complex networks are scale-free, what is the mathematical model and how could we justify it? Some people explain this phenomenon by using a dynamical preference model (Barabási 2000, Manna 2003). A brief explanation of that model is:

A complex network is evolved from a primary network with a single node or a few nodes. When a new node is added into the network, at least one link must be created to connect the new node to the existing nodes. If the probability to connect the new node to a particular node is the same among the existing nodes, then gradually, the early nodes will have more number of links than the later nodes. However, this model cannot explain why a few nodes grasp much larger number of links than the others, so a modified version has been introduced. In the new version, the probability to create a link between the new node and an existing node depends on the number of links on the existing node. In other words, the more links a node has, the higher chance for it to be connected with the new nodes. The probability to link to a certain node can be linear or non-linear proportion to the degree of the node.

In software engineering, this explanation has some validation. When a system becomes large, programmers tend to use the most familiar classes and this habit will make those nodes more popular (from a network's point of view, those classes are hubs). There actually exist two opposite types of "hubs" in a dependency network. One is of the primary classes such as class "String" and interface "IOException". These classes are usually simple but can be reused in many different situations. Another type of "hubs" is the controlling or system classes such as "ClassNetFrame" in package "classnet", "NetpCanvas" in "netp" or "System" in "java". These classes are usually large and complex. Some of them are systems that control many other components or sub-systems, some are working as bridges to link different parts in a system. These classes usually provide many functions and are closely linked to many other important classes in the system, so new classes are

more likely to create links to these classes to retrieve different kinds of resources.

6.5.2 The Order of Importance

To study a new software system, one of the most challenging tasks is to determine where to start. A system may include hundreds or thousands of classes, without a well organized tutorial, it can be very difficult for other people to understand, to use or to maintain the system. The dependency network provides an overall view to inspect the system as a whole. But in order to really understand a system/package and reuse it, functionalities of individual classes must be explored.

According to the discussion in the previous section, we know there are usually two types of classes that have many numbers of links. One is the primary classes that can be frequently reused and the other type is the system or controlling classes. From programming experience, we know both types of classes are very important. Therefore, the number of links affords a simple criterion to justify the importance of classes.

Because of the scale-free property of the dependency network, only a few nodes have a large number of links while most other nodes have only a small number of links. Focusing on the classes with the most number of links can be a good strategy to study a new software package/system.

6.5.3 Progressive Activities and Anti-regressive Activities

It is well known that there are two different types of activities in software development cycle: the progressive activities and the anti-regressive activities (Lehman, 1974, 2001). Progressive activities directly contribute to the implementation of software's functionalities but also increase the software's complexities or entropy, which, when reaching a certain level without control, may cause the system to be difficult to maintain or continue with development. Anti-regressive activities are defined as those activities that do not directly increase the functions of a software product but improve its manageability, so that the software itself has the potential to grow in the future. This kind of activity includes updating of system documentation, rewriting some modules and complexity control (Lehman, 2001).

From the architecture's point of view, the progressive activities usually increase the complexity of a dependency network by adding new nodes and new links. However, some of the anti-regressive activities such as re-constructing the architecture, re-writing some models may result in removing of some links and nodes in the dependency network and eventually reducing the complexity of a dependency network. Based on our previous study, the topological structure of software's dependency network can be independent to its functional requirements (Wen 2003, 2004). The anti-regressive activities can, under extreme conditions, make the system's dependency network to the simplest forms which are trees. Of course, in the real project, this seldom happened.

The subtle relationship between the complexity of a dependency network and the

progressive and anti-regressive activities implies the possibility to use the dependency network's complexity as an indicator or a guideline for the software's manageability or the efficiency of the anti-regressive activities.

6.5.4 Optimized Architecture

Without a clearly defined criterion, there is no possible way to discuss the optimization. A software system, just like any other kind of organized system, includes two opposite elements: freedom and restriction or chaos and order. For a dependency network, there are two extreme statuses: One is with the maximum number of links and the other is with the minimum number of links. The first is actually a complete network, which means for any two arbitrary nodes in the network, there is a link between them. The other form is a tree, which means between any two nodes in the network, there is a unique path between them[22].

In a network, when a message is transferred from one node to another node, it can go through one of many possible paths. The number of possible paths can be defined as the freedom within the network. In a complete network, the number of possible paths reaches the maximum or has the maximum freedom. In a tree, because there is only one path between any two nodes, the freedom comes to the minimum. In a network with less freedom, there are fewer choices for the message passing between nodes, or it is less ambiguous or easier to manage. However, too few links may cause some links or nodes to be over-loaded and, in certain cases, may cause some paths to be too long and eventually affecting the efficiency. An

[22] A path is a sequence of connected links with no node occurring more than once on it.

optimized architecture of the dependency network needs to balance these two factors.

The association between the scale-free networks and the optimized sorting algorithms also suggests that for complex systems, an optimized architecture may have the scale-free property.

6.5.5 Dependency Network and Dependency Tree

In the previous section, we have examined the dependency networks of several software systems written in Java. Even though a dependency network contains all the dependent relationships within a system, due to the complexity of the diagram, except the statistic figures, dependent relations on a single class can hardly be clearly traced from that diagram. In this situation, a new type of diagram, *dependency tree* is introduced.

The concept of a dependency tree is not an invention of this thesis; it is quite similar to the concept of architecture slice used by Zhao (2002). A dependency tree is actually a different view of a dependency network. In a dependency network, each class is directly dependent on other classes, and other classes may be directly dependent on further more classes. These relationships can be simply represented as a tree. In this tree, each node represents a class, and its child nodes are classes that are directly dependent on the class of the parent node. The root of the tree can be any selected class from the dependent network. Because several classes may be dependent on a single class, a class could have multiple instances in a dependency tree. In a dependency network, if the dependency relations of several classes form a

circle, the dependency tree will expand infinitely. To solve this problem, in a dependency tree, only one instance of each class can have child nodes. Figure 78 shows the dependency tree of class **Vector** in java package **java.util**.

In this tree, each node represents a class or an interface. The size of a node is determined by the number of child nodes under that node. If a class has several corresponding nodes, only one of the nodes will have child nodes and the other nodes are marked by a dash under the nodes.

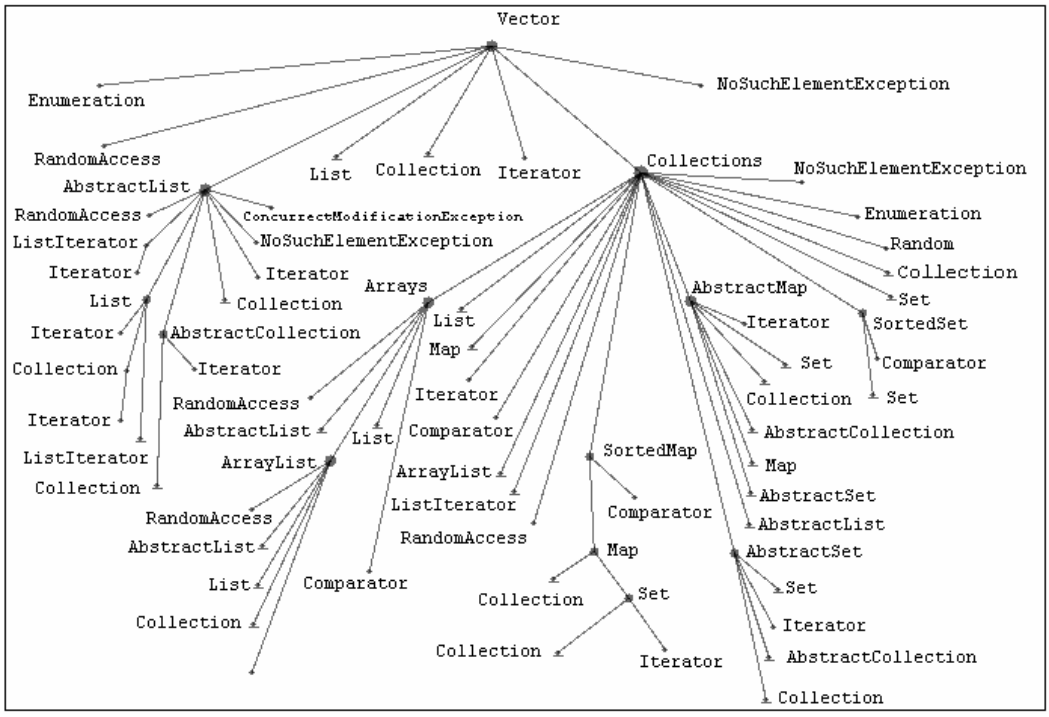


Figure 78. The dependency tree of the class **Vector** in package **java.util**.

Different from a dependency network, through a dependency tree, all the classes that are directly or indirectly dependent on the root class can be easily detected. By using dependency trees, we can retrieve relatively smaller groups of classes from a large class set. From these smaller groups, architecture problems can be spotted and solved relatively easier. For example, we can search certain links that can be removed to significantly reduce the depth of a dependency tree. A dependency tree, even in a

medium sized package such as java.awt (with 345 classes) can be large and deep. An example is given in Figure 79, which is the dependency tree of class **Button**.

A dependency tree can also be drawn in a reversed method so that all the nodes are directly dependent on their parent node. From the reversed dependency tree, we can figure out if the root node class is modified, what other classes may be directly or indirectly affected.

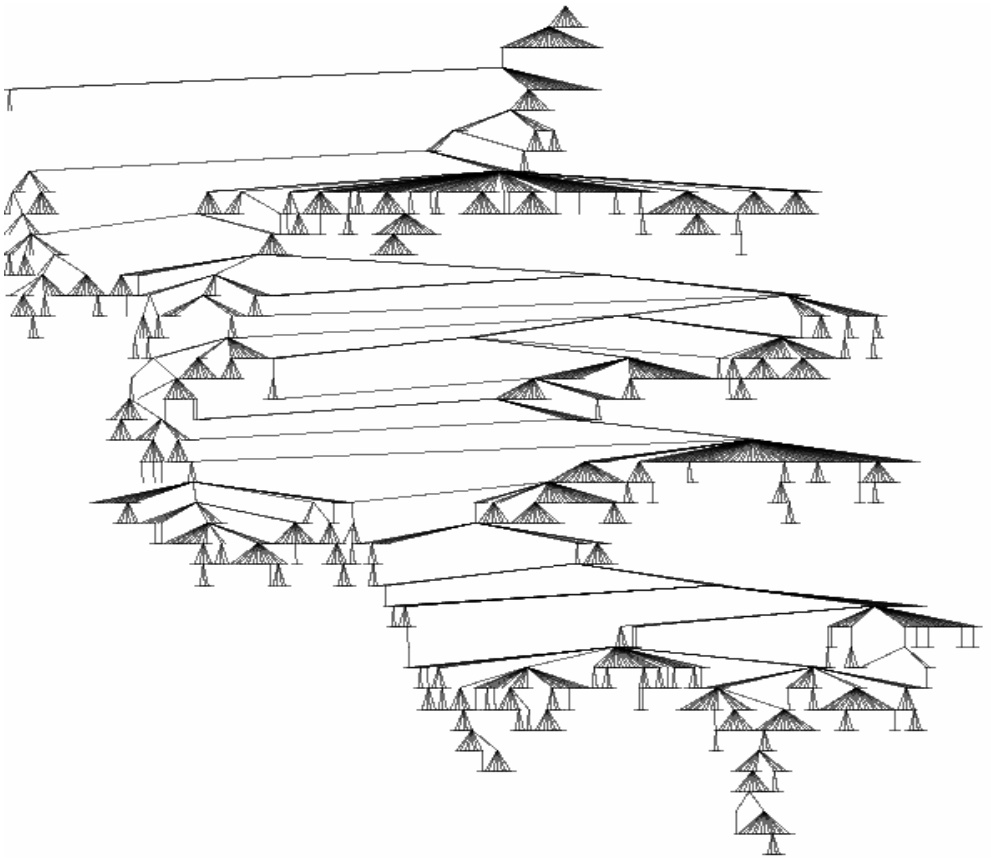


Figure 79. The dependency tree of class Button in package java.awt.

6.5.6. Conclusion and Further Research

Parallel studies between different disciplines frequently inspire new ideas. In recent years, people start to research the commonalities between software and biological

evolution (Svetinovic 2005), the similarity between a virtual society and the real society (Alberich 2002). The results are usually positive.

In this chapter, the topological structures of seven Java packages have been investigated. The interesting phenomenon is that all of them clearly show scale-free properties. This result leads to a good assumption that same laws are working behind the evolution of CDNs as well as other complex networks such as social networks and biological networks. Continuous studies may reveal more commonalities among the structure and evolution of those different complex systems and therefore provide new approaches for the understanding of the evolution of large software systems.

Another interesting research result of this chapter is the discovery of the relationship between scale-free networks and the optimized sorting algorithms. This result may raise a conjecture that under certain circumstance, a scale-free network can be an optimized form as the architecture for large systems. If so, it is not so surprising that we have found so many large complex networks are scale-free. Some people may believe that for a large system, after a long time evolution, it tends to reach a stable and somehow optimized status. Even though the argument is not yet strong, the research into the relationship between scale-free networks and the optimized structure for complex systems is promising.

Besides the theoretical research into the scale-free networks and optimized software architectures, another research topic, which has more practical value, is to develop a software tool to monitor and manage the evolution of SCNs of large software systems. As we have seen in this chapter, the SCN of a large software system can be

very complex. If the tool is introduced when a SCN is still small and simple, and the tool can help the designer to keep the SCN in a fixed form such as layered form as the SCN evolves, we can expect to have a much simpler SCN even when the system becomes large. This feature will definitely make the system easier to maintain.

Chapter 7 Software Tools

In this research, three software tools have been developed to simulate the GSE process, collect data and prove conjunctures. The first tool is the "Genetic Software Engineering Toolkit" (GSET), which is used to simulate the GSE process and demonstrate the proposed traceability model and the software normalization. The second tool is called "Class Network" that is used to investigate the class dependency network of Java packages. The third tool is called "Sorting Comparison Network Explorer" (SCNE) and it is used to investigate the sorting comparison networks of different sorting algorithms.

7.1 GSET

7.1.1 Introduction

GSET (Genetic Software Engineering Toolkit) is an automation tool written in the Java language to implement the concepts of GSE. Most of the GSE diagrams shown in this thesis are drawn or validated using GSET.

GSET has the following features:

1. **Fast construction of RBTs**. Most times, the user only needs to click or drag the mouse, unless when he inputs the name of a new component, a new method, a tree or in rare cases, he does not need to touch the keyboard. Once a component, a method or a behavior tree is created, it can always be selected from a drop down list or from the listing window. This feature saves time and avoids typos; it may also help users by preventing from creating two interfaces with similar names that perform the same task. Compared to other GUI drawing tools, because GSET is specially designed for GSE, it is easier to generate GSE diagrams than any other tools.

2. **Automatically generate and update diagrams**. In GSET, except the RBTs that require the user to manually draw them, all the other diagrams are generated and updated automatically or semi-atomically by the tool. This feature is due to the traceable property provided by GSE and it is one of the most important features of GSET. It will dramatically save the user's time when drawing the GSE diagrams and also reduce human errors during the process of integrating behavior trees or projecting other diagrams from the DBT. Also, when a RBT is changed, the other diagrams can be updated automatically or semi-automatically; this feature simplifies maintenance.

3. **Flexible and user friendly display**. For a large project, the behavior trees especially the design behavior tree can be very large and it is very difficult to view the tree as a whole and also to check the details of the tree at the same time. GSET uses different approaches to deal with this problem.

    a. **Smooth zooming function** that can change the display size of the behavior tree. This is similar to the zooming function in other common computer drawing tools.

    b. **Multiple level information hiding**. This means when the user is building a behavior tree, each node is assigned a detail level, which ranges from 0 to 6. The smaller the number, the higher the level of this node. When the tree is shown, the user can select a certain detail level. All nodes with the level lower than the selected level will be hidden. A node with some lower level details hidden under it will be clearly marked,

so the user can select the node and ask GSET to pop up a separate window to show the hidden tree under this node (an example of this feature is presented in the next section). This feature enables the user to view the DBT in a very high level so the overall functionality of the system can be examined and at the same time to quickly check the details of the design of a particular function.

c. **Collapse a branch of a tree**. Apart from multiple level information hiding, GSET provides another way, collapsing a branch of a tree, to reduce the size of a tree. The user can select to collapse any branch of any tree (a RBT, a DBT or a CBT). If a node has a collapsed branch under it, it will be clearly marked and the user can expend it at any time.

d. **Component based display scheme**. Each node in a RBT or DBT is associated with a component. The user can select a display scheme (such as a certain color, shadowed or double line) for a certain component so each node in behavior trees associated with such component will inherit such display scheme. This feature will help the user to trace a certain component in a big behavior tree. Also, for individual nodes, the user can also set its own display scheme that can be different from the display scheme inherited from the component.

4. **Validation of the RBT.** GSET can help to find incompleteness of the behavior tree and so help users to find out the incompleteness in the requirement specification.

5. **Implement the traceability model**. GSET has implemented the traceability model. Therefore, if there are two different versions of DBT, it can compare their difference by creating EDBT and project out different type of edit design behavior tree automatically

6. **Export to PDF file.** All the diagrams shown in GSET can be exported to PDF formatted files. The user can select the size of the page and also can select to export a single diagram or the whole set in a project. When exporting the whole set, the user can also select which diagrams will be exported and which will not.

Generally, GSET is a user-friendly design tool for GSE and it covers most parts of

the original GSE specification. The screen shots of GSET can be found in appendix F.

7.1.2 Future Development Plan

1. **To integrate with Visio or other commercial drawing tools**. Even though GSET has an easy to use and user-friendly GUI, the functions are still limited compared to some powerful commercial drawing tools such as Visio and SmartDraw. Continual improvement of the GUI of GSET is a method to increase the usability of GSET. Another method is to integrate GSET with existing tools. For example, make GSET can export diagrams in some common diagram file format so that can be viewed or edited by other drawing tools.

2. **Concurrent multiple user collaboration system**. GSE is supposed to handle the design of large systems. Large systems usually have many designers working together. Further improvement may change the GSET into client-server architecture. The server side will handle the user authentication, data storage, privilege management, version control and change synchronization while the client side should only focus on the user interface.

3. **Simulation code generation**. Through the process of GSE, CBTs and CIDs can be retrieved and they are not far from the implementation if a suitable platform is set up properly. An ambitious idea is that the automation tool not only creates the design diagrams but also creates the source code or at least a good framework in which further implementation code can be added easily.

4. **CSP, EBNF and XML schema support**. CSP (Communicating Sequential Processes) provides a mathematical approach to describe concurrent processes and the concept is suitable for component based software design and implementation (Hoare 1985). If a DBT can be translated into CSP, tools used to validate the CSP can be used to validate the corresponding

DBT as well. EBNF (Extended Backus-Naur Form) is a notation originally invented to describe important parts of the syntax of the Algol-60 programming language (Dromey 1989). It can also be used to define the syntax of other languages. XML (eXtensible Markup Language) is a type of markup languages and it attracts many interests in recent years (W3C). The main strength of XML is that it provides a very flexible and powerful ability to present a broad range of information. Another strength is because an XML file is a text file so that it is human readable, at least theoretically, and also easy for a computer to parse and process. These features make it suitable for EDI (Electronic Data Integration) and other types of data exchange between different systems. An XML schema is a kind of definition files that helps XML processors to validate and properly process an XML file. Supporting EBNF and XML schema will increase GSET's ability to cooperate with other systems.

The theory and notation of GSE is still under development. With the evolution of GSE itself, the GSET will also need to be updated to match the latest methodology of GSE to validate the concept of GSE and benefit the uses, which use GSE to design and/or maintain their software systems. Based on GSET, a new version of the BT approach environment Integrare has been developed. Integrare has more functions than GSE and the work of Integrare has been introduced in (Wen 2007a and Wen 2007c)

7.2 Class Network

Class Network is a tool used to explore the component dependency network of Java packages. This tool is also written in Java. The main part of it is a Java package called *classnet* and it utilizes another package *netp* which is a general Java package used to draw diagrams and it is also developed by the author. The tool includes the following major functions:

- **Parse and collect raw data**: After selecting a root directory of a Java project, it will scan and parse all the Java source code files under the directory and the

sub-directories. Then it will find out all the public classes and the dependency relationships between these classes.

- **Calculate statistic figures**: Once the source code files have been scanned and the raw information about the nodes and links of the dependency network has been collected, the tool will calculate some statistical figures such as number of nodes, number of links, average number of links on each node, standard deviation of link number, average distance between nodes, diameter of the network and clustering coefficient.

- **Visualize the dependency network**: The tool also provides facility to visualize the dependency network. It uses the force directed (Cruz and Tamassia, 1998) algorithm to arrange the layout of the dependency network so it is easier to recognize the clusters and topological structures.

- **Draw the histograms of the degree distribution:** One of the most important features of a dependency network is the power law degree distribution that suggests the scale-free feature of the network. The tool can show the histograms of the degree distribution of the number of input links, output links and total links; and it can also infer the parameters of the power law distribution as $P(k) \sim k^{-\gamma}$.

The screenshots and detailed functions are in appendix G.

7.3 Sorting Comparison Network Explorer

Sorting Comparison Network Explorer (SCNE) is a tool written in Java to explore the sorting comparison networks. The current version includes 5 different sorting algorithms (bubble sort, heap sort, quick sort, binary insertion sort and merge insertion sort). The tool can create a SCN, display the histogram of the degree distribution, and animate the evolution of the SCN. Details of this tool and some of the screenshots are presented in Appendix H.

Chapter 8. Conclusions and Future Work

In this thesis, we have used two approaches to study the nature of software changes and practical methods to manage the changes and the change impacts.

The first approach is the traditional traceability analysis approach and a new traceability model and its extension have been proposed. Through this traceability model, once the functional requirements of a software system are changed, designers can identify the change impact on the software system's architecture as well as other different design documents. The extension model can trace the change impacts caused by multiple times or changes, and it can be used to review the evolutionary history of a software system. This model is based on the behavior tree design approach (Dromey 2003), which implements behavior trees as a formal notation to describe functional requirements. For a targeted software system, once some requirement changes are input, a new version will be assigned to the system to identify the changes. During the lifetime of the software system, many versions can be created regarding to times of changes. The main ideas of the traceability model can be summarized in the following steps:

The first step is that for each version of the software system, it is described as a design behavior tree with the same version. The second step is to compare those different DBTs by a tree merge algorithm and to generate an evolutionary design behavior tree; this tree includes the information of system of all the versions. The third step is to project out different evolutionary design diagrams from the evolutionary design behavior tree. These evolutionary design documents not only host designs of different version, but they also visualize the evolutions. With the evolutionary information and the traceability information stored in the evolutionary documents, questions such as what the current design is, how it comes to be this and when it becomes this can be answered. Based on those answers, the design rationale questions (Bratthall 2000) of why the current design is like this might also be answered. One of the advantages of this traceability model and its extension is that most of the procedures can be implemented by automatic software tools (Wen 2007a, 2007c).

The second approach is different. It studies the common laws of the system architectures of large software systems regardless of the differences in their associated functional requirements. For a long time, people believed that the software architecture is determined by the system's non-functional requirements (Bass 1998), in other words, the software architecture may be independent of the functional requirements. However, there is no mathematical proof of this conjecture. This thesis has proved that, for a software system, the component architecture is independent to the software system's functional requirements. This proof is based on software systems that are designed using the BT design approach, but the principal is suitable for general software systems. Based on this result, the thesis conjectures that there could exist a universal optimal architectural structure

regardless the functional requirements of the associated software system; it also proposes that a tree structured architecture could be one form of the optimal architectural style due to some of the unique features of trees. Another aspect of this approach is related to scale-free networks. We have discovered that component dependency networks of many large software systems are scale-free networks (Wen 2007b), and we have also discovered that for sorting algorithms, the more optimized sorting algorithms tend to provide sorting comparison networks that are more like scale-free networks, while for not so optimized sorting algorithms, the sorting comparison networks are more like random networks. These results inspire us to guess that there is some connection with the scale-free networks and optimized software architecture. The following statements are a summary of the second approach. Even though some of them may not have been theoretically proved, the test results in the thesis support the conjectures.

1. Large software systems are built by an incremental and evolutionary process, or we can say that large systems are built through a series of changes.
2. The architecture of a software system can be independent of the functional requirements of the system.
3. The component dependency networks or the software architectures in the component level of large software systems are scale-free networks.
4. The sorting comparison networks of highly efficient sorting algorithms are scale-free networks.
5. The study of sorting algorithms provides a new approach to exploring the evolutionary process of large systems.
6. The form of scale-free networks is an optimized topological form for the architecture of large systems.

7. Hierarchy is an optimized structure to manage large systems.
8. The architecture of a software system can be improved and simplified into a tree-formed hierarchical structure through a process called architecture normalization.
9. A software system with a normalized architecture is relatively easier to understand and maintain than systems that have not been normalized.

Similar to other research, before a problem has been solved, new problems may emerge along the way. During this research, new ideas were inspired all the time; some of them may become future research work such as why all complex networks are not scale-free and how to make a reusable component. Discussion of these ideas is presented in Appendix I.

Generally, the research has some positive results to manage the change impact on software systems, propose a new approach to study the evolutionary nature of the architectures of large software systems. These results will contribute to further studies of large and complex software systems.

Finally, before the end of this thesis, I would like to thank my supervisor Professor Geoff Dromey again for his consistent support and encouragement in my Ph.D study.

Appendix A The Five Sorting Algorithms

a. Bubble Sort

Bubble sort is one of those sorting algorithms that are most easily implemented by software program. However the conciseness in the software code does not provide a short execution time. To sort a random sequence with *n* records, the average comparison times is (Knuth 1997c):

$$C = C_{n+1}^2 - \frac{1}{n!}\sum_{0\le r<s\le n} s!r^{n-s} = O(n^2)$$

The fact that $O(n^2)$ times comparison makes bubble sort one of the slowest sorting algorithms.

Let's consider a sequence $r_1 r_2 \cdots r_n$. The procedure of bubble sort is to scan from the left most record in the sequence to the second right most record and compare each record with its next record. If the compared pair is in the right order, then moves to the next pair; otherwise swaps the two records and then move to the next pair. After one round of this operation, larger records tend to move to the right and smaller records tend to move to left. Repetitions of the process will eventually make the sequence sorted. After the first round of scan, the largest element will be moved to the right end, so in the second round of scan, we will scan one record less. Similarly, after each round of scan, the scan length will reduce one. If there is no exchange in one round of scan, the whole sequence is sorted and no further operation is needed.

b. Heapsort

Heapsort uses sorting by selection approach. Each time, the maximum record is selected and put at the right most position, and then the second maximum record is selected etc. To find out the maximum record in a sequence of n records requires at least $n-1$ times of comparisons (Knuth 1997c, p141), so without any optimization to the procedure to continuously select the maximum record from a sequence. The total number of comparisons is:

$$C_n = \sum_{i=1}^{n-1} i = \frac{n \times (n-1)}{2} = O(n^2)$$

However, after the first maximum record is selected, we have already collected some information about the order of the remaining records, so it may need less times of comparisons to discover the second maximum record. The same rule applies to the third maximum record and etc. Therefore the total number of comparisons can be much less.

To store and use the comparison results, heapsort uses an interesting data structure that is called "heap". A heap can be visualized as a complete binary tree where each node represents a record; the special part of the binary tree is for every node, if it has child nodes, the value of the parent node is larger than those of the child nodes. Obviously, in a heap, the root node holds the largest record. After the record in the root node is removed and put into the right most position, there is a vacancy at the root node. The candidates for the new root node can only be the records in the original root's two child nodes. The larger record in the two child nodes will be "promoted" to the new root node, and it will create a new vacancy at the child node,

and then recodes under that new vacant node will be promoted etc. Finally, a new heap with one record less will be formed and we can remove the record from the new root node again. Repeat this procedure we can eventually sort the whole sequence.

The above paragraph describes the main concept of heapsort. The design of heapsort is very elegant; it does not require an auxiliary output area or extra storage to store the binary tree-like heap structure besides a few indexes. Heapsort includes two phases: the heap-creation phase and selection phase. In the first phase, the original sequence is transferred into a heap and in the second phase, the largest record in the heap is selected and put in the right position and the remaining data will be adjusted to become a heap again.

A heap stored in a sequence of records can be mapped into a complete binary tree. For example, a heap of 16 records $r_1 r_2 \cdots r_{16}$ is mapped into a complete tree shown in Figure 80.

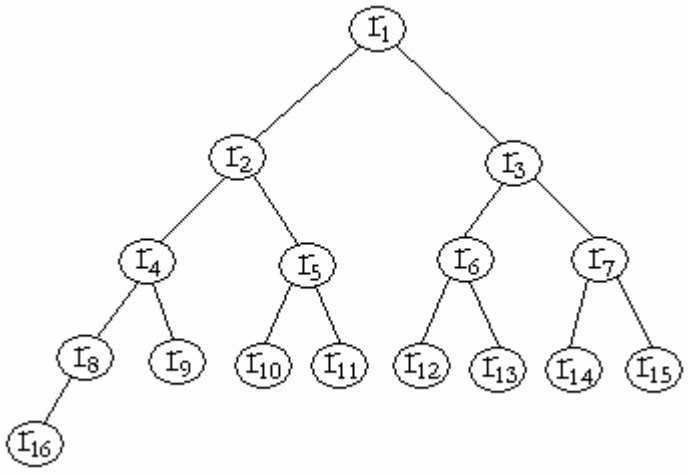


Figure 80 A heap of 16 records mapped into a complete binary tree

As discussed before, a heap, when it is visualized as a binary tree, the record in each

node is larger than the records in its child nodes. Therefore, for the given sequence $r_1 r_2 \cdots r_{16}$, if it is a heap, it must satisfy $r_{\lfloor j/2 \rfloor} > r_j, 1 \le \lfloor j/2 \rfloor \le j \le 16$. Actually, this is used as a definition of heap.

Definition: A sequence of records $r_1 r_2 \cdots r_n$ is a heap if:

$$r_{\lfloor j/2 \rfloor} > r_j, 1 \le \lfloor j/2 \rfloor \le j \le n$$

The first phase of heapsort is to convert the given sequence into a heap. To archive this goal, the algorithm to convert sequence $r_1 r_2 \cdots r_n$ includes following steps:

1. Let $k = \lfloor n/2 \rfloor$.
2. If $2k \le n$ and $r_k < r_{2k}$, swap r_k and r_{2k}.[23]
3. If $2k+1 \le n$ and $r_k < r_{2k+1}$, swap r_k and r_{2k+1}.
4. If $k = 1$, finished; otherwise, let $k = k-1$ and go to step 2.

After the original sequence is converted into a heap, it goes to the second selection phase that uses a so called *siftup* algorithm. The siftup algorithm includes following steps:

1. Let: $l = n$, l is the length of heap.
2. Because sequence $r_1 r_2 \cdots r_l$ is a heap of l records, $r_1 > r_i$, $1 < i \le l$. Let $R = r_l$ and move r_1 to r_l.

---

[23] Step 2 and step 3 are recursive steps. That means if we have swapped r_k and r_{2k} or r_{2k+1}, we have to consider the child nodes of r_{2k} or r_{2k+1}, eventually we may need to update the whole sub-tree.

3. Let $k = 1$.

4. If $2k \geq l$, let $r_k = R$ and go to step 7.

5. If $2k + 1 \geq l$, let $r_k = \max(R, r_{2k})$ and $r_{2k} = \min(R, r_{2k})$; then go to step 7.

6. Compare r_{2k} and r_{2k+1}. If $r_{2k} > r_{2k+1}$, let $r_k = r_{2k}$, $k = 2k$, and go to step 4. Otherwise let $r_k = r_{2k+1}$, $k = 2k + 1$, and go to step 4.

7. Let $l = l - 1$. If $l > 1$, go to step 2. Otherwise stop.

The algorithm of heapsort is elegant and according to Knuth (Knuth 1997c, p145), this algorithm merits careful study.

c. Quicksort

Quicksort is another sorting algorithm that uses sorting by exchange stratagem, but it is far more efficient than bubble sort. In bubble sort, the same pair of records can be compared many times, but this problem does not exist in quicksort. Besides that, in quicksort, larger records will be compared with larger records and smaller records with smaller records. From the point of information theory, this stratagem will retrieve more information from each comparison (Cover, 1991). Generally, quicksort is a very efficient sorting.

The fundamental concept of quicksort is partition. A sequence of records, by applying a partition algorithm, will be separated into two sequences. Every record in the left sequence is smaller than any record in the right sequence. Therefore, if we can sort the left sequence and the right sequence individually, the whole sequence is sorted. To sort the two shorter sequences, the same partition algorithm is applied so the two sub-sequences are separated into even shorter sub-sequences. Performing

this algorithm recursively, the whole sequence is sorted[24]. Because quicksort uses a recursive process, a stack is necessary to keep the positions of the unsorted sub-sequences. According to Knuth, if the length of the original sequence is n, we need at more $\lfloor \lg n \rfloor$ entries to hold the stack.

To partition a sequence, the quicksort starts from two indexes. The lower index is 2 and the upper index is the length of sequence n. Compare the record at the lower index with the first record; if the record at the lower index is smaller than the first record, increase the lower index by one and continue to compare the record at the new lower index with the first record; otherwise, compare the first record with the record at the upper index. If the record at the upper index is larger than the first record, decrease the upper index and continue compare the first record with the record at the new upper index; otherwise swap the record at the lower index and upper index. The result of the process above is to make all the record before the lower index (excludes the first record) is smaller than the first record and all the record after the upper index is larger than the first record. After an exchange of the record at the lower index and upper index, the same process repeats from the new indexes. Finally, when the lower index meets the upper index, the first record will be inserted into the middle (at the point where the lower index meets the upper index), and the original sequence is partitioned into two sequences, the lower sequence and the upper sequence.

---

[24] When a sub-sequence is very short, for example shorter than a given threshold M, other sorting algorithms can be applied such as straight insertion sort. However, in this thesis, we reduce the parameter $M = 1$, so the whole sorting process is a pure quicksort.

d. Binary Insertion

Binary insertion is a by-insertion kind of sorting algorithms. For this kind of sorting algorithms, the records are separated into two sequences, the sorted sequence and the unsorted sequence. (For a random input sequence, we can take the first record as the sorted sequence and the rest of the records as the unsorted sequence). Then records from the unsorted sequence are selected one by one (usually sequentially) and inserted into the sorted sequence in the suitable position so the sorted sequence still keeps the sorted status. When the last record from the unsorted sequence is selected and inserted into the sorted sequence, the whole sequence is sorted.

To find the right position to insert a new record into the sorted sequence, binary insertion compares the new record with the record at the middle point of the sorted sequence. If the new record is larger than the record at the middle point, then compares the new record with the middle point record of the upper half sorted sequence otherwise compares the new record with the middle point record of the lower half sorted sequence. Using this method, each time, the search range will be reduced by half, so it requires about $\lceil \lg(l+1) \rceil$ times comparison to find the correct position to insert the new record (l is the length of the sorted sequence).

Theoretically, the total number of comparisons for binary insertion is very close to the theoretical low bound.

$$C_n = n\lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + 1$$

However, for a traditional array data structure, one single insertion operation may cause the movement of about half of the total records, so it is not very efficient for

large number of records for practical usage.

e. Merge Insertion

Merge insertion is a mixture of merging and insertion. The first step is to divide the original sequence into pairs or records and then compare the two records in each pairs. Then we will receive a group of larger records and a group of smaller records. For the group of larger records, they are sorted by using the same algorithm recursively and then we can have all the records arranged in a graph shown in Figure 81.

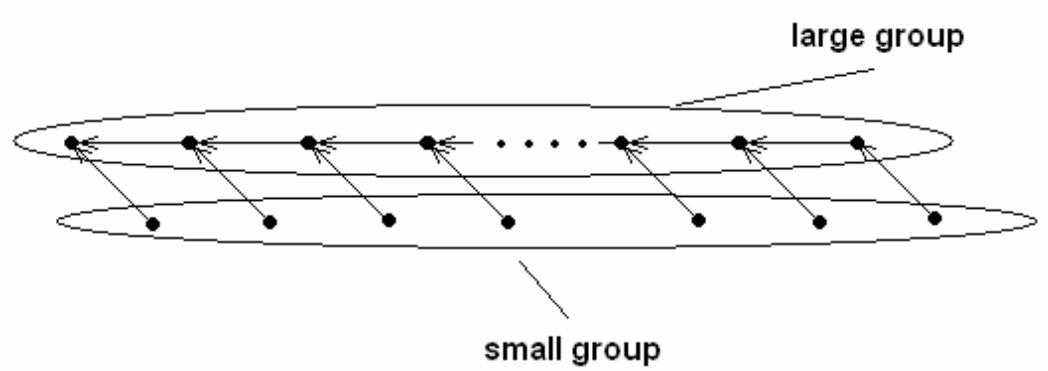


Figure 81. The illustration of merge insertion sorting algorithm

The large group is at the top and already sorted and the small group is at the bottom and partially sorted. We can see that the rightmost record in the small group is already in the right position. For the next rightmost record in the small group, we know it must be inserted into somewhere left of its matching record in the large group. To find the right position, the binary insertion algorithm is applied. Similarly, all the records in the small group will be sorted by using the binary insertion algorithm leftward one by one. After that, the whole sequence is sorted.

Appendix B The Study of the Random Sorting Algorithm

Theorem: *There is a sequence of* n *different numbers. To collect the sorting information, pairs are randomly selected from the sequence for comparison. In order to completely sort the sequence, the average number of comparisons is* $(n-1)(n^2-n+2)/2n$.

Proof: As we have discussed before, the sorting problem is equivalent to the problem of discovering the $n-1$ contiguous links in a n node network. Let p be the number of contiguous links and q be the number of non-contiguous links. We have:

$p = n-1$ and $q = n(n-1)/2-(n-1)$.

If we randomly connect i links ($i >= p$ and $i <= (p+q)$) and just after the last link is connected, all the contiguous links are discovered, we know the last link must be a contiguous link and the rest $p-1$ contiguous links must be among the previous $i-1$ links. In this situation, the combination of $p-1$ contiguous links among the $i-1$ links is C_{i-1}^{p-1}. Let A be the average number of links needed to find out all the contiguous links, we then have:

$$A = \frac{\sum_{i=p}^{p+q}(i \times C_{i-1}^{p-1})}{C_{p+q}^{p}} = \frac{\sum_{i=0}^{q}[(p+i) \times C_{p+i-1}^{p-1}]}{C_{p+q}^{p}} = \frac{p\sum_{i=0}^{q}(C_{p+i-1}^{p-1}) + \sum_{i=1}^{q} i \times (C_{p+i-1}^{p-1})}{C_{m+n}^{m}}$$

$$= p + \frac{\sum_{i=1}^{q} i \times (C_{p+i-1}^{p-1})}{C_{p+q}^{p}} = p + \frac{p\sum_{i=1}^{q} (C_{p+i-1}^{p})}{C_{p+q}^{p}} = p + \frac{p \times C_{p+q}^{p+1}}{C_{p+q}^{p}} = p + \frac{\frac{p \times (p+q)!}{(p+1)! \times (q-1)!}}{\frac{(p+q)!}{p! \times q!}}$$

$$= p + \frac{p \times q}{p+1} = \frac{(n-1)(n^2 - n + 2)}{2n} \quad \bullet$$

According to, by using the random network model, even under the best case situation, only $n-1$ links are required to solve the sorting problem, but the average number of links is $(n-1)(n^2-n+2)/2n$. This is close to the worst situation and much larger than the theoretical lower bound. This result clearly illustrates the inefficiency of the random network model.

Appendix C The SCNs of the 5 Sorting Algorithm

Figure 82 - Figure 86 show SCNs generated from the 5 sorting algorithms. The size of a node is determined by the number of links to the node[25]. All the layouts of the networks are arranged by using a force directed algorithm (Cruz 1998).

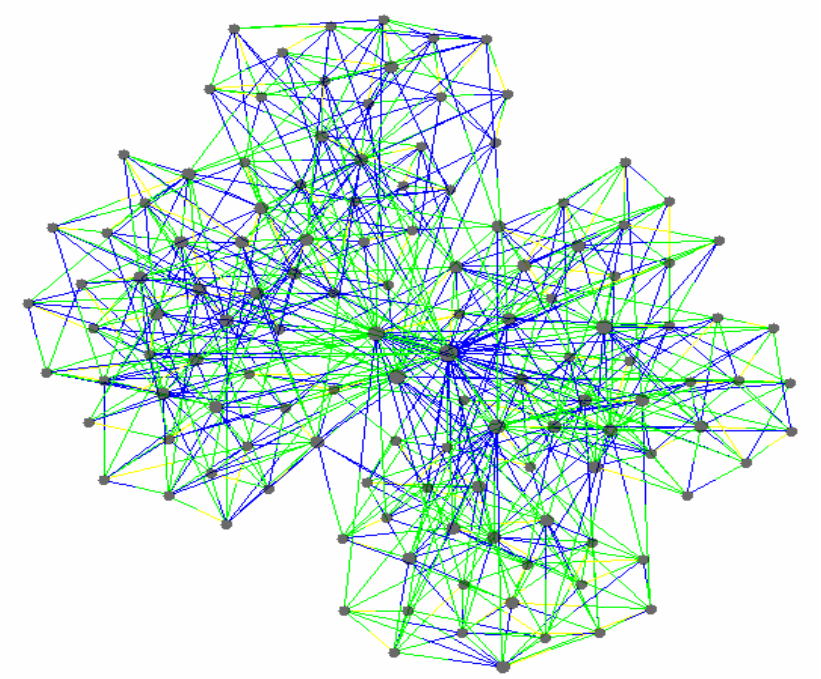


Figure 82. The SCN of binary insertion on 128 nodes.

[25] The original diagrams include colors. The colors on the lines indicate the comparison results. Blue part links to the node with smaller key values and the Green part links to the node with bigger keys; lines with yellow color means contiguous links.

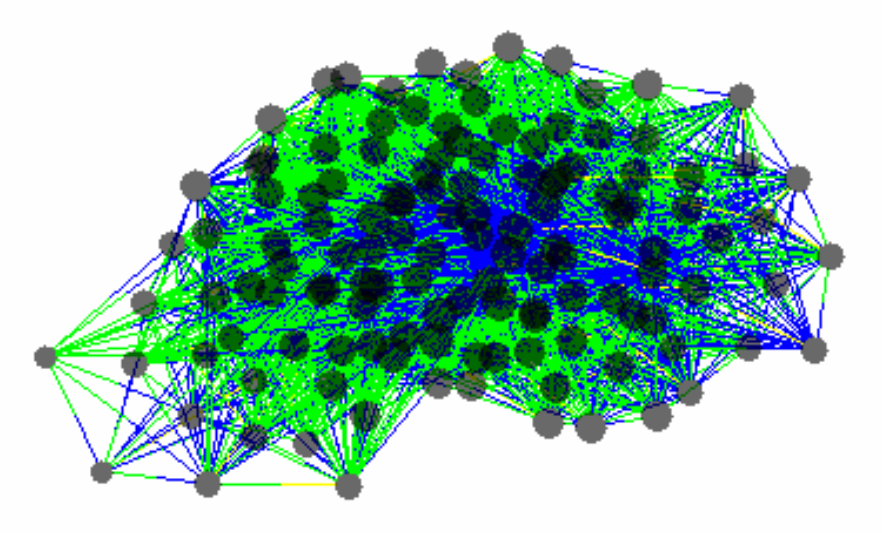


Figure 83. The SCN of bubble sort on 128 nodes.

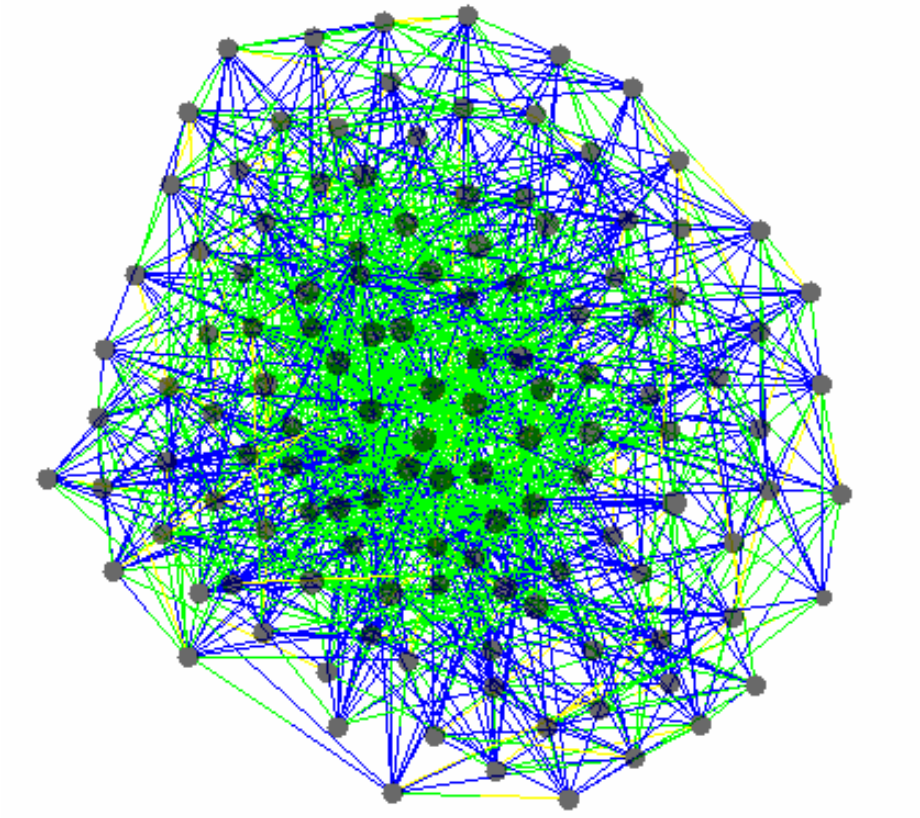


Figure 84. The SCN of heapsort on 128 nodes.

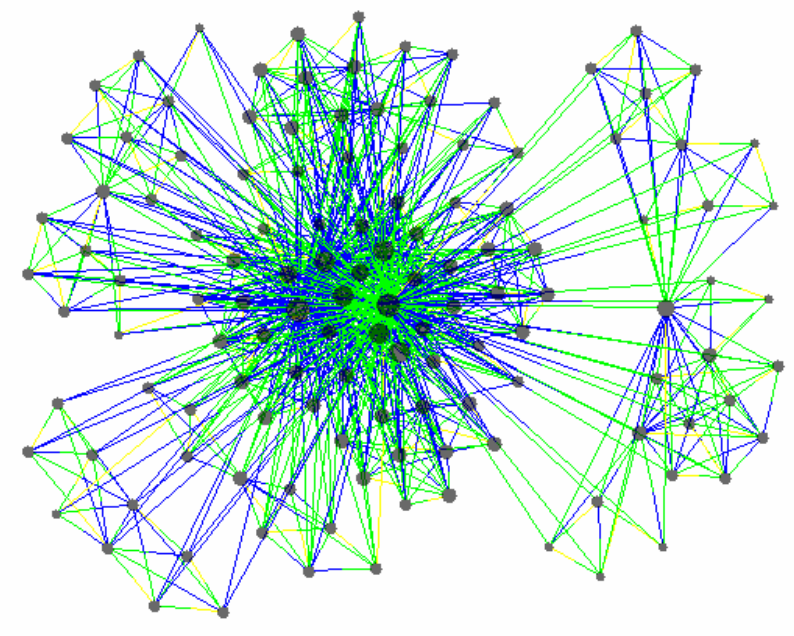


Figure 85. The SCN of quicksort on 128 nodes.

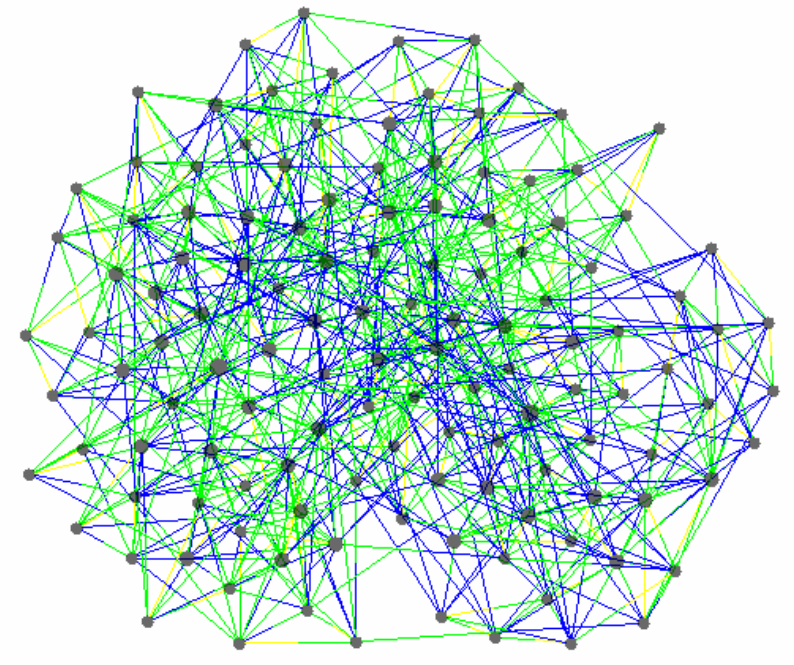


Figure 86. The SCN of merge insertation on 128 nodes.

From those SCNs, we can see that a SCN of each sorting algorithm has its own pattern. For example the bubblesort, because the number of connections is much

larger than other algorithms, the network is very tight and the size of the nodes is much larger than those in other SCNs. In the SCNs generated by binary insertion and quicksort, we can see clusters that are not clear in the SCNs generated by merge insertion sort and heapsort.

Appendix D The Distribution of The SCNs of the 5 Sorting Algorithm

For each sorting algorithm, two groups of tests are performed. The first group is on 1000 randomly generated independent sequence with the length of 256 in each and the second group is on 1000 randomly generated independent sequence with length of 1024. The degree distribution diagrams (Figure 87 -- Figure 96) are drawn based on the average degree of the SCNs from the 1000 independent random sequences.

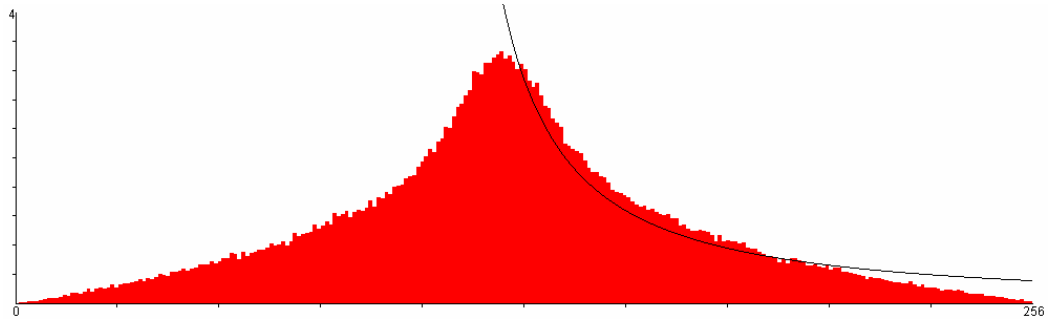


Figure 87. The average degree distribution of SCNs of bubble sort (sequence length 256, sample size 1000).

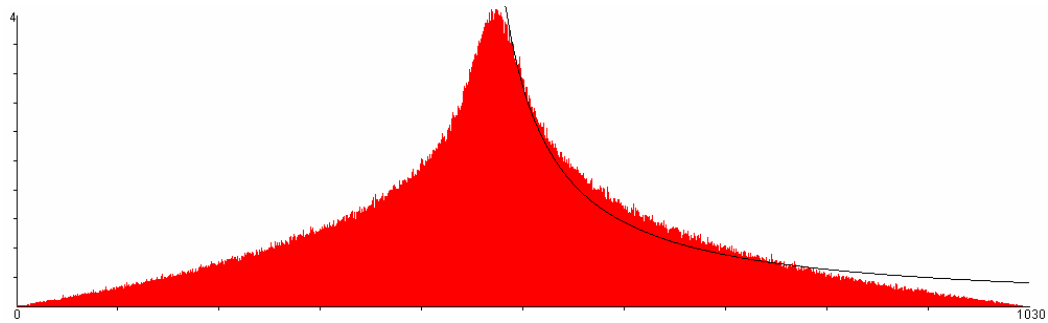


Figure 88 The average degree distribution of SCNs of bubble sort (sequence length 1024, sample size 1000).

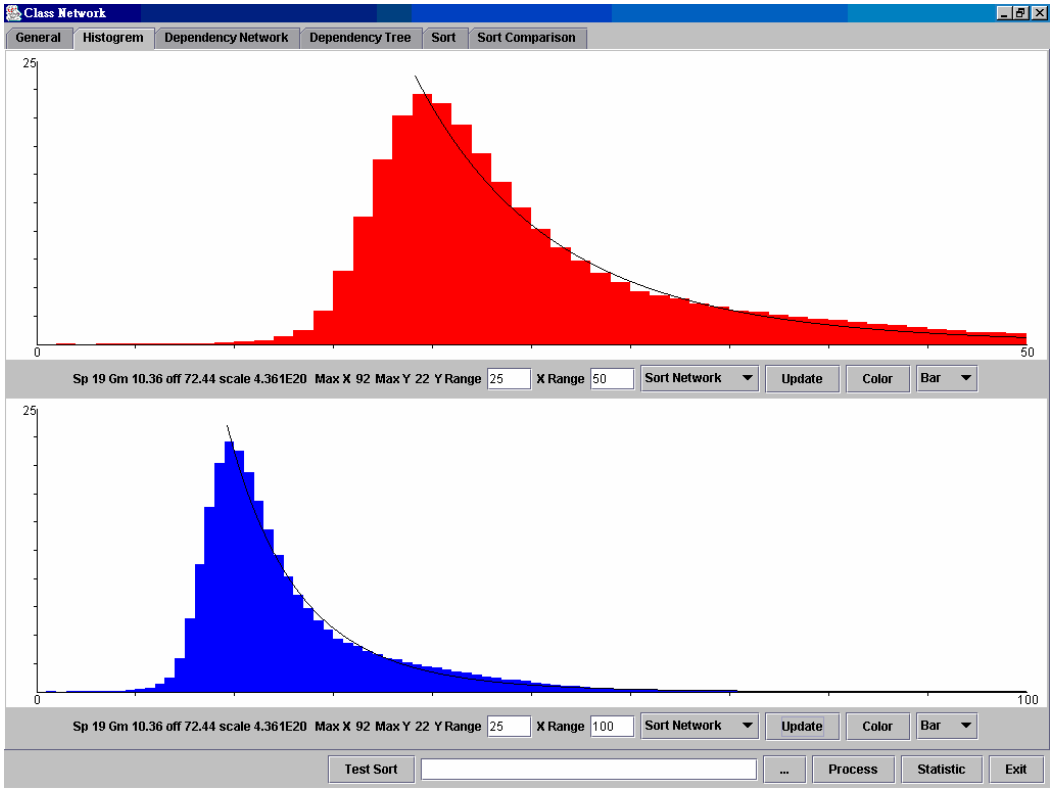


Figure 89. The average degree distribution of SCNs of heapsort (sequence length 256, sample size 1000; the top diagram shows the distribution in the lower range and the bottom diagram shows the distribution of the whole range)

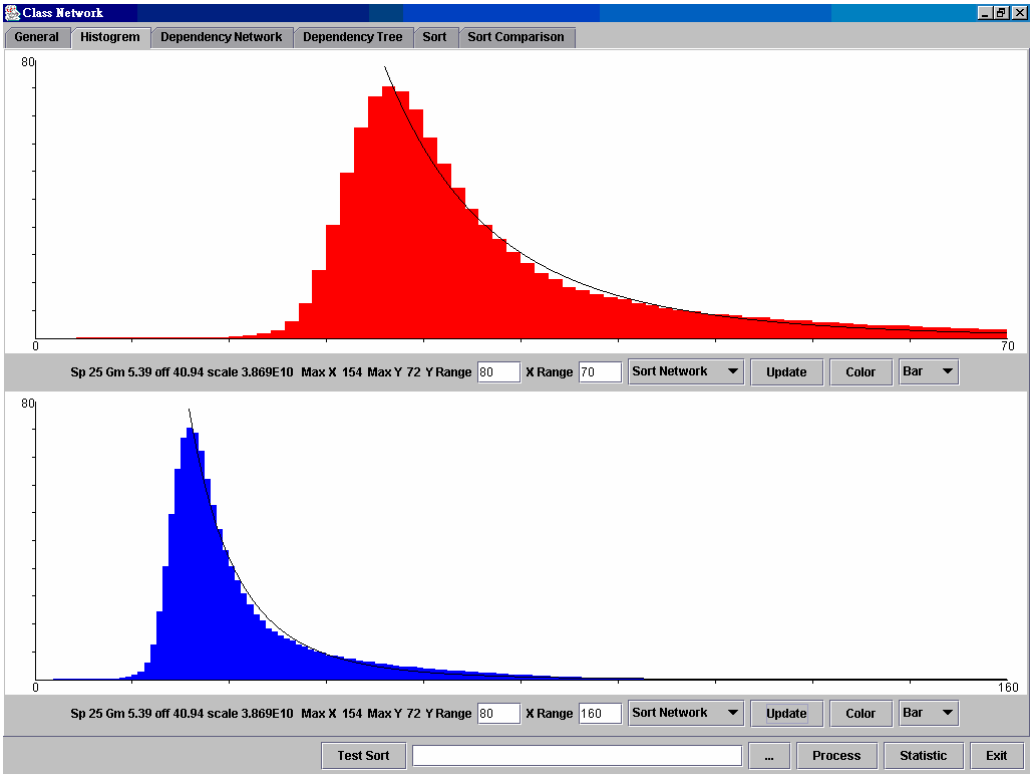


Figure 90 The average degree distribution of SCNs of heapsort (sequence length 1024, sample size 1000; the top diagram shows the distribution in the lower range and the bottom diagram shows the distribution of the whole range)

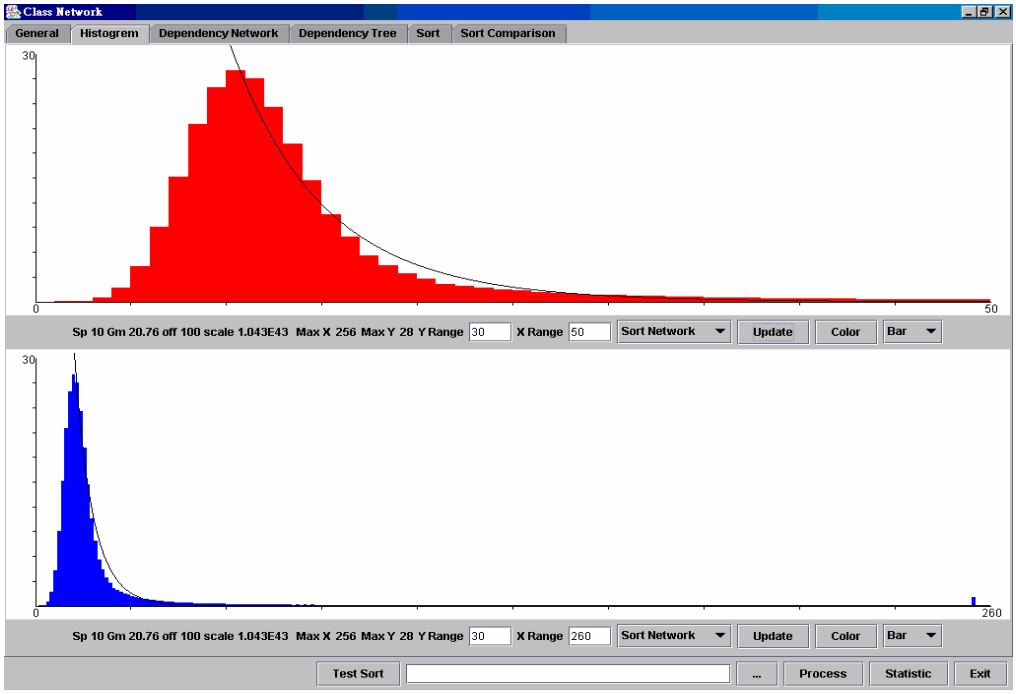


Figure 91 The average degree distribution of SCNs of quicksort (sequence length 256, sample size 1000; the top diagram shows the distribution in the lower range and the bottom diagram shows the

distribution of the whole range).

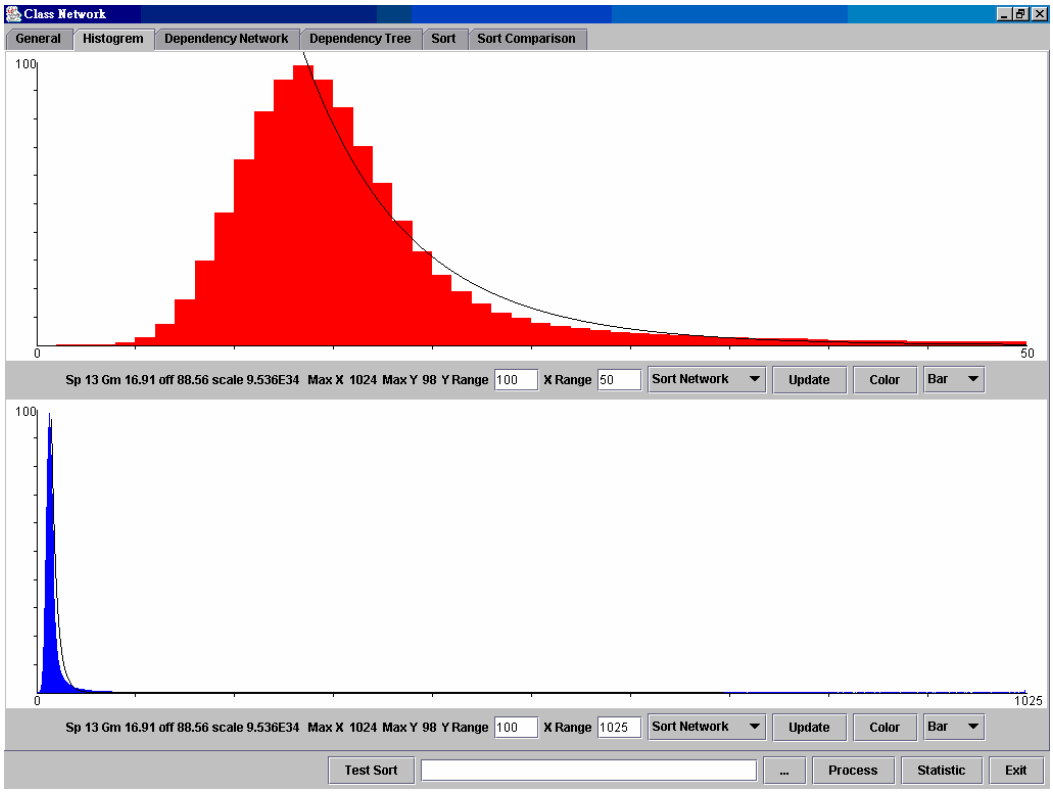


Figure 92. The average degree distribution of SCNs of quicksort (sequence length 1024, sample size 1000; the top diagram shows the distribution in the lower range and the bottom diagram shows the distribution of the whole range).

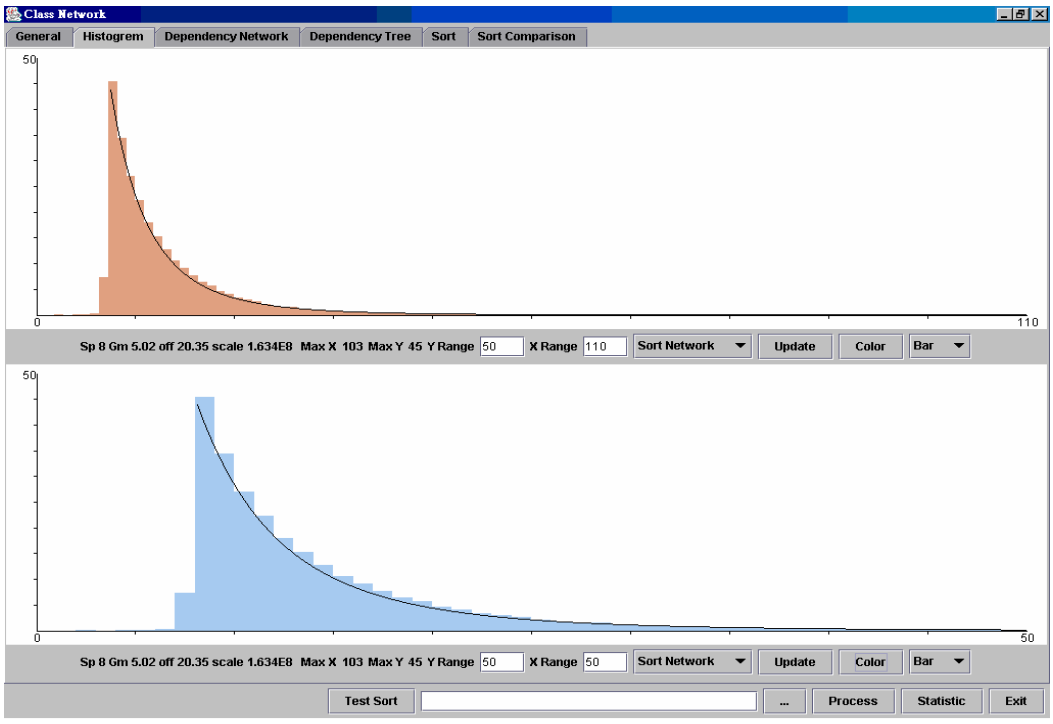


Figure 93 The average degree distribution of SCNs of binary insertion (sequence length 256, sample

size 1000; the bottom diagram shows the distribution in the lower range and the top diagram shows the distribution of the whole range)

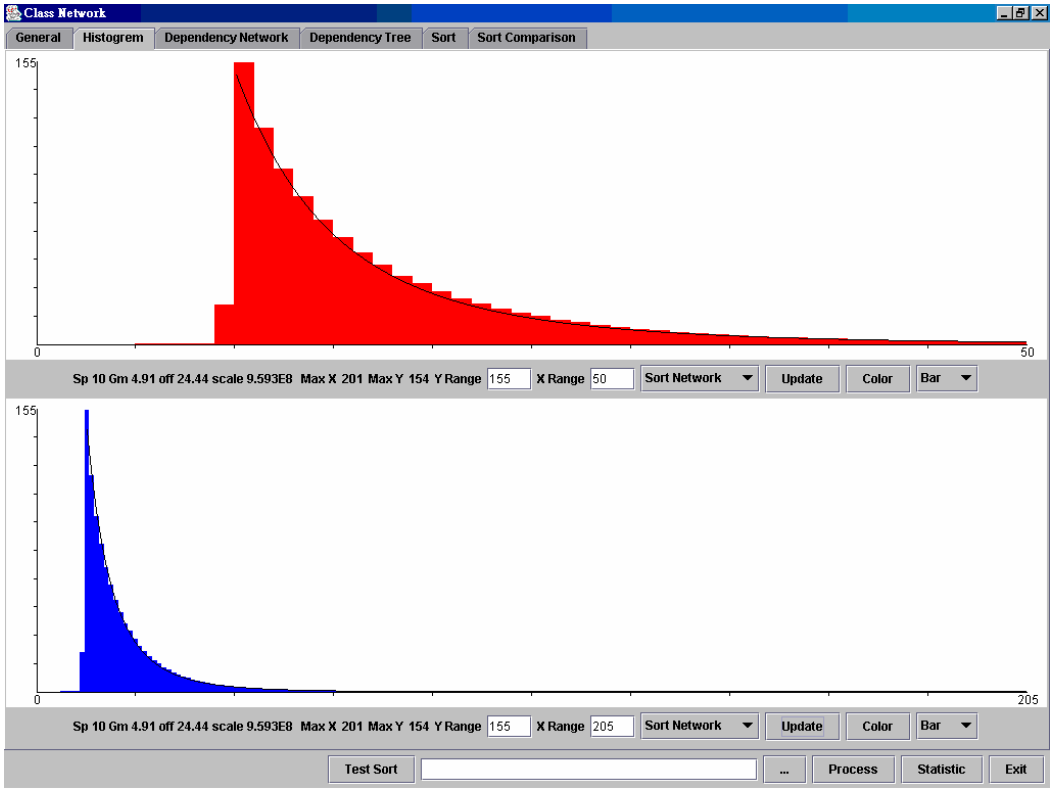


Figure 94 The average degree distribution of SCNs of binary insertion (sequence length 1024, sample size 1000; the top diagram shows the distribution in the lower range and the bottom diagram shows the distribution of the whole range)

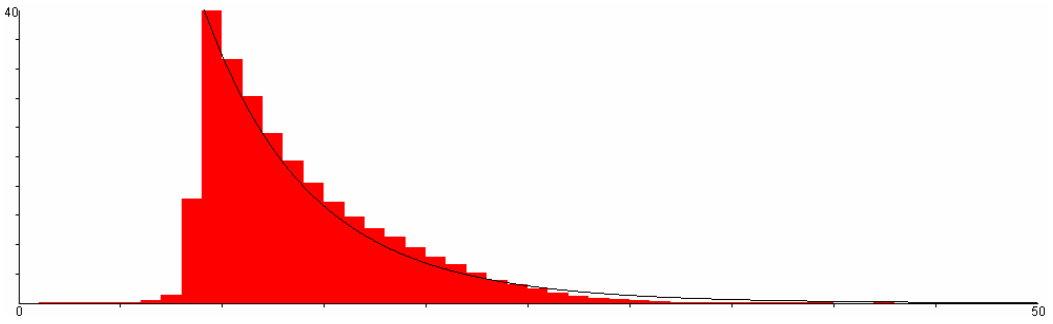


Figure 95 The average degree distribution of SCNs of merge insertion (sequence length 256, sample size 1000).

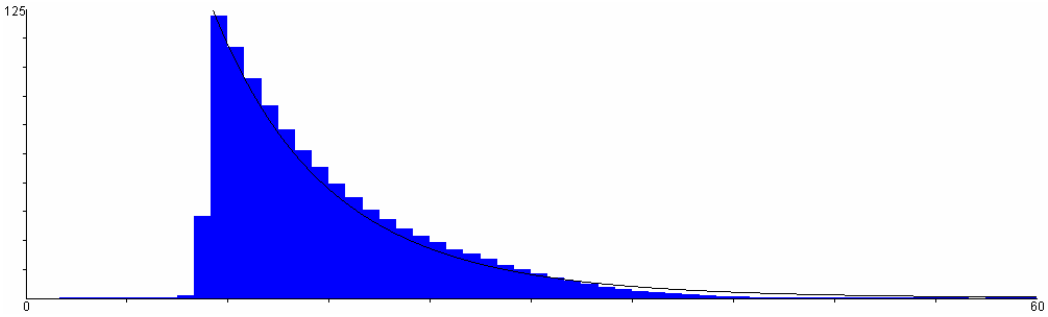


Figure 96 The average degree distribution of SCNs of merge insertion (sequence length 1024, sample size 1000).

From (Figure 87 -- Figure 96), we discover following features regarding to the SCNs of the 5 different sorting algorithms.

Definition: In a distribution diagram $y = f(x)$, x_0 is called **maximum point** if $\forall x \in Z,\ f(x) \le f(x_0)$ and $y_0 = f(x_0)$ is called **maximum value**.

1. All the degree distribution has a unique maximum point.
2. The degree distribution of bubble sort is symmetric from the middle point. The shape is like normal distribution but the tail is longer.
3. The degree distribution of quick sort has a bell shape like a normal distribution around the maximum point. However, the right part is higher with a long tail.
4. The degree distribution of heapsort is similar to that of quick sort. The difference is that the left part of the bell shape of heapsort is steeper and the right part of the bell shape is wider. However, the length of the tail is shorter.
5. The shapes of the degree distribution of binary insertion and merge insertion are similar. There is no bell shape in those diagrams. Only a small number of nodes are on the left of the maximum point. Both tails are following power

law distribution perfectly. The only significant difference between the distributions of binary insertion and merge insertion is that the binary insertion has a longer tail.

In Table 5, parameters of those distributions are listed, where len is the length of the sequence, x_0 is the maximum point, y_0 is the maximum value, l_t is the length of the tail. The other three parameters γ, x_{off} and A are used to draw the simulated power law curve as:

$$\bar{y} = \begin{cases} A \times (x + x_{off})^{-\gamma} & x_0 \le x \le l_t \\ 0 & otherwise \end{cases}$$

Sorting Method	len	x_0	y_0	l_t	γ	x_{off}	A
Bubble sort	256	122	3.2	255	1.3	20.6	2.1E2
Bubble sort	1024	488	4.0	1023	1.0	42.6	2.4E2
Heapsort	256	19	22	91	10.4	72.4	4.4E20
Heapsort	1024	25	72	153	5.4	40.9	3.9E11
Quicksort	256	10	28	255	20.8	100	1.0E43
Quicksort	1024	13	98	1023	16.9	88.6	9.5E34
Binary insertion	256	8	45	102	5.0	20.4	1.6E8
Binary insertion	1024	10	154	200	4.9	24.4	9.6E8
Merge insertion	256	9	39	42	16.4	84.6	1.9E33
Merge insertion	1024	11	122	59	11.8	80.6	4.5E24

Table 5. Parameters of the 5 sorting algorithms' degree distribution.

Supposing the original distribution is $y = f(x)$, then the power law curve is drawn as:

$$\bar{y} = \begin{cases} A \times (x + x_{off})^{-\gamma} & x_0 \le x \le l_t \\ 0 & otherwise \end{cases} \tag{12}$$

The distance between the distribution and the power law curve is defined as:

$$D = \sum_{x=x0}^{l_t} (y - \bar{y})^2 \tag{13}$$

The parameters γ, x_{off} and A are selected by the criteria to minimum the distance D

To calculate the parameters, we use the following method:

1. Set the initial searching range for $Min(\gamma) = 0$, $Max(\gamma) = 100$, $Min(x_{off}) = 0$ and $Max(x_{off}) = 100$. Set the testing steps $T = 20$ and the stopping distance $d = 10^{-6}$
2. Based on the searching range, determine the testing points. We have

$$\gamma(i) = Min(\gamma) + \frac{Max(\gamma) - Min(\gamma)}{T} \times i \quad 0 \le i \le T$$

$$x_{off}(j) = Min(x_{off}) + \frac{Max(x_{off}) - Min(x_{off})}{T} \times j \quad 0 \le j \le T$$

3. For each i, j the corresponding $A(i, j)$ is calculated as:

$$A(i, j) = \frac{\sum_{x_0 \le x \le l_t} y}{\sum_{x_0 \le x \le l_t} (x + x_{off}(j))^{-\gamma(i)}}$$

4. Applying equation (12) and (13), for each i, j , the distance $D(i, j) \quad 0 \le i \le T, 0 \le j \le T$ are calculated. Then compare those distances and identify the i_0, j_0 so the corresponding distance reaches the minimum.

5. If $\frac{Max(\gamma) - Min(\gamma)}{T} > d$ or $\frac{Max(x_{off}) - Min(x_{off})}{T} > d$, then define

$$Min(\gamma) = \gamma(i_0) - \frac{Max(\gamma) - Min(\gamma)}{T} \quad Max(\gamma) = \gamma(i_0) + \frac{Max(\gamma) - Min(\gamma)}{T}$$

$$Min(x_{off}) = x_{off}(i_0) - \frac{Max(x_{off}) - Min(x_{off})}{T}$$

$$Max(x_{off}) = \gamma(j_0) + \frac{Max(x_{off}) - Min(x_{off})}{T}$$

, and go to step 2, otherwise go to next step.

6. We have $\gamma = \gamma(i_0), x_{off} = x_{off}(j_0)$ and $A = A(i, j)$, and the searching is finished.

Appendix E The CDNs and the Degree Distribution of the Java Packages

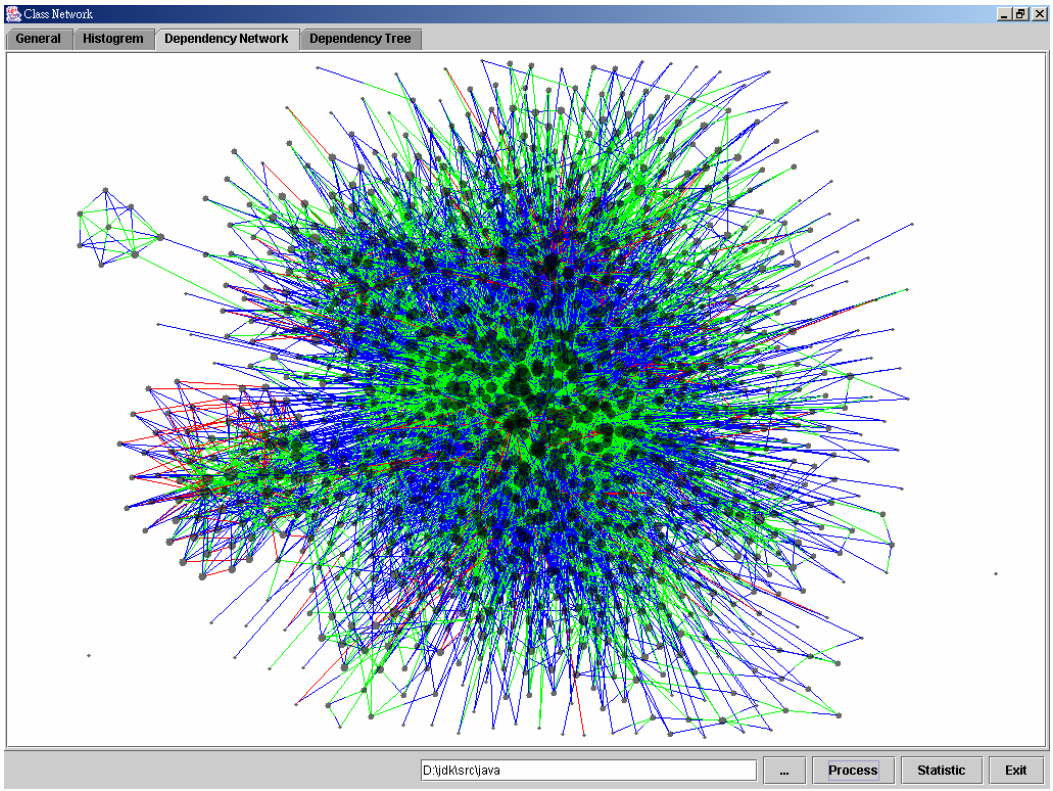


Figure 97. The dependency network of package *java*. This diagram is laid out by force-directed algorithm.

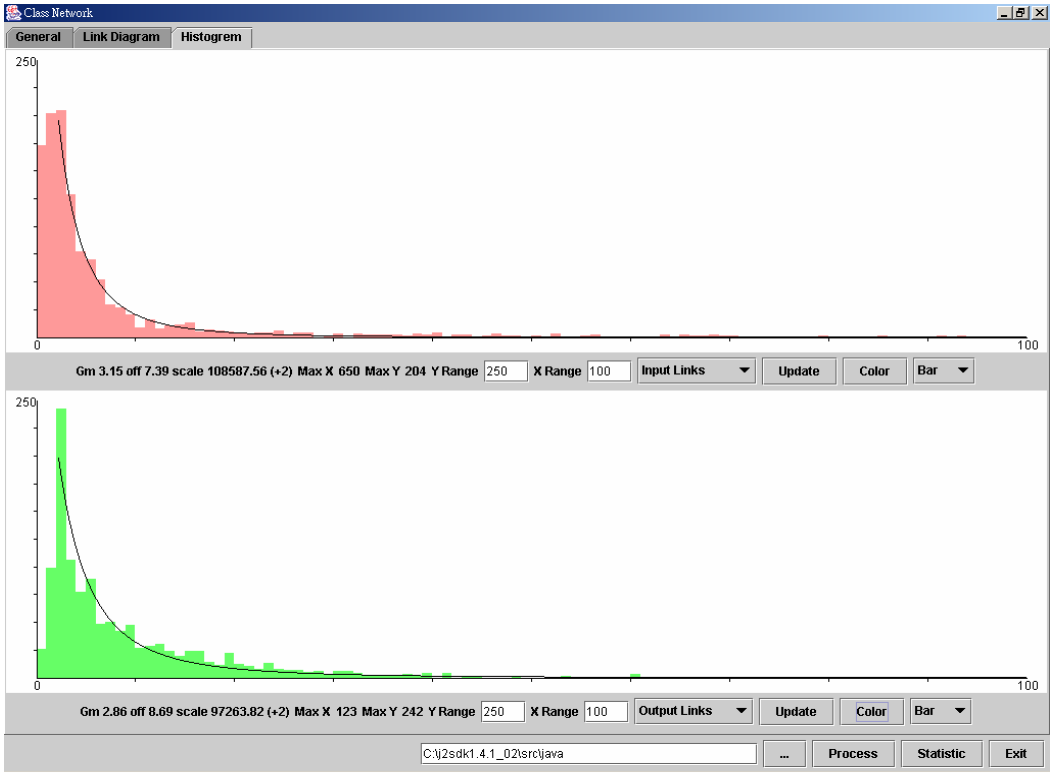


Figure 98. The distributions of input and output link numbers of the dependency network of package *java*

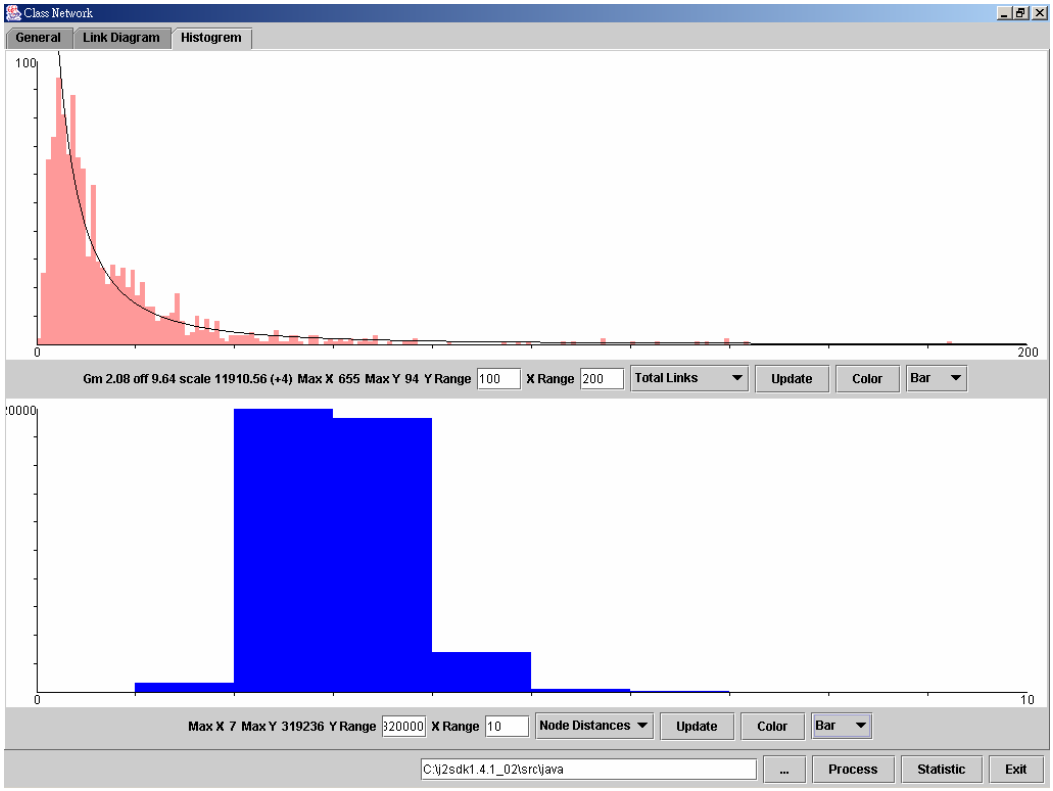


Figure 99. The distribution of the total number of links of the dependency network of package *java* and the distribution of the node distance.

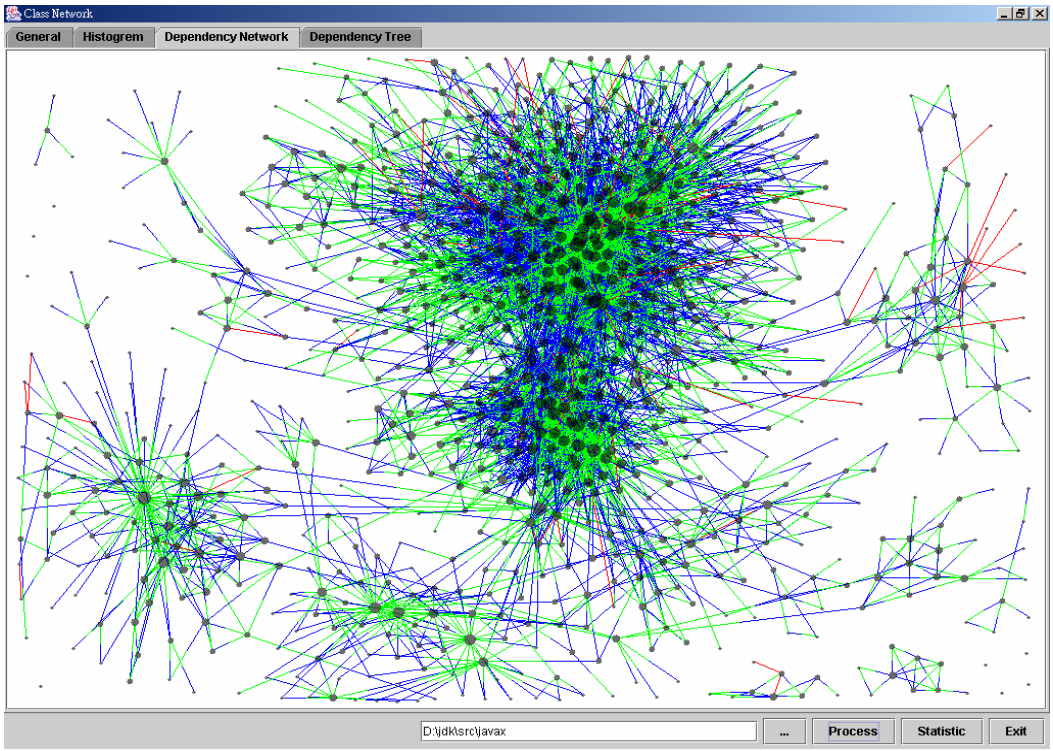


Figure 100. The dependency network of package *javax*.

Figure 101. The input and output link number distribution of package *javax*'s dependency network

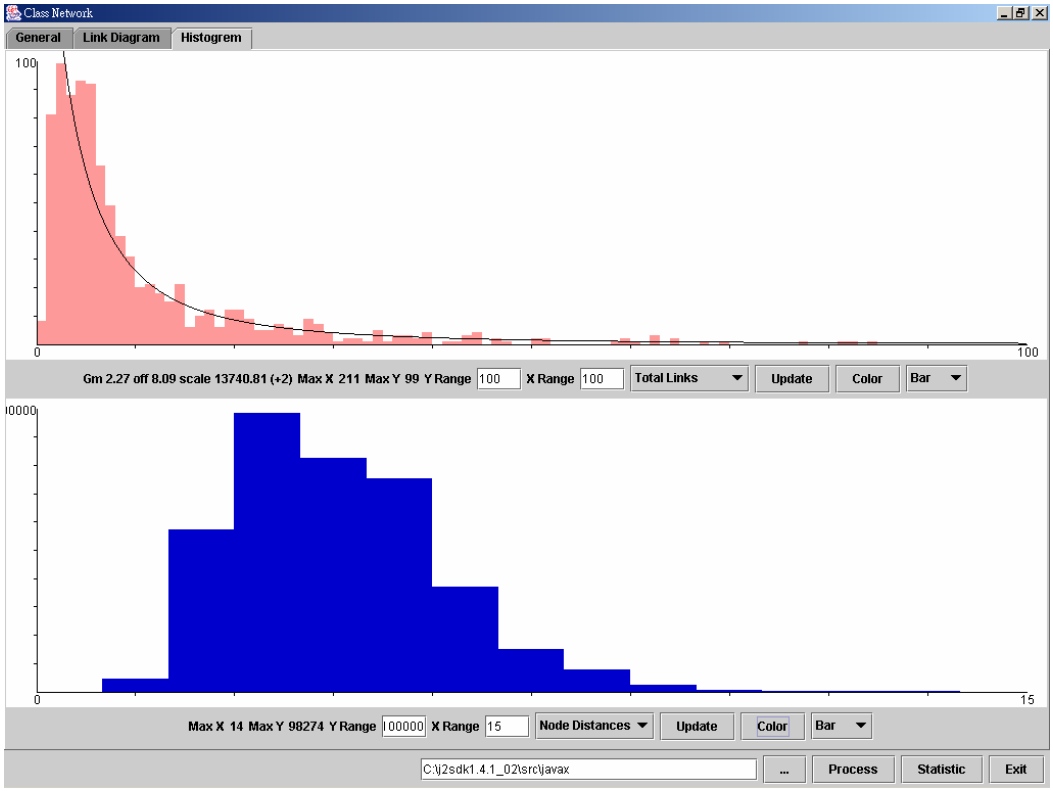


Figure 102. The degree distribution of the total link number and the distribution of the node distance of package *javax*.

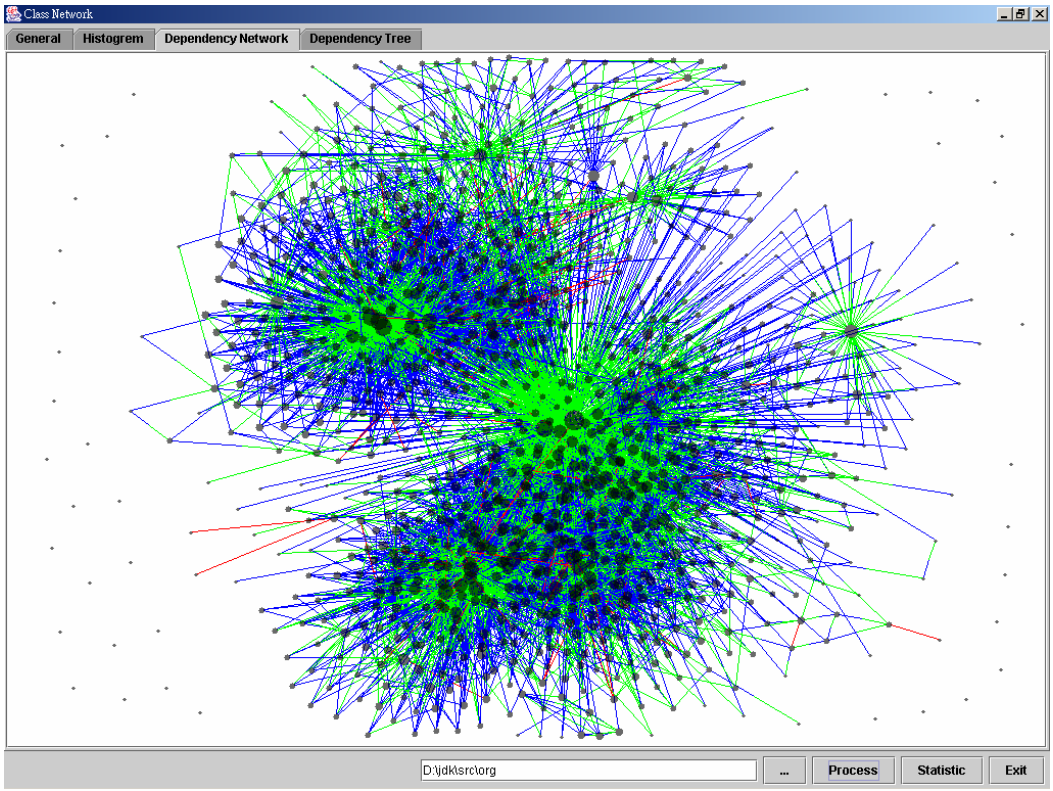


Figure 103. The dependency network of package *org*.

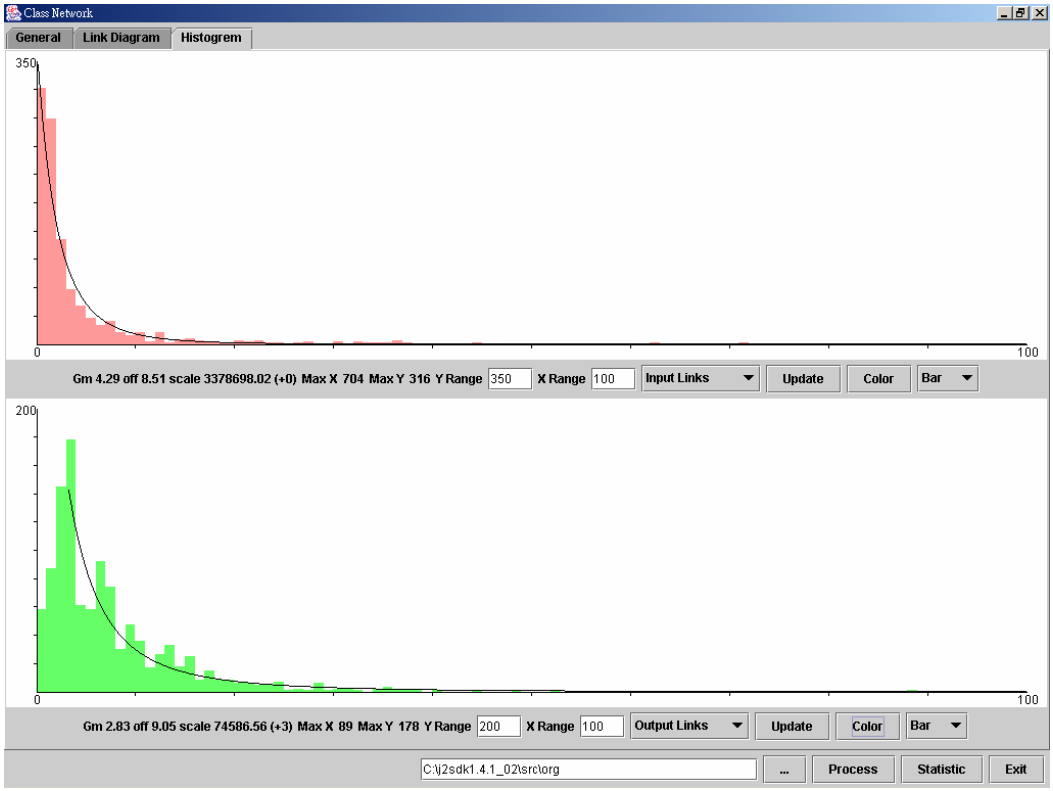


Figure 104. The degree distribution of the income link and outcome link of package *org*.

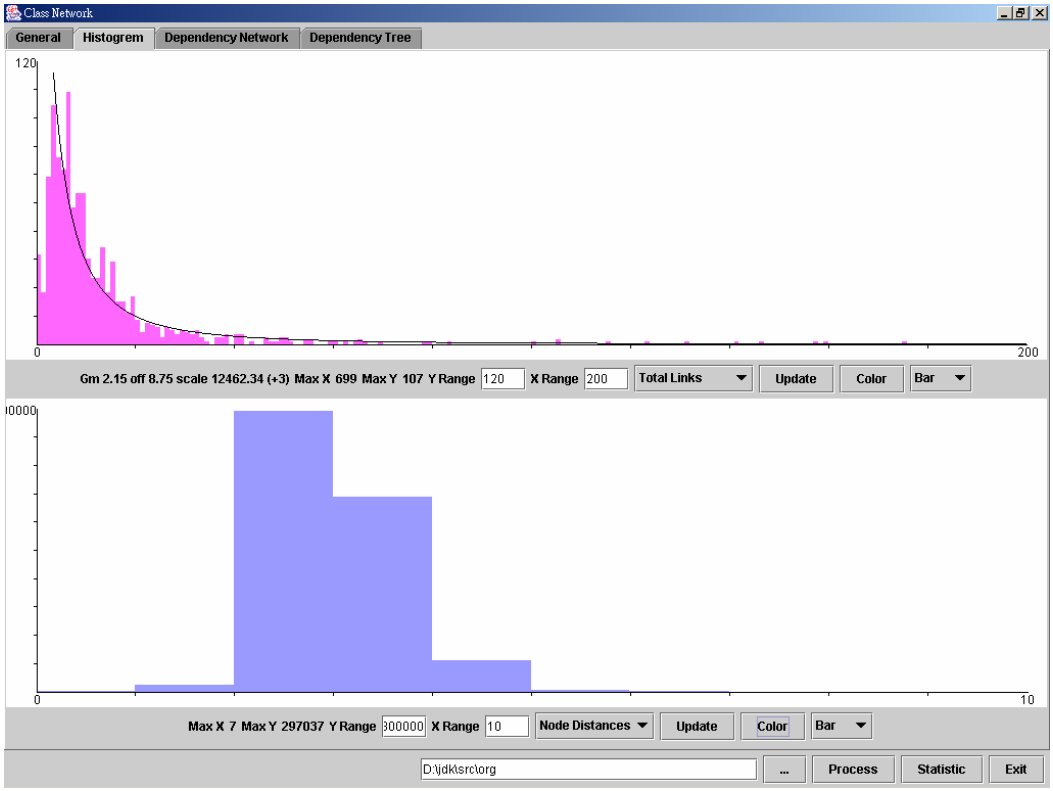


Figure 105. The degree distribution of the total number of links and the distribution of the node distance of package *org*.

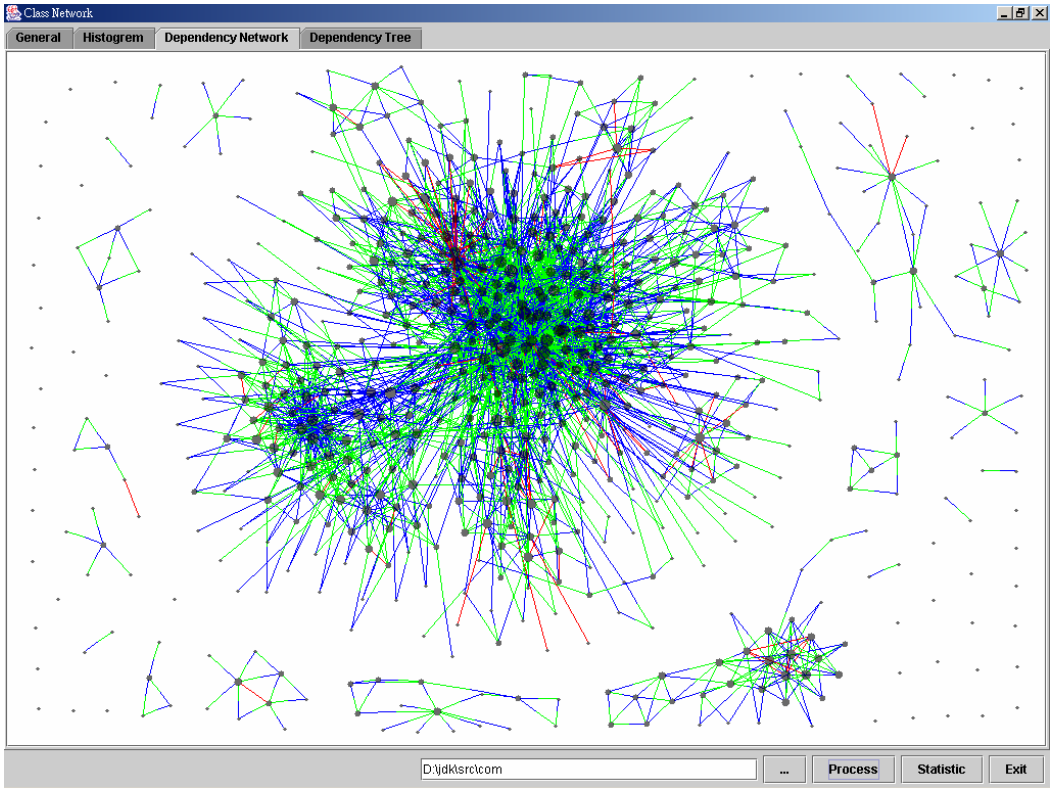


Figure 106. The dependency network of package *com*.

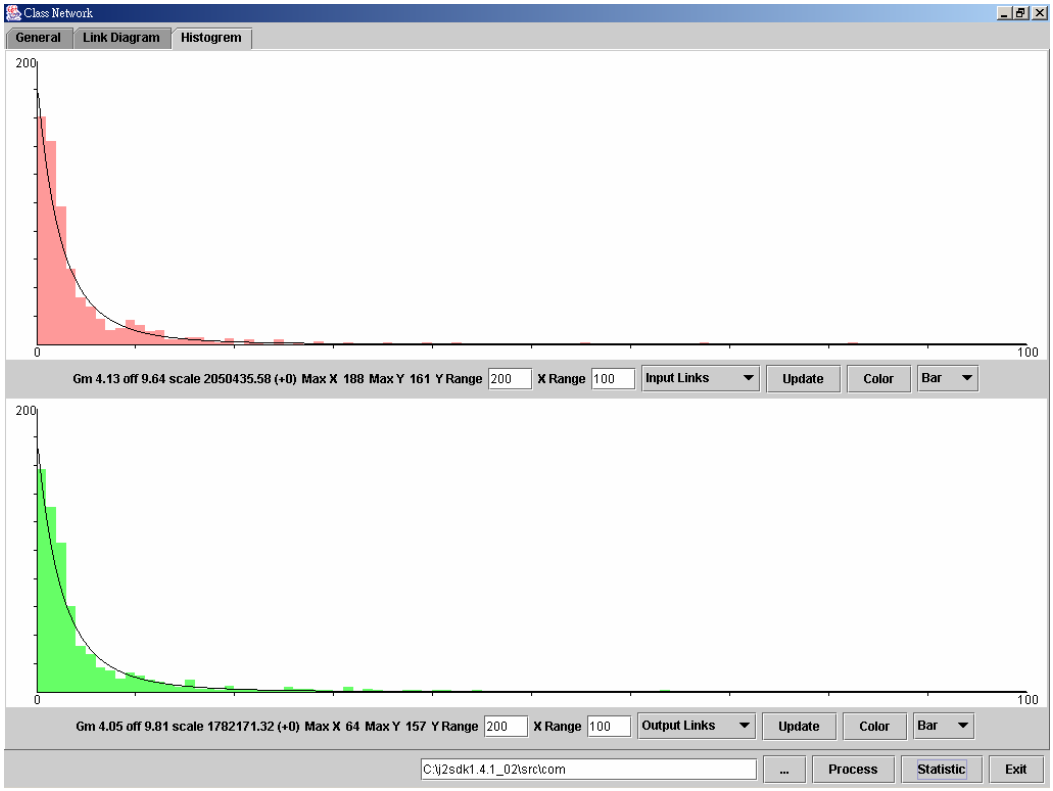


Figure 107. The degree distribution of the number of income links and outcome links of package *com*.

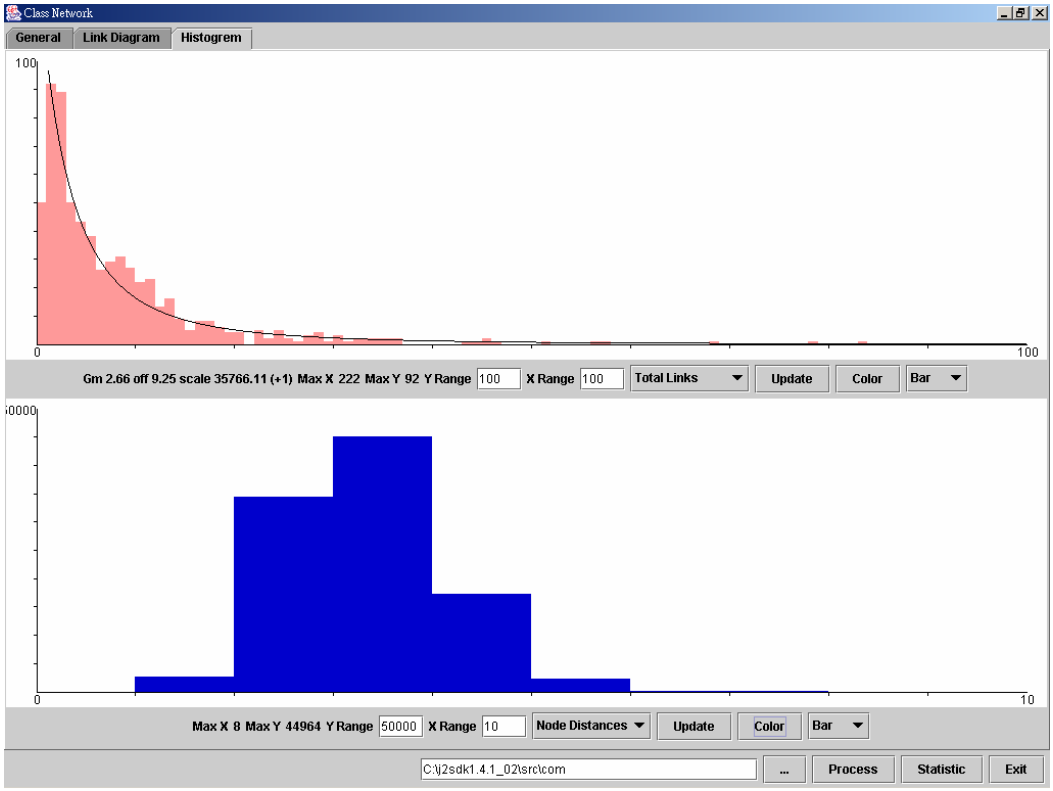


Figure 108. The degree distribution of the total number of links and the distribution of node distance of package *com*.

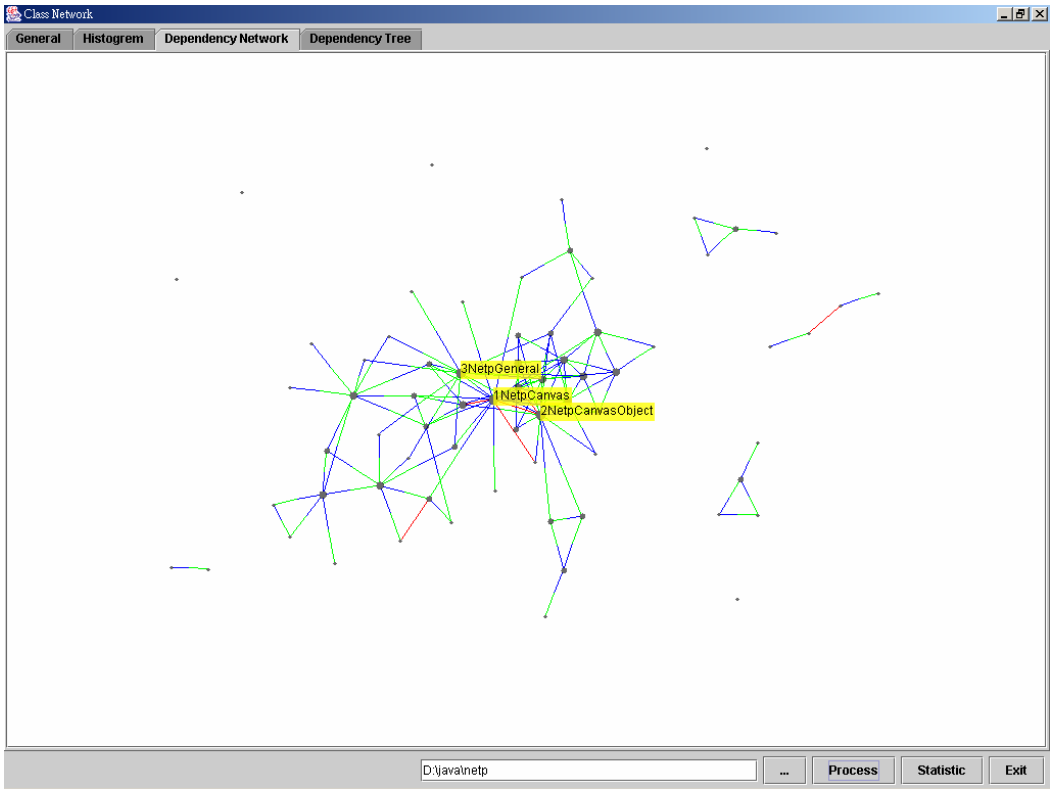


Figure 109. The dependency network of package *netp*.

Figure 110. The degree distributions of the input links and output links of package *netp*.

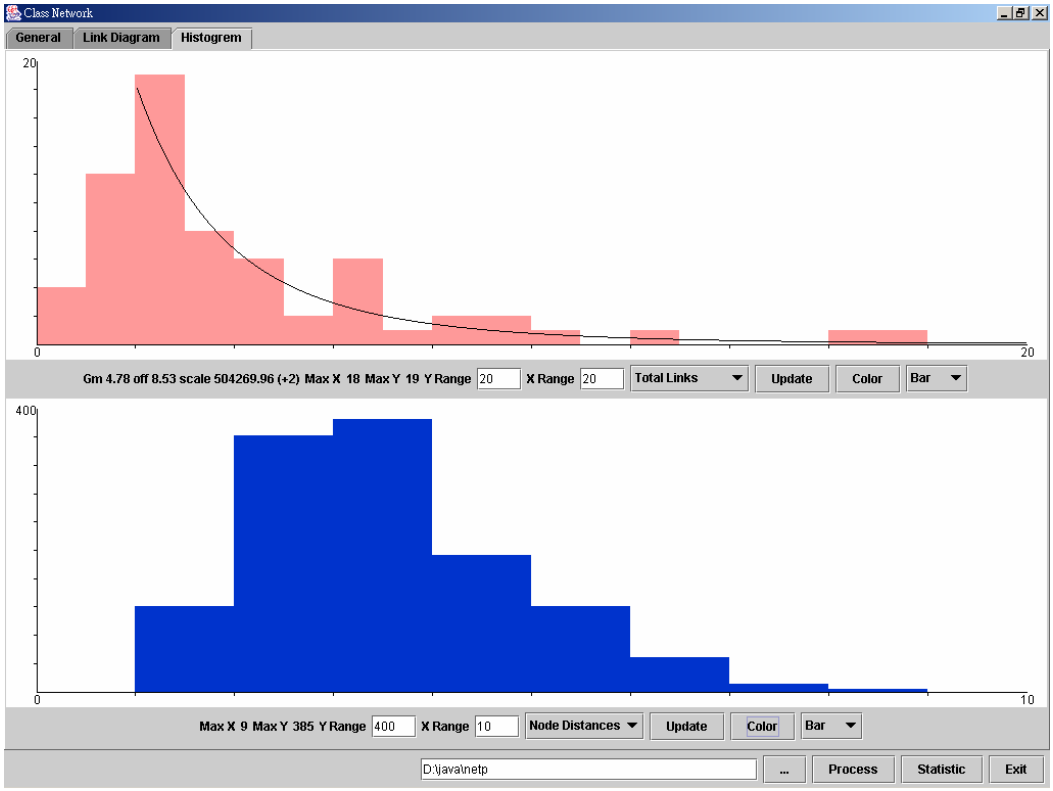


Figure 111. The degree distribution of total number of links and the node distance distribution of package *netp*.

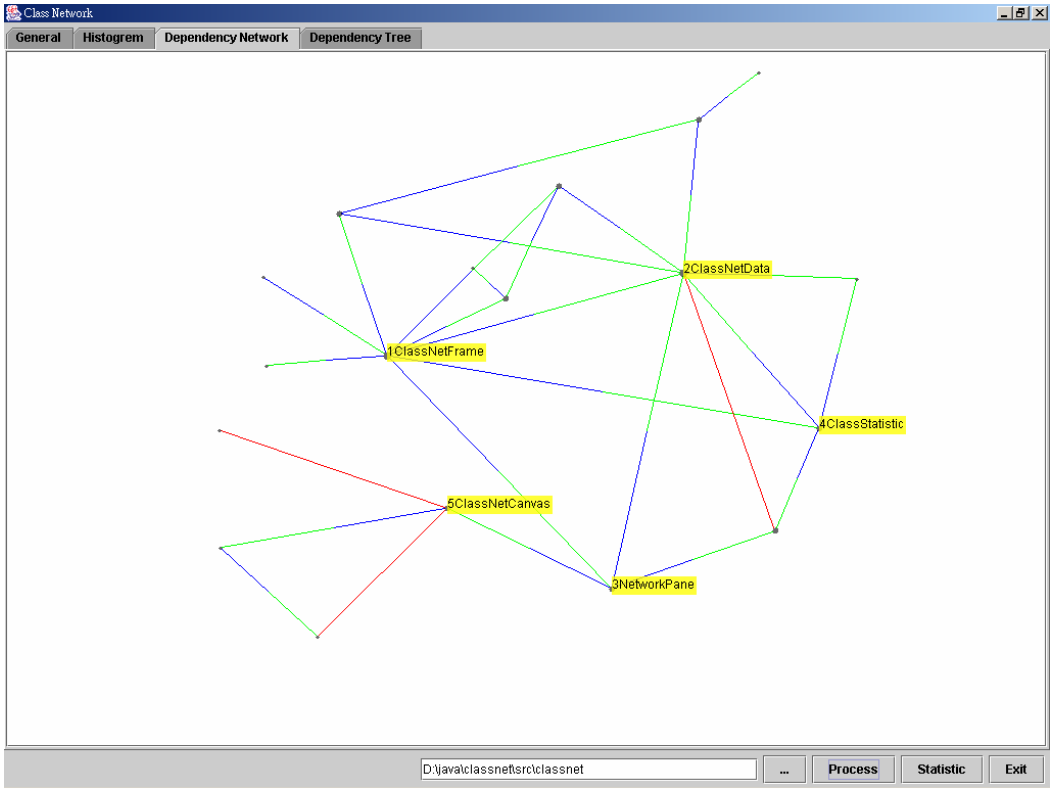


Figure 112. The dependency network of package *classnet*.

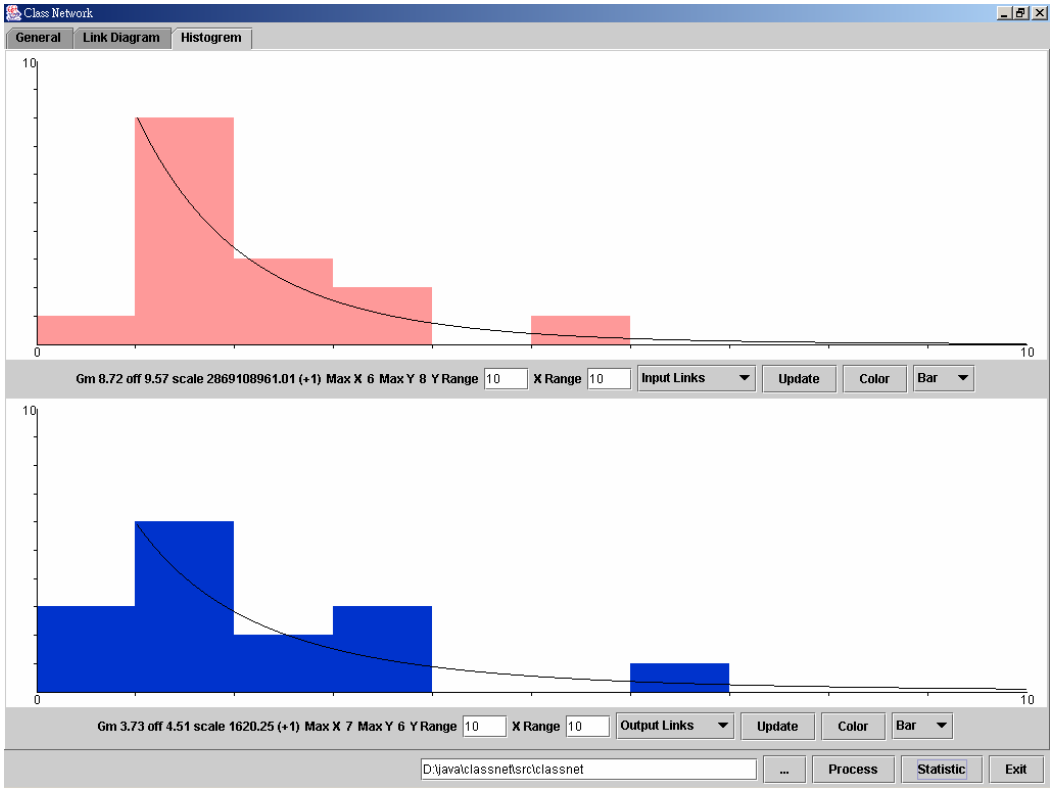


Figure 113. The degree distributions of the input links and output links of package *netp*.

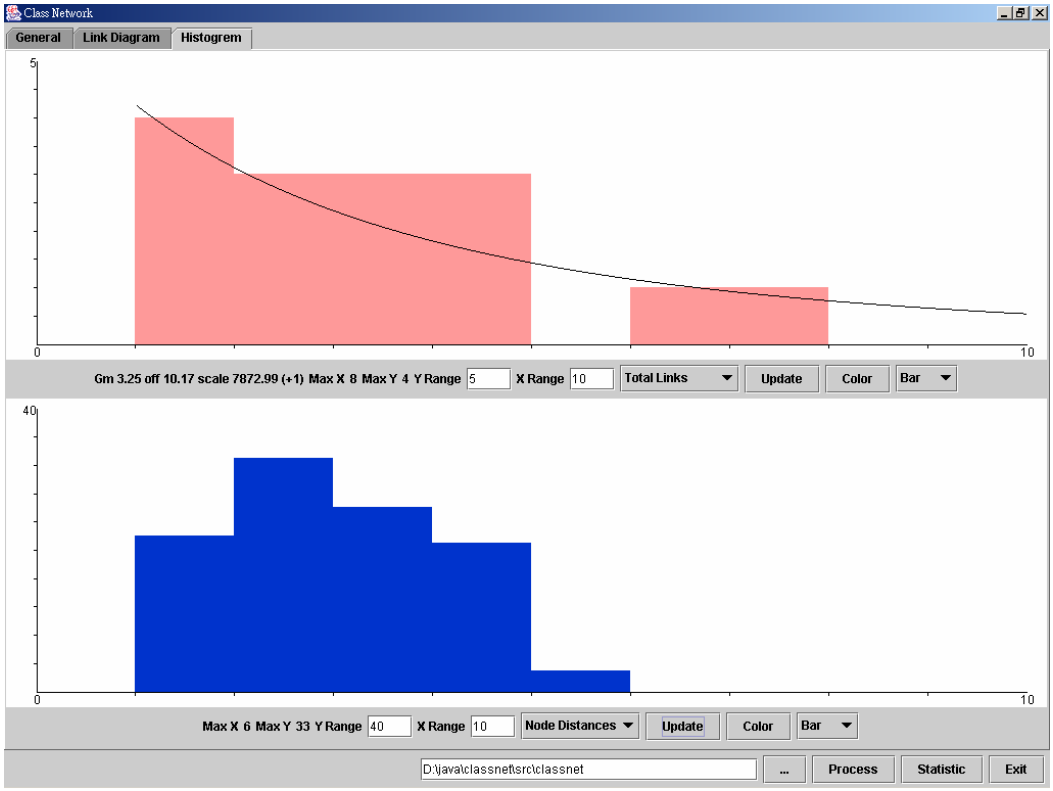


Figure 114. The degree distribution of total number of links and the node distance distribution of package *classnet*.

Appendix F Screenshots of GSET

The Genetic Software Engineering Toolkit (GSET) is a software tool help to realize GSE process. The functionalities of this tool have been introduced in Chapter 7; following images are some of the screenshots of GSET.

Figure 115. The splash screen of GSET.

Figure 116 shows the display style and layout of GSET. On the left side, there is a window showing the navigating tree. In the tree, all the components, RBTs, DBTs and other design diagrams are listed and arranged according to their category. In the right part, there is a panel that can display multiple windows and each window holds a design document.

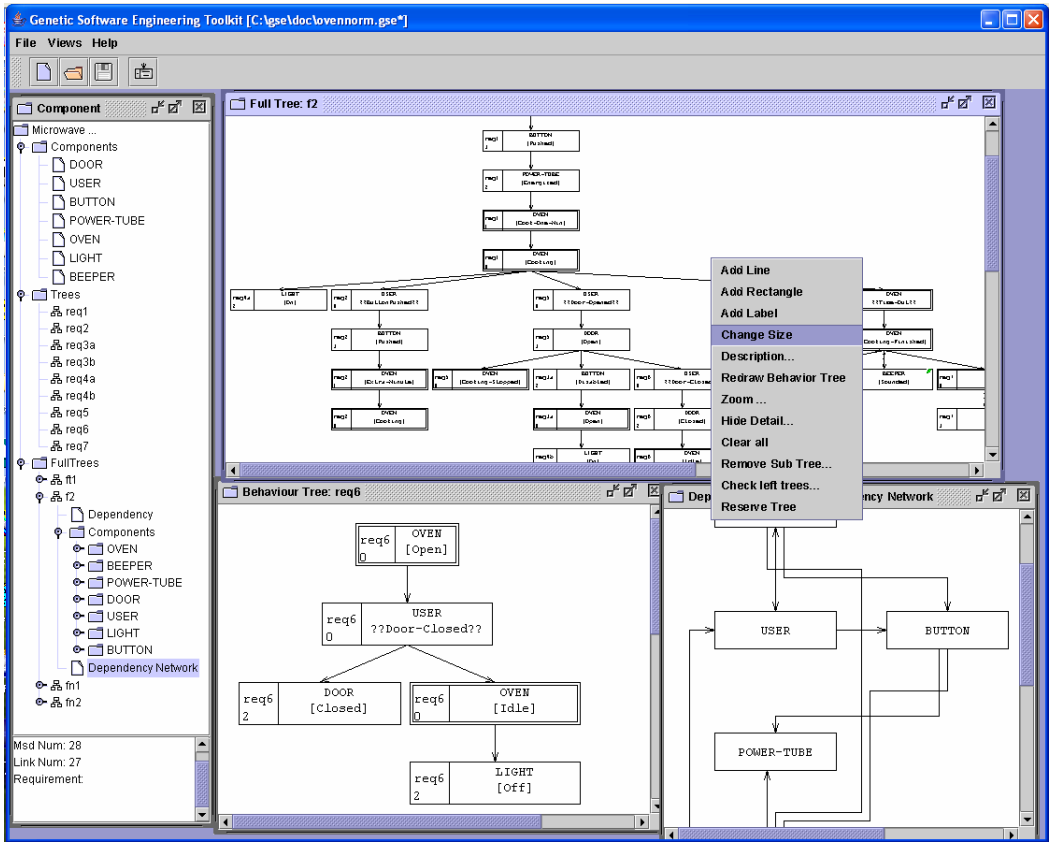


Figure 116. The GUI interface of GSET

All the design documents displayed in GSET can be exported into files of PDF format. The user can select to export a few diagrams as well as the whole design set. The exporting selection dialogue is shown in Figure 117.

Figure 118 shows the dialogue box used to add or edit a message in a RBT. Figure 119 shows the RIT (requirement integration table) that is useful to check if all the RBTs can be integrated into a DBT.

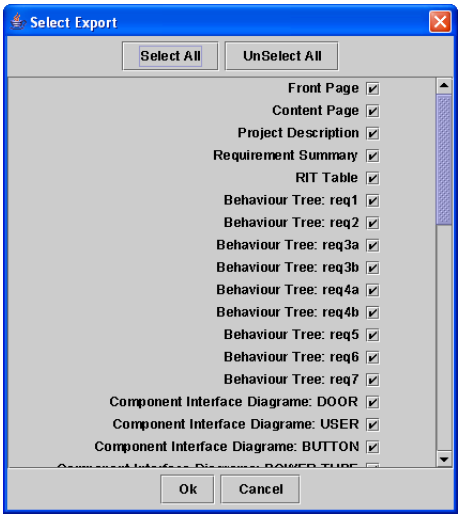


Figure 117. The export selection dialogue box.

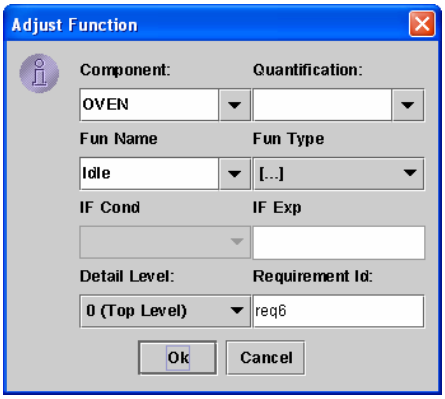


Figure 118. The RBT method editing dialog box.

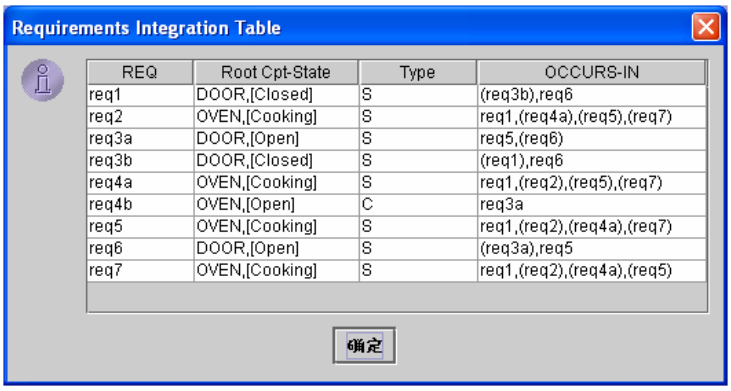


Figure 119. The RIT(Requirements Integration Table) generated by GSET

Figure 120 is a DBT with all the details shown. If we hide all the low level details

and only display the top level behaviors, we will have a high level behavior tree in Figure 121. From Figure 121, people can understand the overall behavior of the targeted system very quickly. In the high level behavior tree, some methods are marked with dot to indicate that there are low level behaviors hidden on that spot. We can exam the hidden behavior on a separate window (See Figure 122). From this figure, the hidden behaviors under the event OVEN??Time-Out?? are shown as a behavior tree.

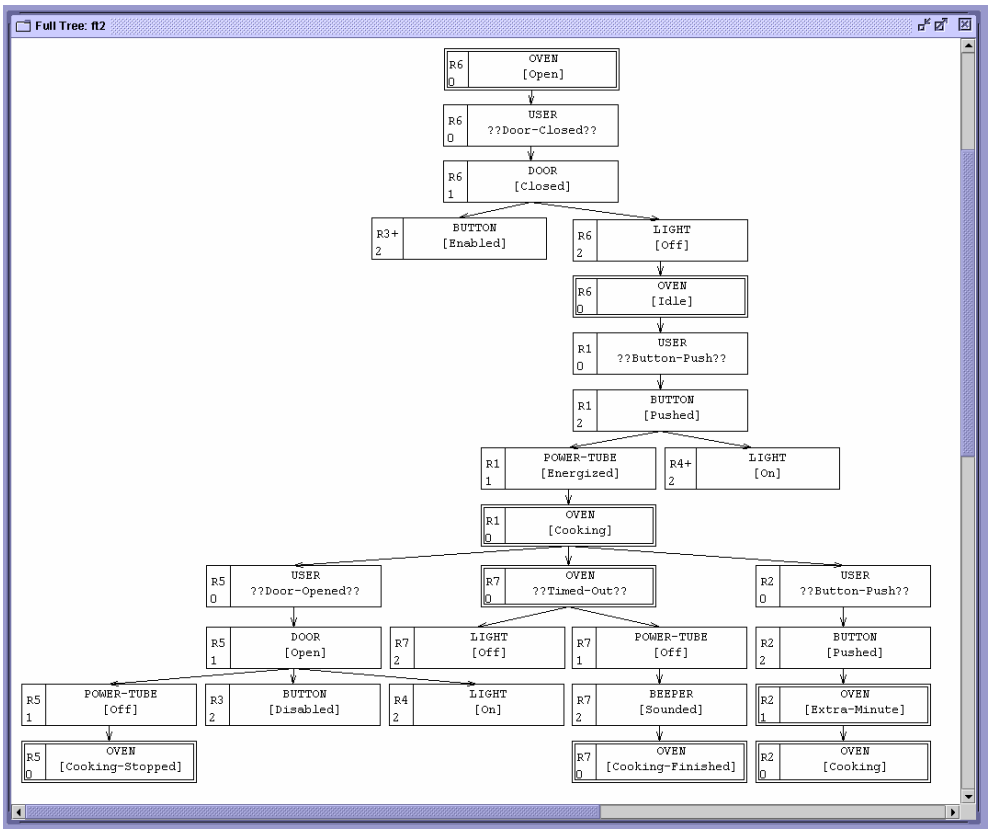


Figure 120. A DBT of all details shown in GSET

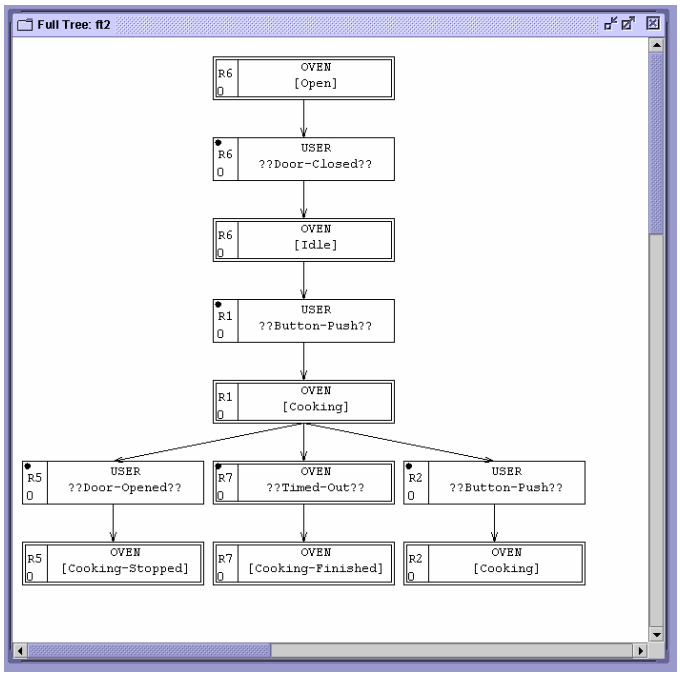


Figure 121. The same DBT with only the top level of information shown.

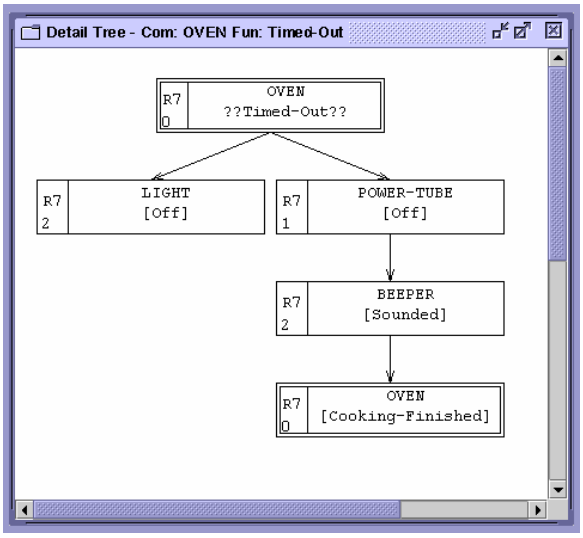


Figure 122. The hidden tree under the node of OVEN??TimeOut?? in the previous figure.

Appendix G Screenshots of Class Network

Class Network is a tool used to explore the component dependency network of Java packages. The functionalities are introduced in Chapter 7. Some of the screenshots are presented below.

a. The General Screen

After the tool is executed, you will see the general screen.

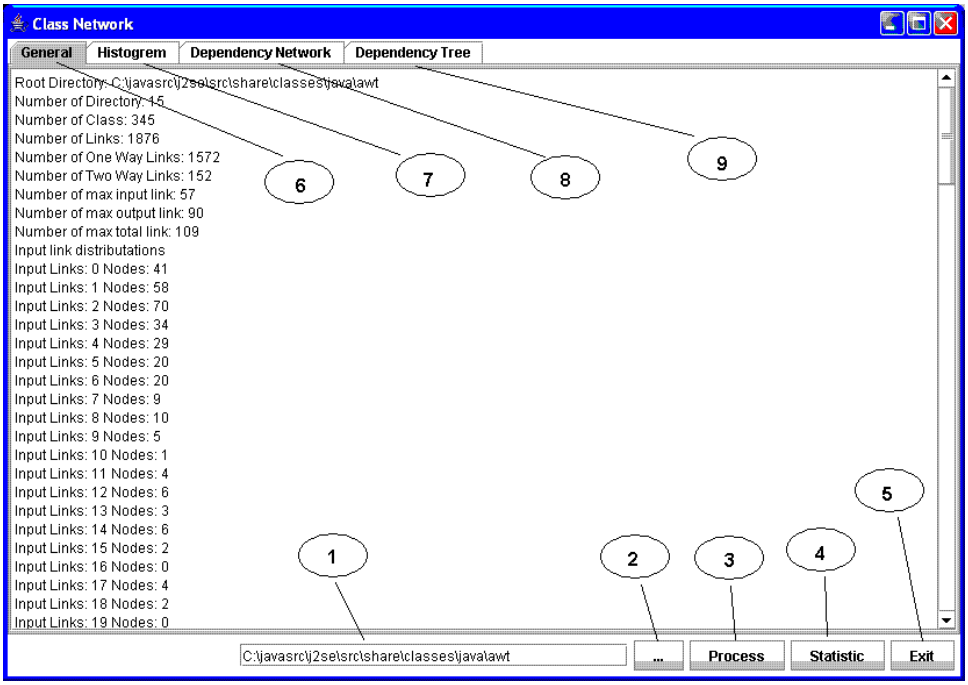


Figure 123. The General Screen

In the General Screen, the middle text area is used to display the statistic information of a Java package's CDN. The functions of other controls are:

1. **The location of the targeted Java package**. To use the tool to investigate the CDN of a Java package, you can manually input the absolute path of that package. Please be aware that this tool can only explore the source code.
2. **Button ...**: besides to manually input the location of a targeted Java package, click this button will pop up a standard file choicer, and you can selected the path by using the file choicer.
3. **Button "Process"**: after a Java package is selected, click this button will start the process. After the process is finished, some information will be displayed in the main display area of this screen. Then you can go to other screens to check further information.
4. **Button "Statistic"**: after a Java package has been processed, click this button will show some statistic data of the Java package's CDN.
5. **Button "Exit"**: click this button will exit the tool.
6. **6-9**: switch between different screens.

b. The Histogram Screen

After a Java package is selected and processed. You can switch to the histogram screen to check the degree distributions. Figure 124 shows a typical histogram screen.

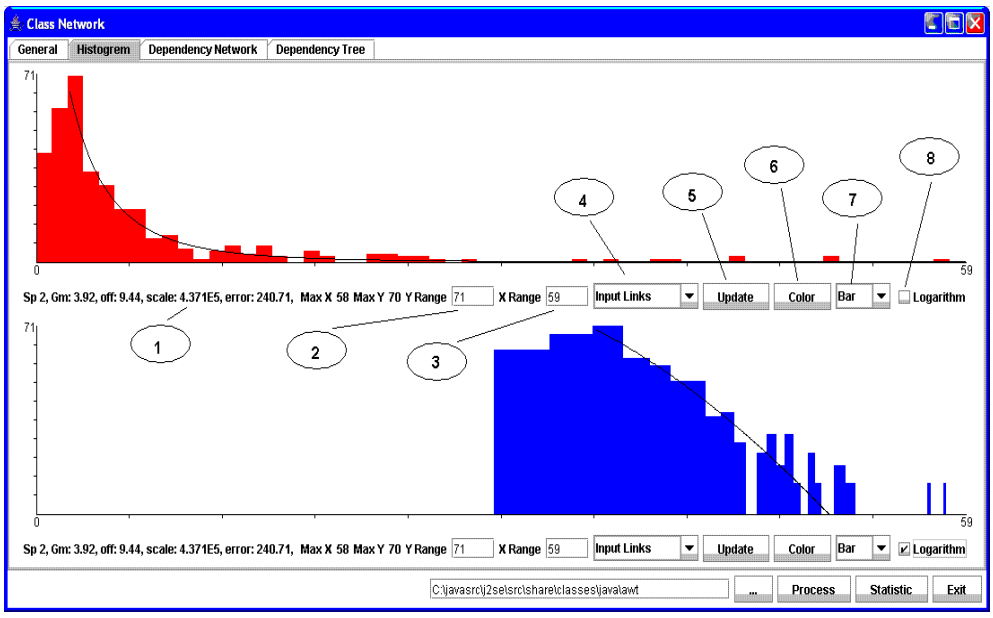


Figure 124. The Histogram Screen

The histogram screen includes two display panels so it can display two different distributions independently. The functions and controls for each panel are exactly the same. Here we only introduce the controls for the top panel. In a distribution diagram, the x-axis is the number of connections and the y-axis is the number of nodes. The black curve is drawn based on the estimated power-law function.

1. **Distribution information**: once a distribution is shown in a panel, some information of the distribution will be shown here.
2. **Text field "Y Range"**: use this field to reset the maximum y shown in the diagram.
3. **Text field "X Range"**: use this field to reset the maximum x shown in the diagram.
4. **Distribution selection**: different distributions can be selected from this list.
5. **Button "Update"**: after a distribution is selected, click this button will show the distribution in the panel. The value in the Y Range and X Range is automatically set based on the data range of the distribution, but they can be changed . After these values are changed, clicking this button will redraw the distribution based on the new x and y ranges.
6. **Button "Color"**: use this button to change the color of the diagram.
7. **Diagram type selection**: use this control select the drawing style
8. **Checkbox Logarithm**: Checking this box will set the distribution display in a logarithm scale. Otherwise in linear scale

c. The Dependency Network Screen

After a Java package has been processed, the CDN will be shown in the dependency network screen as in Figure 125 Right click on the blank part of the screen will bring up a menu. Please notice that right clicks on a node or a link will bring up different menus. Most items on these menus are self-explaining. Here we only briefly introduced three items on the menu if right click on the blank area of the screen.

1. **Menu item "Change Size"**: this item is used to change the size of the canvas. When you change the size of the main frame of the tool, this value can be auto set.
2. **Menu item "Show Tops"**: once you click this menu item, a pop up window will be displayed and ask you to input a value. If you input 10, then the 10 nodes with top number of connections will be displayed with a label to indicate the name of the Java class that is associated with this node.
3. **Menu item "Self Adjustment"**: initially, a CDN is drawn with each node randomly in the screen, check this item will make the system re-arrange the position of the nodes so the CDN is displayed in a simpler view. You may uncheck this item after the nodes are in suitable positions. Please notice that you can also use the mouse to drag individual node or a group nodes to any position in the screen

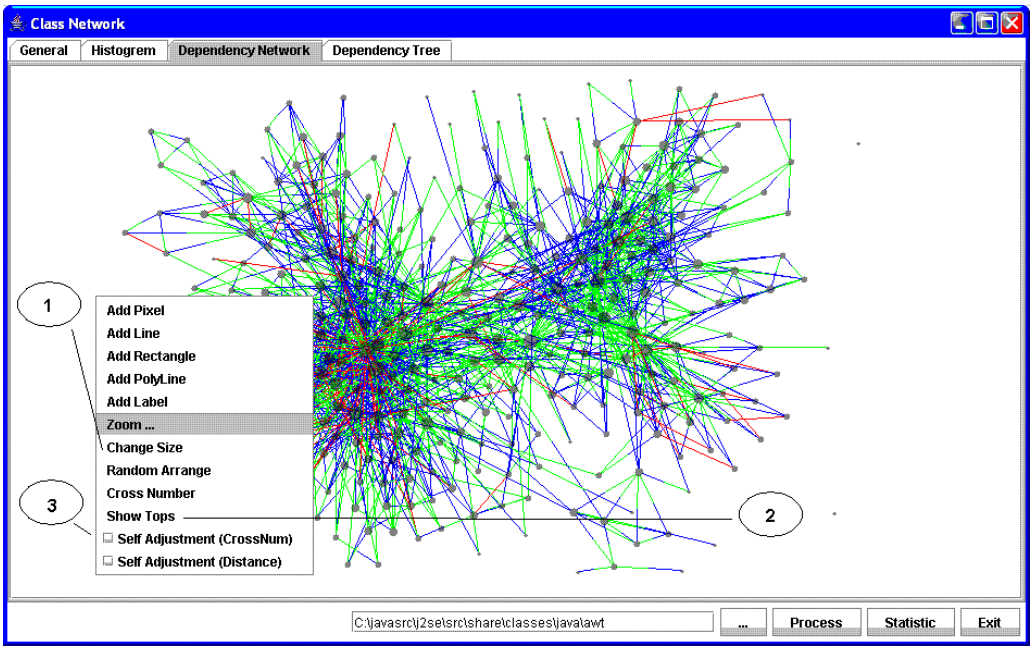


Figure 125. The Dependency Network Screen

d. The Dependency Tree Screen

After a Java package has been processed, the dependency tree of any class (or interface) in that Java package can be displayed in the dependency tree screen (As in Figure 126). Use the mouse to right click a blank part of the screen will bring up a

mean. Most menu items on the mean are self-explaining; we only introduce the function of three items.

1. **Menu item "Add Root Node"**: click this menu item will pop up the list of all the classes (interfaces) in the package, you can select one from the list as the root node. Once a root node is selected, the corresponding dependency tree will be drawn.
2. **Menu item "Zoom"**: sometimes the dependency tree can be very large so you need to use the Zoom function to change the zoom scale so you can view the overall structure of the tree.
3. **Menu item "Clear All"**: after one dependency tree has been displayed and you want to check other dependency trees, click this menu item to clear everything on this screen.

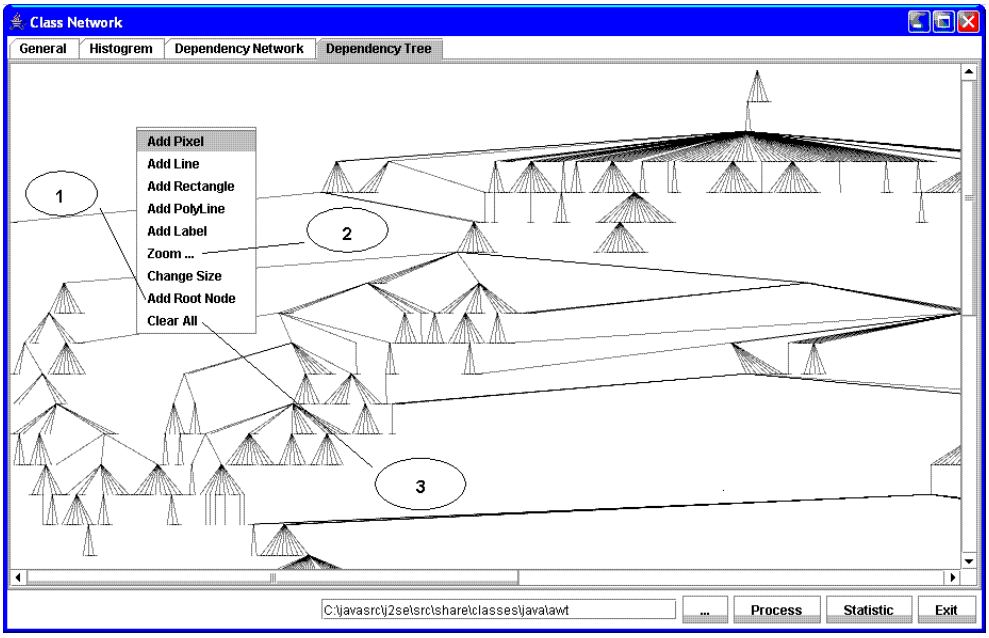


Figure 126. The Dependency Tree Screen

Appendix H Screenshots of the SCNE

a. The Sort Screen

Once the tool is started, the first screen is the sort screen (Figure 127).

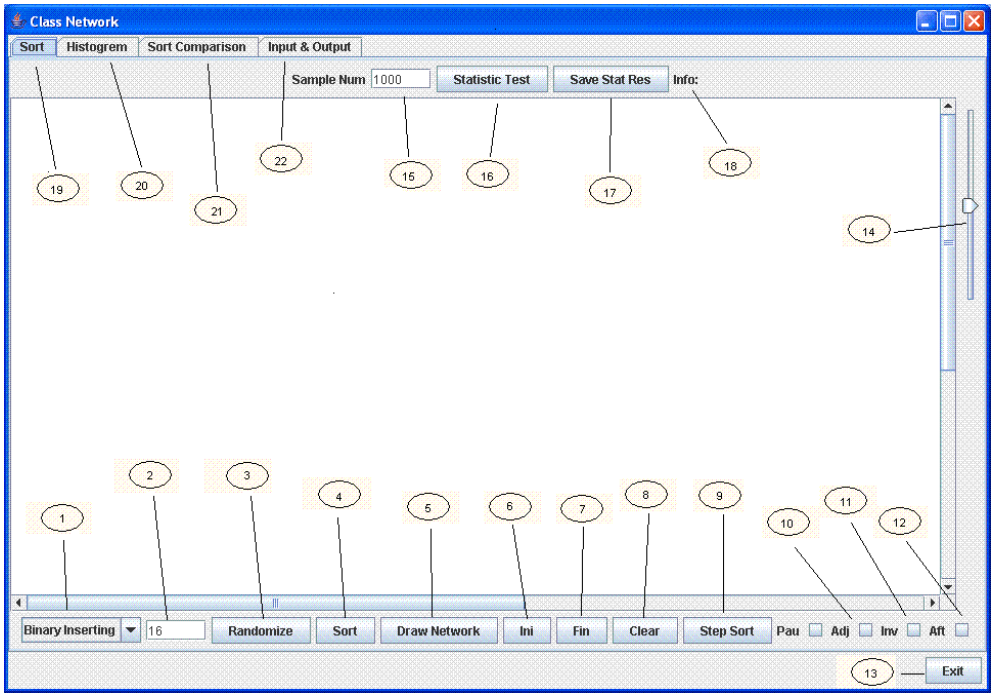


Figure 127. The screen "Sort"

The meaning and usage of those controls are:

1. To use this dropdown list to select different sorting algorithms.
2. To use this text input field to input the length of a target sequence. I usually use 16,256 and 1024 in this field, but of course you can try other numbers.
3. Button "Randomize": once you have selected the sorting algorithm and the length of the record sequence, you press this button to generate a random sequence. The result of the generated sequence can be viewed and adjusted in the "Input & Output" screen.

4. Button "Sort": once a random sequence is generated, click this button will do the "sorting". After the sorting process has been finished, the comparisons can be viewed in the screen "Input & Output".
5. Button "Draw Network": once a sorting process is finished, click this button will draw the sorting comparison network (SCN) in the screen. The position of the nodes are randomly selected. The size of the node is determined by the number of connections to that node[26].
6. Button "Ini": after a SCN is drawn in the screen, click this button will put the positions of all the records in a circle ordered by the positions of the corresponding records in the initial sequence.
7. Button "Fin": after a SCN is drawn in the screen, click this button will put the positions of all the records in a circle ordered by the positions of the corresponding records in the sorted sequence.
8. Button "Clear": after a SCN is drawn, click this button will clear all the connections in the SCN.
9. Button "Step Sort": after a SCN is drawn, click this button will enter step sort mode. In step soft mode, the sorting process will be shown step by step.
10. Check box "Adj": after a SCN is drawn, tick "Adj" will make the system rearrange the location of nodes according to force directed algorithm. Un-tick this check box will stop the procedure. Please be aware this function also works in the step sort mode.
11. Check box "Inv": after a SCN is drawn, tick this check box will hide all the redundant links (working in the step sort mode)
12. Check box "Aft": after a SCN is drawn and the check box "Adj" is ticked, tick this check box will disable the effect of the redundant links when using the force directed algorithm
13. Button "Exit": Exit the system
14. Slide bar: control the sorting speed in step sort mode.
15. Controls 15 - 18 only work in statistic mode. Text field "sample number": the number of independent samples.
16. Button "Statistic Test": click this button will perform the sorting in statistic mode. The sorting algorithm is determined in Control 1, the number of independent samples is determined by the value in control 15 and the

[26] CAUTION: When using bubble sort and the length of the sequence is larger than 512, never try to draw the network. Because the number of connections is too huge that it may consume all the memory.

number of records in each sample is determined by the value in Control 2. After this button is click, independent randomly generated sequences will be sorted and the average SCN properties will be calculated. During a "statistic test" process, the number of samples has been sorted will be printed in the console.

17. Button "Save Stat Res", after a "statistic test" has been performed, click this button will save the result.
18. Info: after a "statistic test" has been performed, some of the information will be shown here.
19. Page "Sort": click it will show the sort screen.
20. Page "Histogram": click it will show the histogram screen
21. Page "Sort Comparison": click it will show the "sort comparison" screen.
22. Page "Input & Output": click it will show the "input & output" screen.

b. Example: show a SCN of Binary insertion

1. Start the tool
2. Select Binary Inserting in control 1
3. Input 256 in the control 2
4. Click Button "Randomize"
5. Click Button "Sort"
6. Click Button "Draw Network", now the screen will show a very messy network.
7. Check "Adj", now the network start to adjust.
8. After about 30 second and uncheck "Adj", you will get a screen similar to the following Figure 128:

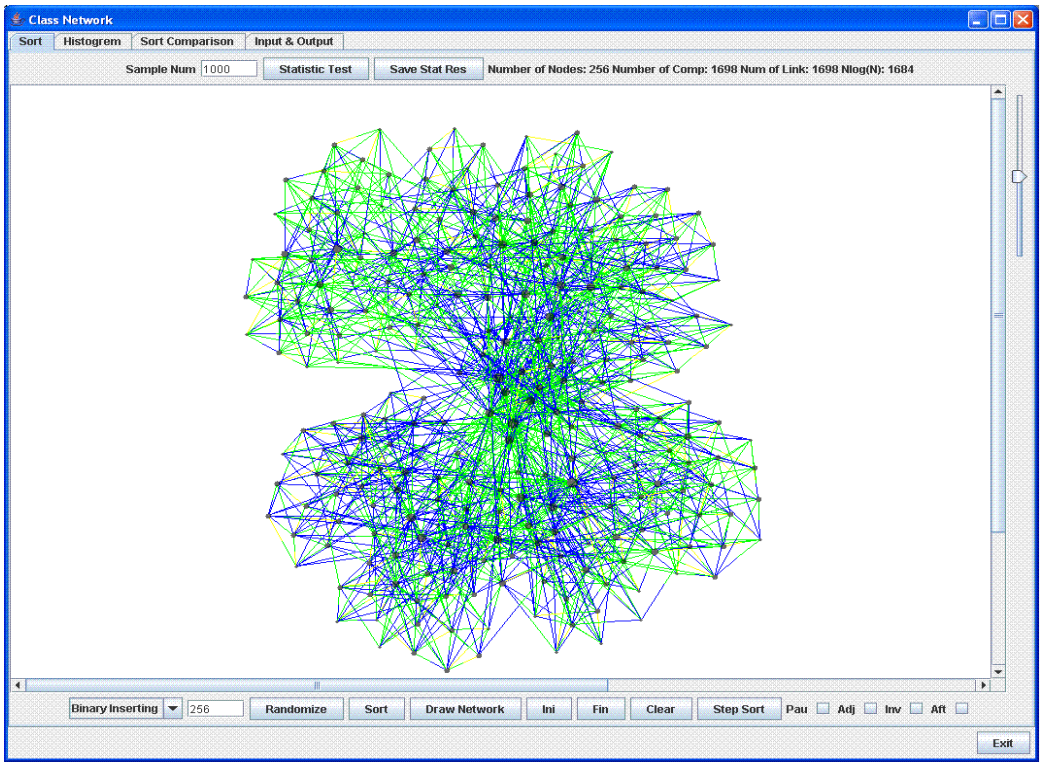


Figure 128. The SCN of a random sorted by binary insertion sorting algorithm. Sequence length is 256

c. The Histogram Screen

Click the page "histogram" will show the histogram screen as below (Figure 129):

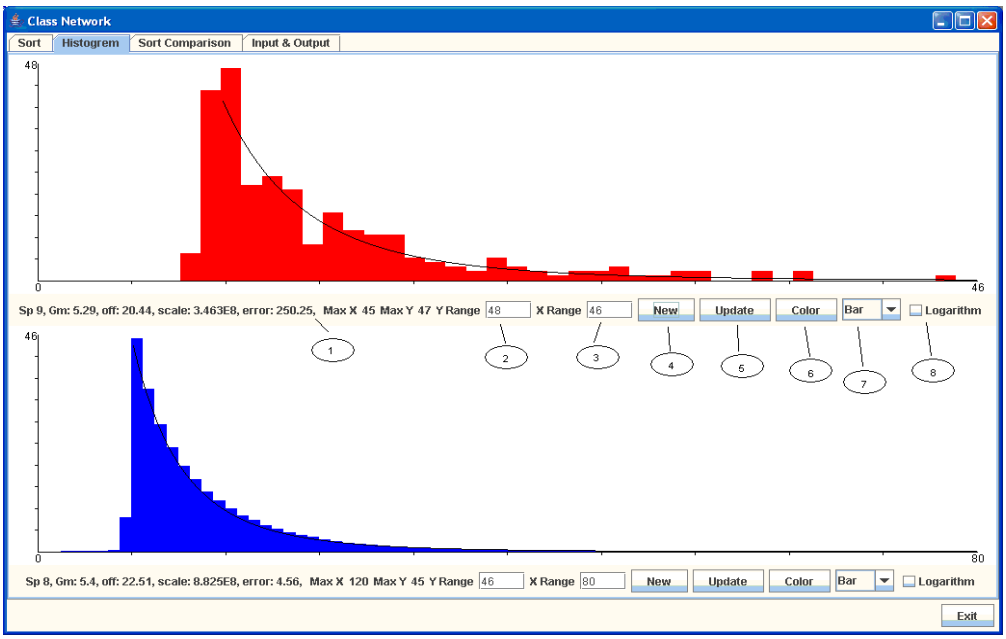


Figure 129. The histogram diagrams. The top one is the degree distribution of a SCN of a single test and the bottom diagram shows the average degree distribution based on 1000 independent tests

The histogram screen includes two display panels so it can display two different distributions independently. The functions and controls for each panel are exactly the same. Here we only introduce the controls for the top panel. In a distribution diagram, the x-axis is the number of connections and the y-axis is the number of nodes. The black curve is the drawn based on the estimated power-law function. For details please check Appendix D.

1. Distribution information: once a distribution is shown in a panel, some information of the distribution will be shown here.
2. Text field "Y Range": use this field to reset the maximum y shown in the diagram.
3. Text field "X Range": use this field to reset the maximum x shown in the diagram.
4. Button "New": after a sort or a static sort has been performed in the sort screen, click this button will show the degree distribution and power-law curve. The maximum x and maximum y in the diagram is automatically set by the software.

5. Button "Update": after a distribution is shown in the panel, the value in the Y Range and X Range can be changed . After these values are changed, click this button will redrawn the distribution based on the new maximum x and y.
6. Button "Color": use this button to change the color of the diagram.
7. Drop down menu: use this to select the drawing style
8. Check this box will set the distribution display in a logarithm scale.

d. The Soft Comparison Screen

Click the page "Sort Comparison" will shown the sort comparison screen as (Figure 130):

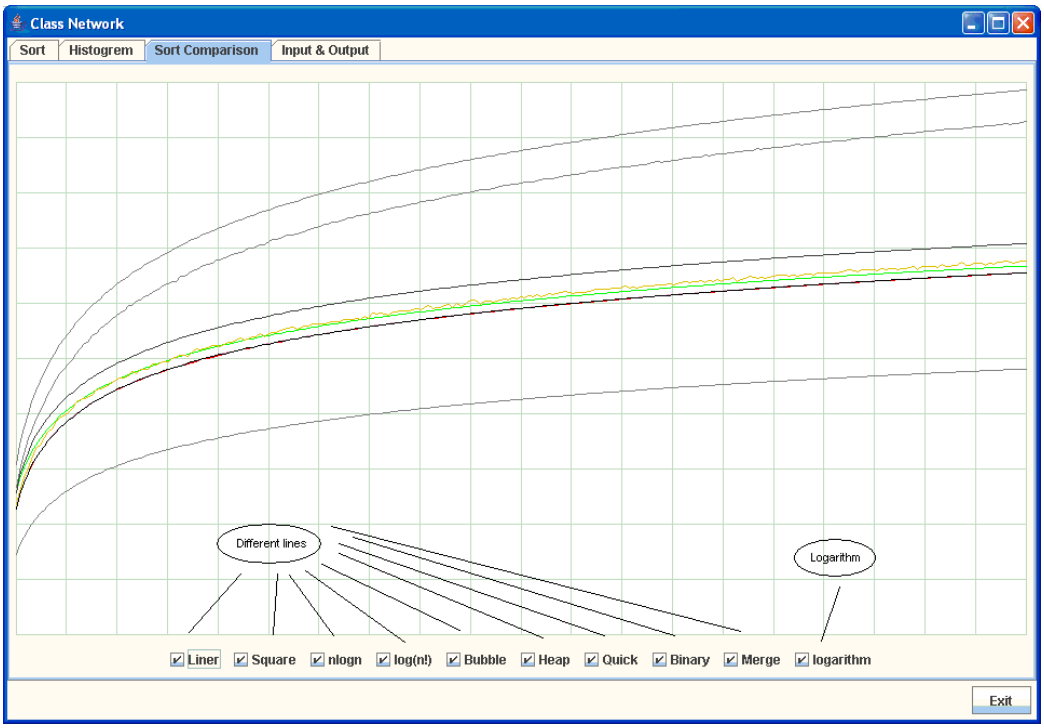


Figure 130. The comparison of different sorting algorithm

In the sort comparison screen, there are 9 different lines. 5 of them corresponding the 5 different sorting algorithm and the rest 4 are compared lines (n, n*log(n),

log(n!) and n*(n-1)/2). The x range from 16 to 1024. At the bottom of the screen, there are 10 check boxes. Each of the first 9 is associated with a displayed line and the last one determined if the curved are displayed in linear scale or log scale.

e. The Input & Output Screen

The last screen is the "input & output" screen as (Figure 131):

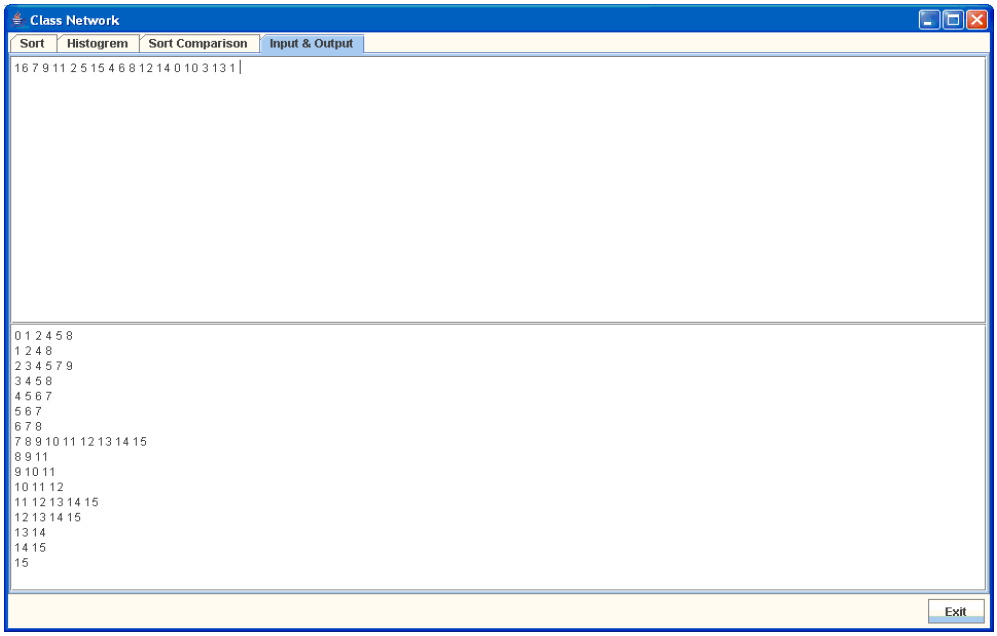


Figure 131. The input & output screen

In the "input & output" screen, there are two display panels. After a sequence is sorted, the top panel display the input sequence. The first number is the number of records of the sequence and rest is the actual sequence. The bottom panel display the real comparisons. Please be aware that the values displayed in the bottom panel are values of the records not their index. The first line is "0 1 2 4 5 8", which record 0 has been compared with record 1,2,4,5,and 8 etc. When the button of

"Randomize" is clicked in the sort screen, the value in the top panel is auto set, but a user can also manually set the value in the top panel to test the sorting process.

Appendix I　Discussion of Possible Future Works

a. Why All Complex Networks Are Not Scale-Free

In Chapter 6, we have briefly introduced the concept of the scale-free network. It has been discovered that most complex networks from different disciplines are scale-free, but this not always the case. A good example is large railway networks (Figure 132).

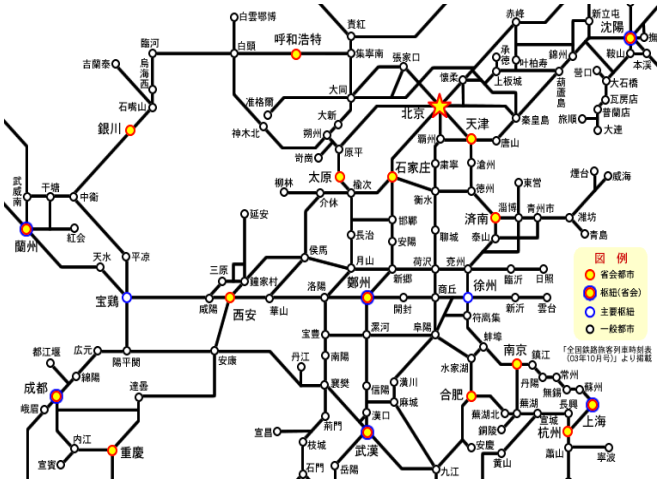


Figure 132. A typical railway network

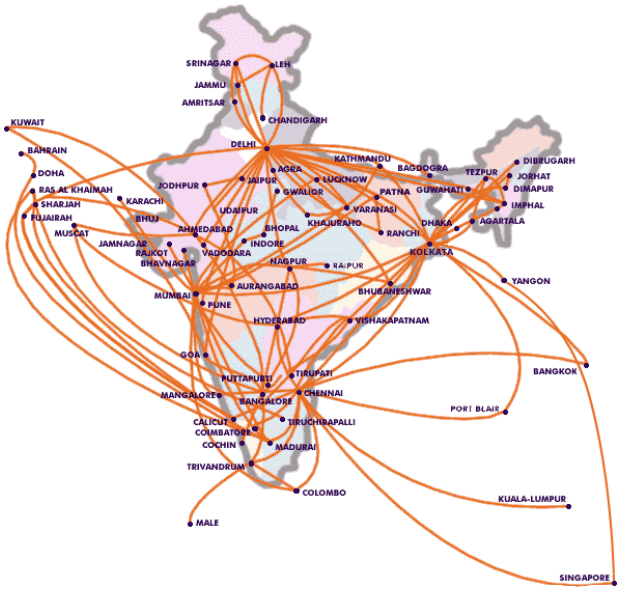


Figure 133. A typical airline network

Figure 132 and Figure 133 show a typical railway network and a typical airline network. It is obvious that the airline network has a few hubs and that is the most significant feature of a scale-free network, but the railway network does not have this feature. Consequently, we may draw the conclusion that large airline networks are likely to be scale-free networks but large railway networks are not. This leads to a very interesting question. Both are transport networks, but why one is scale-free and the other not?

This question may have multiple answers. Here, we propose only one. Unlike an airline network, a railway network is plane graph (Diestel 1999) built on a two-dimensional (2-D) surface. Checking the railway network in Figure 133, we discover that, for a railway network, even though the intersections of links are not totally forbidden, it is very limited for links to intersect in the 2-D network. Then we may have a conjecture, *if a network is built on a 2-D surface and the intersection of links is prohibited, then this network will not be able to evolve into a scale-free network.*

The limitations of a scale-free network on a 2-D surface inspire us to think about another interesting mathematical problem: the four-color mapping problem. The original problem is that, for any map drawn on a 2-D surface, we need at most 4 different colors to draw each area and guarantee that there exists a drawing method so that no two neighbor areas (that means there is a piece of mutual boundary between the two areas) are drawn in the same color. This problem can be transferred into a network problem. Each area can be treated as a node and a link between two nodes indicates the two represented areas are neighbors. The delicate part is that the network is built on a 2-D surface and no two links can be intersected.

If we shift the 4-color mapping problem into a 3-D space, what is the minimum number of colors that is sufficient to define any two connected nodes? The answer is obvious: no fixed number of colors can guarantee that any connected nodes can be colored differently for arbitrary networks built in a 3-D space. When the number of nodes and the number of connections are increased, the minimum number of colors required will be increased without any limitation. This result indicates that a 3-D space is fundamentally different from a 2-D space in regard to the ability to contain complex networks. The 4-color mapping problem addresses a crucial restriction for the complexity of networks that can be evolved in a 2-D space. The circuit layout problems (Vancleemput 1974) also reflect a similar limitation in a 2-D space. However, when we consider a 3-D space, none of the discussed restrictions exist. Networks of any topological structures are capable of being built in a 3-D space. Then we come back to the original question in this section. Scale-free networks are large and complex networks that can only be

developed in a space of more than two dimensions.

Many large scale-free networks are built in virtual space where the dimension is hard to determine, such as the human relationship networks. However physical complex networks such as a brain, which is a network of nerve cells, are built in a 3-D space. We may conjecture that, due to the limitations for forming complex networks, objects of complexity such as life can only be created (or evolve) in a universe of at least three dimensions.

b. Searching Methods

As we have discussed in previous chapters, large networks are built by incrementing of their size. During the growth process, there are two essential operations. The first is to add new nodes, and the second to create new connections in the existing network. Here we discuss only the creation of new links.

For different purposes, a node in a network needs to be connected to other nodes. The problem is how the source node could identify the most suitable target node for it to connect with. Generally, there are three types of searching methods:

1. **Random searching**: where the source node tries to connect to other nodes randomly and there is a chance that it may find its best target. The chance increases if the node keeps trying. This method mostly relies on luck and may be applied in certain circumstance (I wonder whether Edison has adapted this methodology when he was trying to find a suitable material for the thread in a light bulb). Usually, it is not a good strategy and may take a

long time to find a suitable target and establish a useful connection.

2. **External diverted searching**: It means that the source node discovers the best possible connection from an external information provider. This searching method is based on the assumption that, in a given network, for the given source node, there exists a most suitable target node and this knowledge is known by someone. If the source node can acquire this knowledge, it can directly make the best connection. For a system designed by an individual, that person has an overall view of the system and total control over the system; thus the designer can be the external information provider and have the knowledge to directly link the source node to the best target node. However, for many real-world large systems, those preconditions do not apply; the external information provider either does not exist or is unapproachable. For example, in the social network, if a person is looking for a job, there may be a best job opportunity for him but the problem is how he could know that. If he believes that God must have this knowledge, then praying probably is a starting point if he wants to apply the "External diverted searching" method for his job searching purpose.
3. **Self-adjusting searching**: At the beginning, the source node tries to determine a possible best testing node based on its own knowledge and makes a testing connection. (If it does not have any useful information at all, it may apply the random searching method as the first step.) After one or a few testing connections, the source node will be able to collect some feedback information from the connected nodes. This information may help the source node to narrow the search scope and find better testing nodes. After trying the new testing nodes and getting new information, the source node will eventually find a satisfied source node and make a link. This

search method is obviously the most practical one and is adapted by most real networks. In the job search example, a possible scenario is that the person may ask his most knowledgeable friend first; his friend may advise him to try a job agent; the job agent may recommend a potential employer to this person. Generally, this searching method is a learning process. The more the source node learns from the existing connections, the quicker it is able to reach a suitable target node.

Above we have discussed 3 different searching methods for a source node to discover a most suitable target node. The topological structure of the network will be made different by the different searching methods.

If the random searching method is applied, the growing network may have the structure of a random network. If the external diverted searching method is applicable, the generated network will not have redundant connections and can be in its simplest form. (Here we suspect that, for most systems, the simplest form of the corresponding network can be a tree.)

In general, if the self-adjust searching method has adapted an optimized way to use the information collected from the testing connection, we conjecture that the network will be woven scale-free. The testing results of the sorting comparison networks support this conjecture. A sorting algorithm uses the self-adjust searching method to construct the sorting comparison network. After each comparison, the new comparison will be selected based on the previous comparison results. A more efficient sorting algorithm simply means it makes better use of the information collected from the testing connections. An optimized sorting

algorithm can nearly maximize the usage of the information and, in this situation, we know that the sorting comparison networks are scale-free.

For large software systems, since they are usually "grown" rather than built (Brooks 1987), the information about the best connection points for new components is usually not clear. The lack of information leads to more connections than necessary, and the growing process is similar to a network developed on a self-adjusting searching method and it is not surprising that the final structure is a scale-free network.

c. From Scale-Free Networks to Trees

A complex network has a large number of connections. Regarding the functionalities of the network, some of the connections are essential and the others are more or less redundant. The existence of the redundant connections may be the result of the testing connections, poor designs, or changes of the functionality of the network.

In many situations, a new connection may cause some of the old connections to be redundant. For example, in a sorting comparison network, once all the contiguous connections are discovered, all the other connections become unnecessary. In fact, even half way through a sorting process, it is obvious that some of the comparison results can be deduced from other comparison results and this means these links can be removed without affecting the sorting process. When the sorting process is running, new links are being continuously added; during this period, we can run another process to clean up and remove the redundant links. Even though the

cleaned-up process will not increase the efficiency of sorting, it always makes the sorting network relatively clear and simple in form.

It is the same as the job search example: once the person has found a suitable job, his connections to the job agency and even to his friend become redundant and can be cut off if the person's only purpose is to find a job and he enjoys a simple life.

When we examine a large software system, we will discover similar phenomena. New functional requirements and software changes will keep the developers adding new code into the system (sometimes in a 'copy, paste and make a few changes' manner), but the developers are usually reluctant to remove expired code. If we study this problem from the network point of view, we see that this preference will add more nodes and connections into the network and make the system hard to understand and hard to maintain. If a methodology is introduced to clean up all the redundant code at the same time as the software system is being built, the system can be in a simpler, clearer and more efficient form. A similar concept has already been addressed in software engineering (Lehman 1974), the problem is how to identify if all the redundant pieces of a software system have been cleaned up.

The topological structure of the network provides a practical criterion for judging the redundancy of a software system. The simplest form of a network is a tree. In the sorting comparison network, if we remove all the non-contiguous links, the network becomes a path and it is a special form of tree. Although we cannot expect that the component dependency network of a large software system can be easily cleaned up so that it becomes a tree, the number of connections is a quantified indicator for the complexity of a system.

d. Reuse of Components

Large systems are built with numerous components. If the system is very large, the number of components can be huge. Each component is built based on its own specification and usually it cannot be replaced by other components. The problem is whether we can reuse those components in other systems.

Theoretically, the answer is yes, but in practice, it is difficult. The reasons include:

1. A component may be dependant on other components to function properly. Therefore, when one component of a large system is reused, many more components in the old system may need to migrate into the new system to support the reused one.
2. The functionality of common components can be limited. Some components such as Abstract Data Types (ADT) are very common and can be reused in many different systems. However, the functionalities of this kind of component are usually primary. Besides those common components, large systems need high-level components that are usually specific to the functions of the system and can only be reused in different systems with difficulty.
3. To modify a component from another system is challenging. For a new software system, we may find some components from other software systems that have similar functions and have the potential to be reused. However, similar does not mean exactly the same. Even minor differences means that these components cannot be reused without modification, but to modify

components from other systems (if they are not developed by the same team) is very difficult.

4. It is difficult to discover the reusable components. When a new system is being built, it is possible that there may be some components that can be reused in the new system, but the problem is how to find out about those components.

Is there a better approach so we can reuse the components relatively easily?

Let us look at three real systems of huge scale. The first is society, the second is a human body and the third is the Internet. Society is a system and its components are human beings; a human body is a system with cells as components and finally the Internet is a system with millions of computers as the components. The three large systems still include other types of components but what I have mentioned occupies a significant portion of the components in those systems.

There is a parallel feature in those three large systems. Even though each individual component is unique, all the components of one system belong to the same family. They are usually born equal and the difference, which is minor if compared with the similarity between components, is adapted later on. For example, we cannot find two people identical but, in general, the internal structures of any two people are nearly exactly the same. For cells, each cell in a human body bears the same set of genes. For computers, each computer has a similar structure and the same functions such as calculating, storing information and communicating with other computers. By applying the universal computer model (Cover 1991), any computer is theoretically equal to an abstract universal computer.

Because of the isogenic feature of the components, each component has the potential to perform the tasks of different roles. The functions of a component can be easily changed and the component can be reused in another system. For example, the same group of people can form a business company, a sport team or even a tour group if each person is assigned a suitable role.

Inspired by those real large systems, we propose a new approach to building component-based software systems. As other component-based software systems, this component-based software system needs a unified environment as host. The different part is that we will not design and develop different components based on the functional requirements, but we will only build one (or a few) very complex components as prototypes. This component will have the potential to perform most ordinary tasks required for normal components, it can acquire its behavior based on a high-level description language such as CBT (component behavior tree) and it can communicate with other components.

When a new system is being built, the system will duplicate a number of components from the prototype and each component will be assigned a piece of script to describe its expected behaviors and its relationship with other components, then the system is finished. When new functions are added and a component copied from the prototype can not handle them, we will update the prototype rather than the duplicated component, so the prototype will be more powerful and can be reused for more purposes.

The philosophy of this approach is that, even though the cost of building a sophisticated prototype is much higher than building an ordinary component, the

duplication of the prototype will cost nearly nothing and it can be reused many times in one or multiple systems.

For traditional software engineering, besides the analysis of the user requirements, most of the effort is used to translate those requirements into designs and then into source code. During this process, the entropy has been increased dramatically. In other words, the entropy of the system in the solution domain is much larger than that in the problem domain. For two different software systems, even though the entropy difference in the problem domain may be minor, the entropy difference in the solution domain will be much larger and the increasing of the entropy difference results in high costs when an existing system is changed or transferred into another one. One good example for this point is the millennium bug problem. The description of the change in the requirement domain can be as short as one sentence "change the year format from 2 digitals to 4 digitals", but the change in the solution domain is huge and it has cost millions or billions of dollars.

In our proposed approach, a universal component, which is hosted in a well-designed platform, can learn the behavior from a script describing the functional requirements in the problem domain. Once a system has been described in the problem domain, the platform can automatically map the system into the solution domain. In this situation, the conditional entropy of the system in the solution domain equals to the entropy in the problem domain (the conditional entropy means the existing platform is fixed.) The entropy has not been enlarged during the process when the system is transferred from the problem domain to the solution domain. Therefore it keeps the cost to change and maintain the system to the minimum.

Bibliography

Abrial, J-R., "*The B-Book: Assigning Programs to Meanings*", ISBN: 0521496195, Cambridge University Press, 1996.

Ahl, V., Allen, T.F.H., "*Hierarchy Theory, a Vision, Vocabulary, and Epistemology*", ISBN: 0231084803, Columbia University Press, New York, 1996.

Akşit, M., "*Software Architectures and Component Technology*", ISBN: 0792375769, Kluwer Academic Publishers, 2002.

Alanen, M., Porres, I., "*Difference and Union of Models*", Lecture Notes in Computer Science, Vol: 2863, Spinger-Verlag, pp. 2-17, 2003.

Albin, S. T., 2003, "*The Art of Software Architecture, Design Methods and Techniques*", ISBN: 0471228869, Wiley Publishing, Inc., 2003.

Alberich, R., Miro-Julia, J., Rosselló, F., "*Marvel Universe looks almost like a real social network*", oai:arXiv:cond-mat/0202174, v1, 11 Feb 2002.

Albert, R. and Barabási, A., "*Topology of Evolving Network: Local Events and Universality*", Physical Review Letters, Volume 85, Number 24 pp. 5234-5237, December 2000.

Albert, R., Jeong, H., and Barabási, A., "*Diameter of the World-wide Web*", Nature, Volume 401, pp. 130-131, September, 1999.

Albert, R. and Barabási, A., "*Statistical Mechanics of Complex Networks*", Reviews of Modern Physics, Volume 74, pp. 47-97, January 2002.

Albin, S. T., "The Art of Software Architecture, Design methods and Techniques", ISBN: 0471228869, Wiley Publishing Inc, 2003.

Allen R.A, 1997, "*A Formal Approach to Software Architecture*", Doctoral Dissertation, Department of Computer Science, Carnegie Mellon University, Pittsburgh PA, 1997.

Barabási, A., Ravasz, E., Vicsek, T., 2001, "*Deterministic scale-free networks*", Physica A 299 pp. 559-564, 2001.

Barabási, A., "*Linked – The New Science of Networks*", ISBN: 0738206679, Perseus Publishing, Cambridge, Massachusetts, 2002.

Barabási, A., Albert, R., Jeong, H., "*Scale-free characteristics of random networks: the topology of the world-wide web*", Elsevier Preprint August 6, 1999.

Barabási, A., "*Scale-Free Networks*", Scientific American, May 2003.

Barroca, L., Hall, J., "*Software Architectures, Advances and Applications*", ISBN: 1852336366, Springer-Verlag, London, 2000.

Bass, L., Clements, P. and Kazman, R., "*Software Architecture in Practice*", ISBN: 0201199300, Addision Wesley Longman, Inc. 1998.

Bengtsson, P., Bosch, J., "*Architecture Level Prediction of Software Maintenance*", Third European Conference on Software Maintenance and Reengineering, pp. 139, 1999.

Bennett, K., Rajlich, V., "*Software Maintenance and Evolution: A Roadmap*", The Future of Software Engineering , Anthony Finkelstein (Ed.), pp.75-87 ACM Press 2000.

Bianconi, G., Barabási, A, 2001, "*Bose-Einstein Condensation in Complex Networks*", Physical Review Letters, Volume 86, Number 24, pp. 5632-5635, June 2001.

Bigelow, J., "*Hypertext and CASE*", IEEE Software, Volume 5, Number 2, pp. 23-27, March 1988.

Boehm, B., "*A Spiral Model of Software Development and Enhancement*", IEEE Computer, May 1988, pp. 61-72

Bohner , S. A., "*Software Change Impact Analysis for Design Evolution*", Proc. 8th Int'l conf. on software Maintenance and Re-engineering, IEEE CS Press, Los Alamitos, Calif., pp. 292-301, 1991.

Bohner, S. A., "*A Graph Theoretic Approach to Software Change*", Ph.D Dissertation, Information Technology and Engineering, George Mason University, 1995

Bohner, S. A., Arnold, R. S. 1996, "*Software Change Impact Analysis*", ISBN: 0818673842, IEEE Computer society Press Los Alamitos, California, 1996.

Bohner, S. A., 1996b "*Impact Analysis in the Software Change Process: A Year 2000 Perspective*", Software Change Impact Analysis, pp43-51, IEEE Computer society Press Los Alamitos, California, 1996.

Bohner, S. A., Arnold, R. S. 1996c, "*An Introduction to Software Change Impact Analysis*", Software Change Impact Analysis, pp. 1-26, IEEE Computer society Press Los Alamitos, California, 1996.

Beizer, B., "*Black Box Testing: Techniques for Functional Testing of Software and Systems*", ISBN: 0471120944, John Wiley & Sons Ed., 1995.

Bollobás, B., "Degree sequences of random graphs", Discrete Math, Volume 33, pp. 1-19, 1981.

Bollobás, B., 1985, "*Random Graphs*", Academic Press, Inc. London-New York, 1985.

Booch, G, "*Object-Oriented Analysis and Design with Applications*", ISBN: 0805353402, 2nd edition, Addison-Wesley Professional, 1993.

Bouquet, E., Legeard, B., Peureux, F. and Torreborre, E., "*Mastering Test Generation from Smart Card Software Formal Models*", In Procs. Of the Int. Workshop on CASSIS'04, volume 3362, pp. 70-85, 2004.

Bouquet, F., Jaffuel, E., Legeard, B., Peureux, F., and Utting, M., "*Requirements Traceability in Automated Test Generation Application to Smart Card Software Validation*", ACM Software Engineering Notes, 30, 4, May 2005.

Brassard, M., Cardinal, M., "*Addressing Problems with Model Driven Architecture*", http://www.codagen.com/mda/article_developer_com.pdf, 2002.

Bratthall, L., Johansson, E., Regnell, B., "*Is a Design Rationale Vital when Predicting Change Impact? – A Controlled Experiment on Software Architecture Evolution*", PROFES 2000 - Second International Conference on Product Focused Software Process Improvement, Oulo, Finland, pp.126-139, 2000

Brooks, F. P., "*No Silver Bullet: Essence and Accidents of Software Engineering*", Coumputer, Vol. 20, No 4 , pp. 10-19, April 1987.

Buckley, J., Mens, T., Zenger, M., Rashid, A., and Kniesel, G., "Towards a Taxonomy of Software Change", Journal of Software Maintenance and Evolution: Research and Practice, 2005, v(17), pp. 309-332

Callaway, S., Newman, J., Strogatz, H. Watts, J., "*Network robustness and fragility: Percolation on random graphs*", Physical Review Letters. 85(25), pp. 5468-5471, 2000.

Cimitile, A., Lanuble, F., Visaggio, G., "*Traceability Based on Design Decisions*", Conf. On Software Maintenance, IEEE CS Press, pp. 309-317, Los Alamitos, Clif. 1992.

Colvin, R., "Behaviour Tree Syntax and Semantics", http://www.itee.uq.edu.au/~robert/DCCS/semantics/main.pdf, 2006

Conklin, J., 1987, "*Hypertext: An Introduction and Survey*", Computer, pp. 17-41, Sept. 1987.

Cover, T. M., Thomas, J. A., 1991, "*Elements of Information Theory*", ISBN: 0471062596, Wiley Series in Telecommunications, John Wiley & Sons, Inc., 1991.

Cruz, I. F., Tamassia, R., 1998, "*Graph Drawing Tutorial*", http://www.cs.brown.edu/people/rt/papers/gd-tutorial/gd-constraints.pdf.

Diestel, R., 1999, "*Graph Theory*", ISBN 3-540-26182-6, Second Edition, Springer, 1999.

Dijkstra, E. W., "*The Structure of the 'THE' Multiprogramming System*", Communications of the ACM 11, No. 5, pp. 341-346, 1968.

Dikel, D., M., Kane, D., Wilson, J., "*Software Architecture, Organizational Principles and Patterns*", ISBN: 0130290327, Prentice Hall PTR, 2001.

DOD 1985, U.S. Department of Defense, "*Military Standard for Software Quality Evaluation,*" DoD-Std-2168, Apr. 26, 1985.

Dromey, R.G., 1989, "*Program Derivation, the Development of Programs From Specifications*", Addison-Wesley Publishers Ltd., Sydney, 1989.

Dromey, R.G., 2003, "*From Requirements to Design : Formalising the Key Steps*", (Invited Keynote Address), IEEE International Conference on Software Engineering and Formal Methods, SEFM'2003, pp. 2-11, Brisbane, September, 2003.

Dromey, R.G., 2003b, "*Behavior Trees: Amplifying Our Ability to Deal with Requirements Complexity*", http://www.sqi.gu.edu.au/gse/papers/Dromey-LNCS-Final.pdf, 2003.

Dromey, R.G, Powell, D., "*Early Requirements Defects Dection*", TickIT International, pp. 3-13, 4Q05, 2005.

Dromey, G. R., "*Climbing Over the 'No Silver Bullet' Brick Wall*", IEEE Software, pp.96-98, March/April 2006.

Erdős and Rényi, "*On the Evolution of Random Graphs*", Publ. Math. Inst. Hungar. Acad. Sci. 5, pp. 17-61, 1960.

Eriksson, M., Morast, H., Börstler, J., "The PLUSS toolkit -- extending telelogic DOORS and IBM-rational rose to support product line use case modelling", Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, 2005, pp 300-304

Faloutsos, M., Faloutsos, P., Faloutsos, C., "On Power-Law Relationships of the Internet Topology", Computer Communication Review 29, 251, 1999.

Ferrante, J., Ottenstein, K. J., and Warren, J.D., 1987 "*The Program Dependence Graph and Its Use in Optimization,*" ACM Trans, Programming Languages and Systems, vol. 9, pp. 319-349, July 1987,

Fowler, M., Scott, K., 2000, "*UML Distilled A Brief Guide to the Standard Object Modeling Language*", ISBN: 020165783X, Addison-Wesley Publishers Ltd., 2000.

Garg, P. K., Scacchi, W., 1990, "*A Hypertext System to Manage Software Life-Cycle Documents*", IEEE Software, Vol. 7, No. 3, pp. 90-98, May 1990.

Garlan, D., Allen, R., and Ockerbloom, J., "*Exploiting Style in Architectural Design Environments*", Proceedings, 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), ACM Press, Vol 19 (5), pp. 175 -188, 1994.

Glass, R.L., "*Practical Programmer: Is This a Revolutionary Idea, or Not?*", Communications of the ACM. 47(11), pp. 23-25, 2004.

Gonzalez-Perez, C., Henderson-Sellers, B., Dromey, G., "*A Metamodel for the Behavior Trees Modelling Technique*", Third International Conference on Information Technology and Applications, ICITA 05, pp. 35-39, 2005.

Gotel, O.C., Finkelstein, A.C., "*An Analysis of the Requirements Traceability Problem*", Proc. First Conf. Requirements Eng., IEEE CS Press, pp. 94 – 101, Los Alamitos, Calif. 1994.

Grunske, L., Lindsay, P., Yatapanage, N., and Winter, K., "*An Automated Failure Mode and Effect Analysis Based on High-Level Design Specification with Behavior Trees*", the Fifth International Conference on Integrated Formal Methods (IFM'05), Lecture Notes in Computer Science, Vol. 3771, pp. 129-149, 2005.

GSE: Genetic Software Engineering: http://www.sqi.gu.edu.au/gse, 2005.

Halasz, F. G., "*Reflections on Notecards: Seven Issues for the Next-Generation Hypermedia systems*", Comm. ACM, 31(7), pp. 836-852, July 1988.

Hearnden, D., "*Software Evolution with the Model-Driven Architecture*", Phd. confirmation seminar. University of Queensland, Brisbane, 2004.

Hoare, C.A.R., "*Communicating Sequential Processes*", ISBN: 0131532898, Prentice-Hall, 1985.

Hofmeister, C., Nord, R., Soni, D, "*Applied Software Architecture*", ISBN: 0471958697, Addison-Wesley, 2000.

Horowitz, E., Williamson, R. C., 1986, "*SODOS: A Software Documentation Support Environment - Its Definition*", IEEE Trans. Software Eng., Vol. 12(8), pp. 849-859, 1986.

Horwitz, S., Reps, T., and Blinkley, D., 1990, "*Interprocedural Slicing Using Dependence Graphs*", Programming Languages and systems, Vol. 12(1), pp. 26-60, 1990.

IBM Rational Rose, 2007
http://www-306.ibm.com/software/awdtools/developer/rose/, 2007

IEEE, "*Glossary of Software Engineering Terminology*", Std. 729-1993, IEEE Software Eng. Standards, 5th ed., IEEE Press, New York, 1993.

Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G., "*Object-Oriented software engineering. A Use Case Approach*", ISBN: 0201544350, Addison Wesley, 1992.

Keables, J., Roberson, K., Mayrhauser, A., "*Data Flow Analysis and its Application to Software Maintenance*", Proc. Conf. On Software Maineenance, IEEE CS Press, pp. 335-347, Los Alamitos, Calif., 1988

Kingston, J. H., "*Algorithms and Data Structures*", 2nd edition, ISBN: 0201403749, Addison Wesley Longman Ltd, 1998.

Knuth, D. E., "*The Art of Computer Programming, Fundamental Algorithms*", 3rd edition, Vol 1, ISBN 0201896834, Addison Wesley Longman, 1997a.

Knuth, D. E., "*The Art of Computer Programming, Seminumerical Algorithms*", 3rd edition, Vol 2, ISBN: 0201896834, Addison Wesley Longman, 1997b.

Knuth, D. E., "*The Art of Computer Programming, Sorting and Searching*", 2nd edition, Vol 3, ISBN: 0201896834, Addison Wesley Longman, 1997c.

Kuhl, F., Weatherly, R., Dahmann, J., "*Creating Computer Simulation Systems, an Introduction to the High Level Architecture*", ISBN: 0130225118, Prentice Hall RTR, Upper Saddle River, NJ 07458, 1999.

Le Métayer, D., "*Describing Software Architecture Styles Using Graph Grammars*", IEEE Transactions on Software Engineering, Vol. 24 (7), pp. 521-553, July 1998.

Legeard, B., Peureux, P., and Utting, M., "*Controlling Test Case Explosion in Test Generation from B Formal Models*", International Journal of Software Testing, Verification, Reliability 14(2), pp. 81-103, 2004.

Lehman, M.M., "*Programs, Cities, Students, Limits to Growth?*", Inaugural Lecture, Vol. 9, pp. 211-229, 1974.

Lehman, M. M., "*Program Evolution*", ISBN 0124424414, Academic Press, London, 1985.

Lehman, M.M., "*FEAST/2 Final Report – Grant Number GR/M44101*", Dept. of Computing, Imperial College, Sept. 2001.

Lock.S, Rashid, A. Sawyer, P., Kotonya, G., "*Systematic Change Impact Determination in Complex Object Database Schemata*", ECOOP Workshop for PhD Students in OO Systems, pp. 31-40, 1999.

Lorin, H., "*Sorting and Sort Systems*", ISBN: 0201144530, the systems programming series, Addison-Wesley Publishing Company, 1975.

Loyall, J. P, Mathisen, S.A., "*Using Dependence Analysis to Support the Software Maintenance*", Conf. On Software Maintenance, IEEE CS Press, Los Alamitos, Calif., pp. 282-291, 1993.

Luckham, D. C., Augustin, L. M., Kenney, J. J., Veera, J., Bryan, D., Mann, W., 1995, "*Specification analysis of system architecture using Rapide*", IEEE Transactions on Software Engineering, Vol 21(4), pp. 336-355, 1995.

Manna, S., Mukherjee, G., Sen, P., "*Scale-free network on a vertical plane*", arXiv:cond-mat/0307137, v1, 7 Jul 2003a.

Manna, S., Kabakçioglu, G., "*Scale-free Network on Euclidean Space Optimized by Rewiring of Links*", arXiv:cond-mat/0302224, v2, 2 Apr 2003b.

Matlis, Jan, "*Scale-Free Networks*", ComputerWorld, http://www. computerworld.com /networkingtopics/networking/story/0,10801,75539,00.htm, November 4, 2002.

MDA, http://www.omg.org/mda/, 2006.

Medvidovic, N., "*On the Role of Middleware in Architecture-Based Software Development*", SEKE' 02, pp. 299-306, 2002.

Mills, H. D., 1971, "*Top-Down Programming in Large Systems*", ISBN: 0138221227, in Debugging Techniques in Large Systems, Prentice-Hall, 1971.

McClave, J. T., 1997, "*Statistics*", 7th edtion, Englewood, NJ, Prentice Hall, 1997.

McWhinney, W., 1997, "*Paths of Change, Strategic Choices for Organizations and Society*", ISBN: 0803939302, Rivised Edition, Sage Publications, 1997.

Milgram, S., "*The Small World Problem*", Physiology Today, Vol 2, pp: 60-67, 1967.

Miller, E., "*From Dependency to Autonomy, study in organization and change*", Free Association Books, 1993.

More, E., 1998, "*Managing Changes, Exploring State of the Art*", JAI Press Inc., 1998.

Moser, L. E., "*Data Dependency Graphs for Ada Programs*", IEEE Trans. Software Eng., Vol. 16, No. 5, pp. 498-509, May 1990.

Naumovich, G., Avrunin, G.S., Clarke, L.A., Osterweil, L.J., "*Applying Static Analysis to Software Architectures*", Proceedings of the 6th European Software Engineering Conference, pp. 77-93, 1997.

OMG "*OMG Members and Industry Analysts Support the MDA*", http://www.omg.org/mda/mda_files/Member_and_Analyst_Quotes.pdf, 2001.

ORMSC., "*Model Driven Architecture*", Architecture Board ORMSC, htgp://cgi. org. org/docs/orrsc/01-07-01 .pdf, 2001.

Parr, S., Keith-Magee, R., "*The Next Step – Applying the Model Driven Architecture to HLA*", http://members.iinet.net.au/~freakboy/papers/03S-SIW-123.pdf, 2004

Perry, D. E., "*The Inscape Environment*", Eleventh Inter. Conf. on Software Engineering, Pittsburgh, PA, IEEE Computer Society Press, pp. 2-12, 1989.

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

Poole, J.D., 2001, "*Model-Driven Architecture: Vision, Standards and Emerging Technologies*", Workshop on Metamodeling and Adaptive Object Models, http://www.cwmforum.org, 2001.

Podgurski, A. and Clarke, L. A., "*A Formal Model of Program Dependencies and Its Implications for Software Testing, Debugging and Maintenance*", IEEE Trans, Software Eng., Vol: 16(9), pp. 965-979, 1990.

Rajlich, V., "*Software Change and Evolution*", SOFSEM'99, LNCS 1725, pp.189-202, 1999

Redner, R., "*How popular is your paper? An empirical study of the citation distribution*", the European Physical Journal B, Vol: 4, pp. 131-134, 1998.

Rice, A., "*The Enterprise and its Environment*", Learning for leadership: interpersonal and intergroup relations. London: Tavistock Publications, 1963.

Royce, W., "*Software Project Management, A Unified Framework*", ISBN: 0201309580, Addison Wesley, 1998.

Rumbaugh, J., Blaha, M., Permerlani, W., Eddy, F., Lorensen, W., "*Object-Oriented Modeling and Design*", ISBN: 0-13630064 -5, Pretice Hall, 1991.

Schneider, S., "*The B-Method – An Introduction*", ISBN: 033379284X, Palgrave Editor, 2001.

Sedgewick, R., "*Algorithms*", 2nd edition, ISBN: 0201066734, Addison-Wesley Publishing Company, Inc, 1988.

Shaw, M., Deline, R., Klein, D., V., Ross, T., L., Young, D., M., Zelesnik, G., "*Abstractions for Software Architecture and Tools to Support Them*", IEEE Transactions on Software Engineering, Vol: 21(4), pp. 314-335, 1995.

Shaw, M., Garlan, D. "*Software Architecture, perspectives on an emerging discipline*", ISBN: 0131829572, Prentice Hall, Upper Saddle River, New Jersey, 1996.

Shaw, M., Clements, P., "*A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems*", Proceedings, 21st International Computer Society Press, pp. 6-13, 1997.

Shlaer, S., mellor, S.J., "*Structured Development for Real-Time Systems*", ISBN: 0138547955, Vols. 1-3, Yourdon Press, 1985.

Shlaer, S., Mellor, S.J., "*Object Lifecycles: : Modeling the World in States*", ISBN: 0136299407, Yourdon Press, New Jersey, 1992.

Siegel, J., "*Making the Case: OMG's Model Driven Architecture*", Software Development Times (www.sdtimes.com), 15 Oct, 2001.

Smith, C., Winter, K., Hayes, I., Dromey, R.G., Lindsay, P., Carrington, D, "*An Environment for Building a system Out of Its Requirements*", 19th IEEE International Conference on Automated Software Engineering, Linz, Australia, pp. 398-399, Sept 2004.

Sommerville, I., "*Software Engineering*", 7th Edition, ISBN: 0321210263, Addison Wesley, 2004.

Stafford, J. A., Wolf, A. L., "*Software Architecture*", Component-Based Software Engineering, putting the pieces together, Chapter 20, ISBN: 0201704854, Addison-Wesley Professional, pp. 371-388, 2001.

Suydam, W., "*CASE Makes Strides towards Automated Software Development*," Computer Design, pp.49-70, January 1, 1987.

Svetinovic, D., Godfrey, M., "*Software and Biological Evolution: Some Common Principles, Mechanisms, and a Definition*", http://plg.uwaterloo.ca/~migod/papers/iwpse05.pdf, 2005.

Szyperski, C., "*Component Software, Beyond Object-Oriented Programming*", ISBN: 0201178885, Addison Wesley, 1999.

Tolk, A., "*Avoiding another Green Elephant – A Proposal for the Next Generation HLA based of the Model Driven Architecture*", Processdings of the 2002 Fall Simulation Interoperability Workshop, 2002.

Tool Download, 2006a, "*Genetic Software Engineering Toolkit (GSET)*", http://www.sqi.gu.edu.au/gse/tools/gset.html, 2006.

Tool Download, 2006b, "*Component Architecture and Scale-Free Networks*", http://www.sqi.gu.edu.au/gse/tools/classnet.html, 2006.

Tool Download, 2006c, "*A conjecture on Sorting Algorithms and the Scale-Free Network Property*", http://www.scs.carleton.ca/~cgm/scale-free/, 2006.

Trigg, R. H., and Weiser, M. "*Textnet: A Network-Based Approach to Text Handling*", ACM Trans. Office Information Systems, pp. 1-23, Jan. 1986.

Vancleemput, W. M., Linders, J. G., "*An improved graph-theoretic model for the circuit layout problem*", Proceedings of 11th workshop on Design Automation, pp. 82-90, 1974.

W3C's official XML web site: http://www.w3.org/XML/

Wadler, P. and Weihe, K., "*Component-based programming under different paradigms*", Technical report, Report on the Dagstuhl Seminar 99081, February 1999.

Weiser, M., "*Program Slicing*", IEEE Trans. Software Engineering Vol 10(4) pp. 352-357, July 1984.

Wen, L., Dromey, R.G., "*Architecture Normalization – A way to Simplify Software Architecture*", PhD confirmation report, Griffith University, 2003.

Wen, L., Dromey, R.G., "*From Requirement Change to Design Change*", 2nd IEEE International Conference on Software Engineering and Formal Methods, pp.104-113, 2004

Wen, L., Dromey, R.G., "*Architecture Normalization for Component-Based Systems*", Proceedings of the International Workshop on Formal Aspects of Component Software 2005 pp: 247-262, this paper will also be published in ENTCS.

Wen, L., Colvin, R., Lin, K., Seagrott, J., Yatapanage, N., Dromey, G., "*'Integrare', a Collaborative Environment for Behavior-Oriented Design*", in Proceedings of the Fourth International Conference on Cooperative Design, Visualization and Engineering, LNCS 4674, pp. 122-131, 2007a.

Wen, L., Kirk, D. , Dromey G., "*Software Systems as Complex Networks*", in Proceedings of The 6th IEEE International Conference on Cognitive Informatics, IEEE CS Press., 2007b

Wen, L., Kirk, D. , Dromey G., "*A Tool to Visualize Behavior and Design Evolution*", in Proceedings of The International Workshop on Principles of Software Evolution (IWPSE2007), 2007c

Winter, K., "*Formalising behaviour trees with CSP*", Integrated Formal Methods, LNCS, Vol. 2999, pp. 148-167, April 2004.

Witt, B. I., Baker, F. T., Merritt, E. W., "*Software Architecture and Design, Principles, Models, and Methods*", ISBN: 1850328455, Van Nostrand Reinhold, 1994.

Woolsfon, A., "*Life without Genes: Genetic Toys and the Information Zoo*", ISBN: 0006548741, Adrian Publisher Flamingo, 2000.

Yan, S. S., Collofello, J. S., MacGregor, T., "*Ripple Effect Analysis of Software Maintenance*", Proc. Compsac, IEEE Computer Society Press, IEEE Computer Society Press, Los Alamitos, CA, pp. 60-65, 1978.

Zafar, S., Dromey, R. G., "*Integrating Safety and Security Requirements into Design of an Emedded System*", Asia-Pacific Software Engineering Conference (APSEC'05), pp. 629-636, 2005.

Zhao, J., Yang, H., Xiang, L., Xu, B., "*Change impact analysis to support architectural evolution*", Journal of Software Maintenance and Evolution: Research and Practice, Vol:14, pp. 317-333, 2002.

Zhao, J., "*Change Impact Analysis for Aspect-Oriented Software Evolution*", International Workshop on Principles of Software Evolution, IWPSE, pp. 108-112, 2002.

Zheng, X., Dromey, R.G., "*Making Requirements Defect Detection Repeatable*", http://www.sqi.gu.edu.au/docs/sqi/gse/XZ-RGD-ICRE-2003.pdf, 2003