

On Navigation and Analysis of Software Architecture Evolution

by

Qiang Tu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada 2002

©Qiang Tu, 2002

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Qiang Tu

I authorize the University of Waterloo to reproduce this thesis by photocopying or other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Qiang Tu

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Acknowledgements

I would like to thank many people without whose support this thesis could not be possible. First of all, I must thank my advisor, Michael W. Godfrey for his vision, encouragement, and support throughout the preparation and writing for this thesis. I am also thankful to the thoughtful comments and insightful feedbacks from my thesis readers, Ric Holt and Kostas Kontogiannis.

I would wish to thank members of the SWAG lab who developed the wonderful PBS tools on which my thesis work is based upon. They also provides excellent ideals and comments on my work. In particular, many thanks go to John Tran, Thomas Parry, Ahmed Hassan, Eric Lee, Davor Svetinovic and Igor Ivkovic.

And finally I would like to thank my dear father Xiang Zheng Tu and mother Yun Yan Li, brother Peng Tu, girl friend Hin-Chung Yuen, friends Joon and Michael, and everybody who cares about me for their encouragement and supports throughout the process.

Abstract

Software systems must evolve during their lifetime in response to changing expectations and environments. As the software evolves, the system becomes harder to understand and maintain without the proper knowledge about how the system had changed in the past and the context of those changes. Studying software evolution has been extraordinarily costly and time consuming, as it lacks a sound theory, effective research approaches, as well as an integrated research environment.

In this thesis, we present a research approach for studying software evolution, which incorporates evolution metrics, visualization of system change history, and a method of relating similar program entities between different releases in spite of changed name or location. To validate our approach, we have implemented a prototype research environment called *BEAGLE* to aid the software maintainer to understand how large software systems have evolved overtime. *BEAGLE* integrates data from various statistic tools and metric tools, and provides a query engine as well as a web-based visualization and navigation interface. *BEAGLE* aims to provide help in understanding the long-term evolution of systems that have undergone architectural and structural changes.

We performed a case study on the evolution of GNU Compiler Collection (GCC) using a prototype implementation of *BEAGLE*. We were able to discover several interesting evolution characteristics of GCC, and to answer specific questions related to the evolution history of GCC, such as the relationship between the experimental EGCS release and traditional GCC release, and the different evolution patterns shown by GCC releases at different periods of time.

Contents

1	Software Evolution: An Introduction	1
1.1	Overview of Thesis	1
1.2	Biological Evolution and Software Evolution	2
1.2.1	Biological Evolution	2
1.2.2	Software Evolution	4
1.2.3	Software Evolution and Software Maintenance	5
1.3	Description of Research Problems	6
1.4	Our Approach To the Problems	8
1.4.1	Software Evolution Browser	9
1.4.2	Analysis of Architectural Evolution	12
1.5	Major Contributions	14
1.6	Organization of Thesis	15
2	Related Research On Software Evolution	16
2.1	Software Evolution: A Discipline of Software Engineering	16
2.1.1	Lehman: Laws of Software Evolution	17
2.1.2	Perry: Dimensions of Software Evolution Environment	18
2.2	Software Evolution Metrics	19
2.2.1	Evolution Metrics From Program Source Information	20
2.2.2	Metrics From Version Control Management Information	23
2.2.3	Comparison of Evolution Metrics	27
2.3	Visualization of Software Evolution	27
2.3.1	GASE Tool and KAC System	29

2.3.2	Gall: Software Evolution in Color and 3-D	31
2.3.3	Comparison of Evolution Visualization Techniques	32
2.4	Empirical Studies of Software Evolution	33
2.4.1	Review of Software Evolution Empirical Studies	35
2.4.2	Conclusion of Related Work	37
3	BEAGLE: An Integrated Platform for Studying Software Evolution	39
3.1	Challenges to Software Evolution Research	40
3.2	Discussion of Methodologies	40
3.2.1	History Data Repository	41
3.2.2	Navigation of Evolution Information	45
3.2.3	Analysis of Software Structural Changes	47
3.3	BEAGLE: An Integrated Environment	54
3.3.1	Database Tier	56
3.3.2	Application Logic Tier	67
3.3.3	User Tier	71
3.4	Conclusion	72
4	Architectural Evolution of GCC:A Case Study	73
4.1	Background and History of GCC Project	73
4.1.1	Origin of GCC	73
4.1.2	GCC 2.0 and Cygnus	74
4.1.3	EGCS and Web-based Software Development	75
4.2	Common Software Architecture of GCC Releases	77
4.2.1	Reference Architecture of Compilers	77
4.2.2	GCC Conceptual Architecture	78
4.2.3	Concrete Architecture of GCC	81
4.3	Related Research Work on GCC	83
4.3.1	GCC System Size Growth	83
4.3.2	GCC Build-Time Behaviors	85
4.3.3	Dominance Tree Analysis of GCC Evolution	88
4.4	Elaboration of Research Questions On GCC Evolution	88

4.4.1	From GCC 1.0 To GCC 2.0	90
4.4.2	From GCC 2.x To EGCS 1.x	97
4.4.3	Stable Releases vs. Development Releases	103
4.4.4	Build Configurations and GCC Architecture	112
4.4.5	Refactoring and Rearchitecting	115
4.4.6	Distribution of Evolution Effort	118
4.5	Summary of Observations	125
5	Summary and Future Work	127
5.1	Summary	127
5.2	Future Work	129

List of Tables

2.1	Research on Software Evolution Metrics	28
3.1	Release archives of three open source projects	43
4.1	Origin analysis results on EGCS 1.0 — parser	102
4.2	Origin analysis results on EGCS 1.0 — code generator	103
4.3	Change history of file <code>gcc/combine.c</code>	108
4.4	Change history of function <code>add_method</code>	110
4.5	Bertillonage analysis on function <code>assign_parms</code>	115
4.6	Callee analysis on function <code>assign_parms</code>	117
4.7	Call analysis on file <code>enquire.c</code>	118

List of Figures

1.1	Data Repository and Query Interface of BEAGLE	10
1.2	Software Structural Changes with Similar Naming Scheme	13
1.3	Software Structural Changes with Different Naming Scheme	14
2.1	Screenshot of KAC System	30
2.2	Screenshot of 3-D Evolution Diagrams	31
2.3	Screenshot of 2-D Evolution Diagrams	32
3.1	Example of Origin Analysis	48
3.2	Example of Call-Relation Change Analysis	53
3.3	Conceptual Architecture of BEAGLE Environment	55
3.4	Schema of BEAGLE Data Repository	58
3.5	Procedures to Build Data Repository	61
3.6	Result of the first example query	64
3.7	Result of the second example query	67
3.8	Screen Shot of BEAGLE Architecture Comparison: GCC 2.0 vs GCC 2.7.2	68
3.9	User Interface for Entering Query Options	71
4.1	Components and Phases of Compiler	78
4.2	Conceptual Architecture of GCC and its Components	79
4.3	Conceptual Architecture of Language Compiler	80
4.4	Concrete Architecture of GCC	81
4.5	Concrete Architecture of Language Compiler - Call Relation	82
4.6	Concrete Architecture of Language Compiler - Data Reference	82

4.7	System Growth of GCC Releases	84
4.8	GCC Bootstrapping Build	85
4.9	GCC Build-Time Code Generation - Generated Code	86
4.10	GCC Build-Time Code Generation - Code Generation Procedure	87
4.11	Architecture of GCC 2.0 Comparing to GCC 1.39 - Selection Screen	92
4.12	Architecture Comparison of GCC 2.0 and 1.39 - Top Subsystems	93
4.13	Comparison of GCC 2.0 and 1.39 - Parser	95
4.14	Comparison of GCC 2.0 and 1.39 - RTL Generator	96
4.15	Comparison of GCC 2.0 and 1.39 - Code Generator	98
4.16	Comparison of EGCS 1.0 and GCC 2.7.2.3	99
4.17	Origin Analysis on EGCS 1.0	101
4.18	Evolution of EGCS Releases - Make Query	104
4.19	Evolution of EGCS Releases - Code Generator and Optimizer	105
4.20	Evolution of EGCS Releases - Parser	106
4.21	Evolution of GCC 2.x Releases - Code Generator and Optimizer	109
4.22	Evolution of GCC 2.x Releases - Parser	111
4.23	Compare C-Only and All Build Configuration - Parser	113
4.24	Compare C-Only and All Build Configuration - Optimizer	114
4.25	File Level Origin Analysis Example One	116
4.26	File Level Origin Analysis Example Two	119
4.27	Distribution of Code Distance Across Subsystems - GCC 2.0	121
4.28	Distribution of Code Distance Across Subsystems - GCC 2.8.0	122
4.29	Distribution of Code Distance Across Releases - Code Generator	124

Chapter 1

Software Evolution: An Introduction

1.1 Overview of Thesis

Software systems must evolve during their lifetime in response to changing expectations and environments. The context of a software system is a dynamic multi-dimensional environment that includes the application domain, the developers' experience, as well as software development processes and technologies [41]. Software systems live in an environment that is very complex and dynamic.

While change is inevitable in software systems, it is also risky and expensive, as careless changes can easily bring down the whole system. It is a challenging task for developers and maintainers to keep the software evolving, while still maintaining the overall stability and coherence of the system. To achieve this goal, software engineers have to learn from history. By studying how successfully maintained software systems has evolved in the past, researchers can find answers to questions such as “why and when changes are made”, “how changes should be managed”, and “what the consequences and implications of changes are to continue software development”. *Software Evolution*, one of the emerging disciplines of software engineering, studies the history of software systems, explores the underlying mechanisms that affect software changes, and provides guidelines for better software evolution processes.

Lehman suggested that the software evolution is a feedback system where complex interaction and feedback control exists among software systems, development processes and

the application environment [32]. To understand this complex mechanism, we can start by studying the evolution patterns of software systems from artifacts such as program source code, comments, and design documentation. Then we can relate the “evolution patterns” discovered in software systems with those discovered in the development process and the surrounding environment. Eventually, we can discover the underlying mechanisms that decide the evolution of software systems.

To discover the evolution pattern of past software systems, we need practical browsing and analysis tools that can guide users in navigating through software evolutionary histories. Browsing tools help us to visualize *what* changes had happened to the software system in the past. Analysis tools can aid in discovering “undocumented” changes, assisting us to find out *why* such changes happened.

In this thesis, we will describe an approach towards efficient navigation and visualization of evolution histories of software architectures. Furthermore, we will also introduce several methods to track and analyze the software structural changes from past releases. Finally, the evolution history of a real-world software system, the GCC compiler suite, will be used as a case study to demonstrate the effectiveness of our approach.

1.2 Biological Evolution and Software Evolution

1.2.1 Biological Evolution

In the natural world, living organisms may alter their characteristics over time, and the traits that record the changes are passed from one generation to the next. The study of biological evolution attempts to understand the forces that have caused ancient organisms to evolve into the great variety of life forms that exists today. It also addresses how species branch off into entirely new species, and how different species may be related through family trees.

Similar to living organisms, a software system also evolves to adapt to its changing environment by releasing new versions, with enhanced features and improved quality. Robertson [45] distinguished the similarities and differences between biological evolution theory and the concepts of software evolution. In that study, he summarized the following char-

acteristics that dominate biological evolution:

- In a population of given species, every individual exhibits a unique set of attributes. A variety of attributes are carried by all the individuals in the population.
- Specific attributes may benefit individuals, to allow them to live longer and stronger under a given environment. Thus the individual that carries such attributes will be in favor for reproduction.
- The offspring of successful individuals will inherit a significant portion of the attributes from their ancestors.
- Species survive by continuing evolution to keep up with the changes of their environment.

Robertson then discussed two common misunderstandings about biological evolution. The first misunderstanding is that changes are considered same as evolution. Changes of individuals that are directly caused by the environment are not evolution: evolution is driven by the new permutations of DNA sequence in the specie's gene, and these new DNA permutations are prorogated to future generations via reproduction; on the other hand, individual changes do not cause the entire species to change their DNA sequence, thus they cannot be prorogated as individuals of future generation cannot inherit them. For example, there are frogs found to have three legs as the result of pesticide overuse. This type of change is limited to a small group of individuals and not inheritable by the whole species, thus it is not an evolutionary phenomenon. Evolutionary changes are those can be inherited by offspring via genes. The whole species evolves, as nature selects specific groups of individuals that carry critical genes to survive and reproduce. These critical genes make them better adapted to the environment. For example, germs that cause human to develop flu all start to carry special genes that made many anti-biotic drugs less effective towards them.

The second misunderstanding is that biological evolution is all about organisms changing their forms from simple to complex, and the more “complex” they become, the “better” evolution it is. Species are only considered “success” in biological evolution term if they can adapt to the current environment and reproduce enough offspring. Successful species must

continuously evolve and quickly enough to keep up with the changes of the surrounding environment.

1.2.2 Software Evolution

It was suggested by Lehman that software systems evolve in a manner similar to that of living organisms [31]. Software systems have to keep evolving in order to adapt to the changing environment, *i.e.*, different requirement, business process, and supporting technologies.

The study of software evolution covers all aspects related to long-term software change, especially systemic changes that exhibit repetitive patterns. Software evolution study also tries to discover the relationship between the changes of software development process, program code, and software maintainability. The purpose of studying software evolution is to understand the underlying mechanisms that decide how software system evolves, so that software developers can adopt more effective development practices and guidelines that make software systems evolve quickly enough to keep up with the changing environment, while maintaining system stability and low maintenance cost.

There are two major research directions in the software evolution study. One direction is to extend source control systems to support fast and safe code changes. Research work on configuration management systems that have built-in supports for software evolution include [29] [35] [55]. Code features that assist software evolution include program invariants and function clones. For example, Ernst *et al.* [14] developed a technique to dynamically discover invariants of program properties, so that software developers who are working on the new release will not break the system by accidentally violating exiting assumptions that have to be preserved. Lagüe *et al.* [30] introduced function clone detection techniques into the software development process with Datrix tools. When software developers apply quick changes to the program source code, they tend to copy chunks of code from existing code base to implement similar functionality. However, improper code cloning will degrade system maintainability as it introduces implicit dependencies between code sections from different modules. If the original segment of the code need to be modified later, without explicit documentation, maintainers usually forget to apply similar changes that are necessary to all the clones scattered throughout the system. By applying clone detection in the

development process, such undesirable clones will be found and removed from the system, or replaced by more appropriate techniques such as template in C++ and interface in Java.

A second direction focuses on the relationship between software development process, source code metrics, and software maintainability. Some studies analyze the growth patterns exhibited by the software system, and then associate these growth patterns with causing factors found in the development process [32, 31, 20, 17, 56, 36, 33]. Other studies try to build predictive models that use the metrics measured from development process or source code to predict the future software maintainability and defects [43, 40, 10, 21, 27, 37, 13, 49, 5, 38].

Some researchers attempt to build a theoretical foundation for software evolution research. Lehman believes the evolution of software system is driven by multiple feedback loops between software development and usage of the system [32]. Perry examined the surrounding environment of software system, and list three dimensions that will influence the way software evolves [41]. The three dimensions of software context are the application domains, developers' experience with the system, and the development techniques and process. Any changes in these dimensions will affect the evolution of software system.

1.2.3 Software Evolution and Software Maintenance

When researchers discuss software evolution, they sometimes unintentionally equate the term with “software maintenance”. These two concepts are very closely related, and there are not yet standard definitions that can clearly distinguish them. However, from our research experiences, we believe there are subtle differences between them, and we need to clarify them at the beginning of this thesis.

Software maintenance deals with correcting program defects, adapting to new requirements, and enhancing software functionalities. It mainly consists of planned changes made to the software system as the results of explicit maintenance requirements. The software maintenance process is usually carefully planned and closely monitored.

Software evolution, on the other hand, concerns about what actually has happened to the software system over a long period of time, especially change patterns exhibited by the whole system, as well as its software architecture. It also emphasize the dynamic interaction, including *mutual selection*, between software system and its environment.

In summary, *software maintenance* tries to plan ahead and take control of the change process. On the other hand, *software evolution* examines the actual changes made to the system in the history, both planned and unexpected, and studies the relations between observed history events and environmental factors.

1.3 Description of Research Problems

The surprising truth about legacy software systems is that their lifespan is commonly much longer than the developers had originally imagined. Many systems that had to be fixed for Year 2000 problems were written in COBOL or FORTRAN, and some are even older than the people who are trying to fix them. Users are reluctant to replace existing software systems that are proven to be reliable with complete new systems that are yet thoroughly tested. The alternative is to develop new versions of the system that are based on the existing infrastructure and proven technologies. New features and bug fixes are introduced into the system gradually without disturbing normal business operation. Software tends to evolve gradually, rather than by revolution.

To understand how software systems evolve, empirical study on large systems with long historical releases has been proven to be an effective research method. Observing the evolution patterns and dynamic behaviors exhibited by software systems over a long period of time helps researchers to discover the fundamental mechanisms that influence or shape the way software evolves.

Related research works on software evolution have developed many metrics, empirical study methodologies, and assisting tools with data interpretation and visualization capacities to aid the understanding of the vast amount of empirical data archives. However, despite this progress, software evolution remains a challenging research area for several reasons:

- The complexity involved in collecting and analyzing the history data that records how software systems had changed in the past is enormous. Many commercial software systems have lived for decades, with many versions released during their life. It requires more resources and different methodologies to understand the entire change

history of a software system, whereas traditional software comprehension usually focuses only on one release.

- There are few empirical studies that meet the depth and breadth requirements to make generalized conclusions on the underlying mechanisms of how a software system evolves and why. Some studies emphasized on the evolution of operating systems and system software [20, 10], others focused on applications in telecommunication domain [17, 36, 37]. However, much more empirical study is needed to cover other application domains before we can generalize the existing findings.
- Empirical data on software evolution is currently interpreted heuristically, without sound theoretical support. After the proposal of “law of software evolution” almost two decades ago [31], little progress has been made on the theory of software evolution.

As Integrated Development Environment (IDE) and CASE tools have approved, to increase the productivity of developers to design and implement software systems, fully automated tools and powerful supporting environment are essential to overcome the complexity and high cost associated with collecting and analyzing empirical data on software evolution.

Here we list some of the general requirements for such supporting tools and environment that we felt would improve the productivity and lower the cost to conduct empirical study on software evolution:

1. Similar to the fact that software architecture has multiple views, researchers study the empirical data of software evolution history from different perspectives, depending on their individual interest. The supporting tools and environments must be flexible and extensible in functionality to satisfy the diversity of needs.
2. Software evolution analysis tools should be able to “zoom-in” to the low-level details such as the change history of individual function definitions or data references between modules, while also be able to “zoom-out” to the higher level to get the “big picture” of the software evolution history. For example, the distribution of changes

among different subsystems tells us which parts of the software system have shown the most changes, and which parts have the most stable structure.

3. The supporting environment for software evolution research should provide powerful navigation and visualization capabilities so that user can find interesting patterns or phenomena from the vast amount of history data quickly.
4. Tools that automatically collect, analyze, and present software evolution metrics are essential to help us quantify changes made to the software system in the history. Then we can use many numerical methods to analyze the quantified changes and to discover evolution patterns.
5. When software evolves, its architecture must also changes to reflect changed functional and non-functional requirements. Architectural evolution study focuses on the change characteristics of software architecture, and its relations with other aspects of software evolution. We need tools and supporting environments that can assist us to collect and analyze architectural change information.

1.4 Our Approach To the Problems

In this thesis, we will introduce our approaches to these problems associated with software evolution research. Our solution involves an integrated environment, which incorporates a data repository, several automated tools and a web-based user interface into one system. Its features include collecting and storing archived software history information, detecting change patterns by applying evolution analysis algorithms, providing navigation and comparison facility for researcher to study software architecture changes, and also sharing all the information with other researchers or automatic tools over the Internet.

Our software evolution study environment is named *BEAGLE*, after the British naval vessel on which Charles Darwin served as a naturalist for an around-the-world voyage. During that historical voyage, Darwin collected many specimens and made some valuable observations, which eventually provided him the essential materials to develop the theory of evolution by natural selection. We expect our tools and the BEAGLE environment could help researchers conduct more efficient empirical study on software evolution with

lower cost, so that more valuable observations about software evolution could be made towards the creation of software evolution theory. The BEAGLE environment includes three major components: architecture comparison, history visualization and navigation, and origin analysis. Each of them provides a valuable functionality to assist researchers observe and analyze the software change history. We will discuss further details of these three components in the following sections.

1.4.1 Software Evolution Browser

Studying software evolution requires quick and convenient access to the enormous amount of archived data that records all the adjustments made to the software system during its entire life. In addition to quick access to history information, researchers also need to be able to jump between releases to compare the characteristics of particular program component at the different times during the program's life. This requires the supporting tools to provide a flexible navigation interface, and to be able to compare many aspects of the software systems across many releases.

In BEAGLE, archived evolution information is stored in a central repository. There are three types of history data stored in BEAGLE: architecture information extracted from source code, program code metrics, and development activity information collected from other sources, such as release notes, revision control system data, and design documents. The same types of information are collected and stored for every version of the software system that was previously released. The BEAGLE repository provides valuable first-hand research material, from which we can observe interesting evolution patterns and make conclusions for software evolution theory. The repository is also equipped with a standard access and query interface. Evolution navigation and analysis tools can easily retrieve relevant information from the repository through this interface, as shown in figure 1.1.

We have designed a browser-based navigation interface for BEAGLE so that researchers can easily explore and compare the different characteristics exhibited by different software releases. Primarily, the user needs to select the past releases that he is interested, and choose a reference release on which all the comparisons are based. BEAGLE will then process the query requests by executing related engine to complete the software comparison. During the processing, original evolution data is retrieved from the repository for analysis

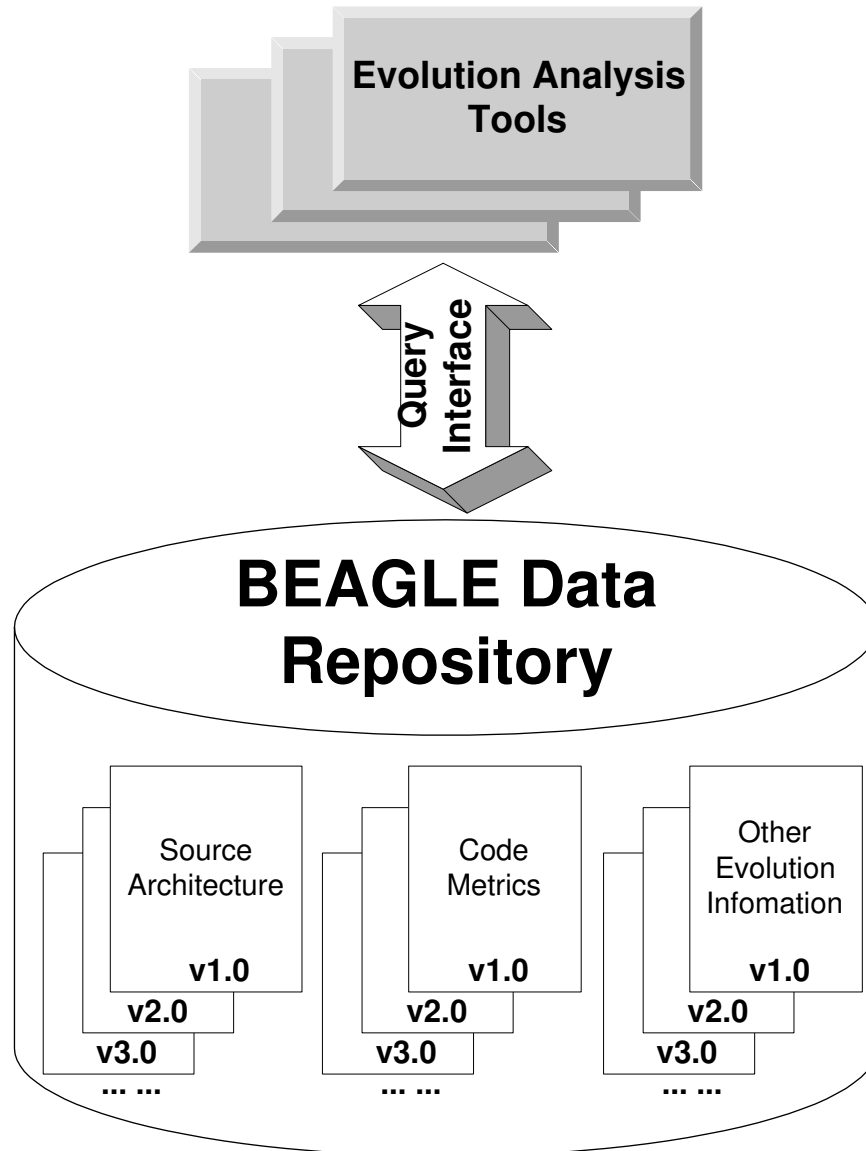


Figure 1.1: Data Repository and Query Interface of BEAGLE

through the access/query interface provided by the repository.

The result returned from the BEAGLE comparison engine has three parts. The software structure of the reference release is displayed as an expandable tree. The branches and leafs of this structure tree represent program entities at different levels, such as subsystems, files, and functions. A color schema is used to represent the change status of each program entity. For example, red color means the program entity is new to system, with respect to the other releases in the comparison.

If the user is interested to investigate evolution history of individual program entities in greater detail, he can click on the corresponding branch or leaf in the structure tree. The first part is a text table that displays the change history of the select program entity in terms of code metrics. For functions, we list seven of its code metric values in all the releases that participate in the comparison, including lines of code, lines of comment, cyclomatic metrics, s-complexity, d-complexity, Albrecht and Kafura metric [28]. For program files, we collect six metric values, including lines of code, lines of comment, number of functions defined, average cyclomatic metric value, average fan-out and maintenance index. This table tells us the evolution history of a particular program entity from a numerical aspect.

The last component of the result is a visualization diagram that displays the software architecture landscape for the selected program entity. This landscape diagram shows the contained modules (files) within a program subsystem, along with the relations between these files, “supplier” subsystems, and “consumer” subsystems. The same color schema is used to represent the evolution status of files and relations as in the program structure tree. Users are provided with a rich set of query tools so that user can “zoom-in” to a second-level subsystem that is contained in current subsystem, or “jump” to another subsystem that has dependency relations with current subsystem. Users can also select viewing criteria such as what kind of relations will be shown. For example, options can be set to only display new relations introduced in the reference release.

The philosophy behind BEAGLE’s evolution browser interface is to display as much information as we have collected about the selected program entity under study in a well-organized way. The feedback contains text description, numerical tables, and visual representation. Users can study the information from many angles and aspects. At the same time, users can also study relations between the main program entity and other program

entities in the system of the same release, or the same entity from other releases, through the “hyper link” style of navigation interface provided by BEAGLE.

1.4.2 Analysis of Architectural Evolution

Software architecture of a software system is the structure of the system, which comprises software components, the externally visible properties of those components, and the relationships among them [4]. Software architecture has multiple views. In this thesis, we will discuss mainly about the code view, which focus the structures and relationships between source code components, including subsystems, modules or files, and functions.

Software architecture concerns the behaviors and interactions of high-level components, and is developed as the first step of the software development process to achieve a collection of desired functional and non-functional properties. As the environment of software system changes, the desired functional and non-functional properties of the system also change. As the result, the software architecture has to continue evolving as well.

The code view of software architecture can be treated as a finite, directed graph that allows multiple edges between a pair of vertexes with different semantics [8]. For example, there should be two different paths between two subsystems, one for call dependency, and the other for data reference dependency. When software architecture evolves from release to release, its graph topology also changes to reflect the structural and dependency changes of code components. Because software architecture could be abstracted as a graph, we can also model the evolution of software architecture as the *morphing* of its graph, as shown in figure 1.2. In these graphs, nodes represent program modules, and edges between a pair of nodes model the various dependencies between program modules. The topology of G2, which models the software architecture of a more recent release, is different from that of G1. However, since the vertex names in both graphs have very similar scheme, it is very easy to tell that vertex v2 is no longer in the graph, and there is a new vertex v6 introduced in G2.

When a graph G1 morphs into G2, if all the vertexes are labelled under the same naming scheme, we can easily identify those vertexes in G2 that were originally in G1, and those that are newly introduced into the graph by G2. We can use this technique to identify new source components introduced in the newer software architecture. However, if a new

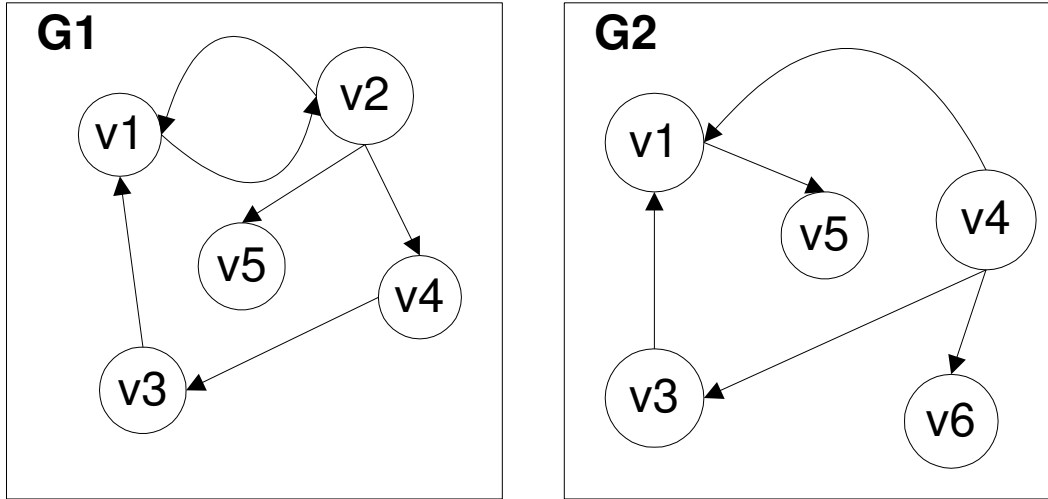


Figure 1.2: Software Structural Changes with Similar Naming Scheme

naming scheme is used in the newer release, we lose the traces between the vertexes in the new graph and those in the old graph. This situation usually happens when the software architecture of a new release has to be modified to address major changes in requirements or implementing technologies. Other activities such as the re-architecting of the older software architecture for easier maintenance could also cause this type of changes. Figure 1.3 shows two graphs, one models the software architecture found in original release, and the other with totally different vertex names models the software architecture of the later releases after a major re-architect effort. From the new graph G2, we can observe that some vertexes are clustered into subsystems ss1 and ss2, and all the vertexes are now named after letters, instead of numbers in G1.

We need more sophisticated techniques other than straightforward name comparison to relate vertexes in the newer graph with those in the old graphs. In this thesis, we will introduce two algorithms that relate vertexes in one graph with vertexes in another graph. We call this technique “Origin Analysis”, as it help us to find out the “origin” of new source components in the new software architecture. One algorithm compares the feature set of a vertex, which represents a function or file in the modelled software architecture, with all the vertexes in another graph, and attempts to find the most similar

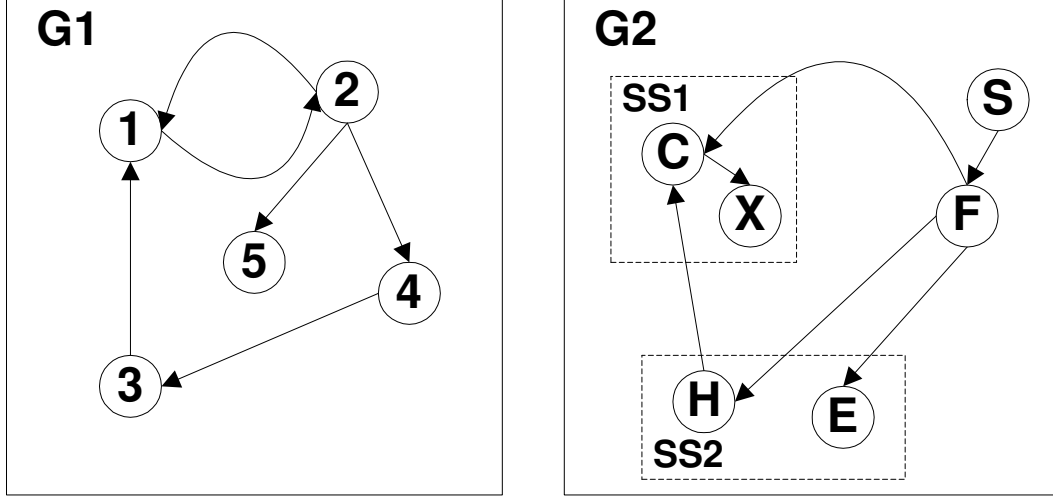


Figure 1.3: Software Structural Changes with Different Naming Scheme

one. Another algorithm relates vertexes from different graphs using the change patterns of paths (relations in software architecture) between them and their neighboring vertexes. Our case studies demonstrate that both algorithms, especially when used together, are very effective in tracking the evolution of source components in software architecture.

1.5 Major Contributions

This thesis makes the following contributions to the research of software evolution:

- We introduces an approach to studying software evolution that integrates the use of metrics, software visualization, and structural analysis techniques.
- We presentees a prototype implementation of the proposed integrated environment called *BEAGLE*. It incorporates data from various statistical and metrics tools, and provides a query engine as well as a web-based visualization and navigation interface.
- We developed "origin analysis", set of techniques for reasoning about structural and architectural change across multiple releases.

- We use the various functionalities of BEAGLE to analyze the structural evolution of GNU Compiler Collection (GCC).

1.6 Organization of Thesis

The remainder of this thesis is organized as follows. In the next chapter we review the state of art in software evolution research, with emphasis on works related to this research, such as evolution metrics and empirical studies. Chapter 3 we discuss our approach to design an integrated research environment to browse and navigate software evolution. We also propose two algorithms for tracking software architectural changes. In Chapter 4, we present a case study we have conducted that demonstrates how to use BEAGLE to study the architecture evolution of GCC software system. Several findings about the evolution history of GCC are shown in this chapter to illustrate the effectiveness of our approaches. Finally, Chapter 5 summarizes our work, and indicates future research directions.

Chapter 2

Related Research On Software Evolution

There has been a great amount of research in software engineering concerning the evolution of software systems. In this chapter, we discuss research that is related to our approach to provide an integrated software evolution study environment. We will include discussions on the theory of software evolution, evolution metrics, visualization of program structural changes, and empirical case studies.

2.1 Software Evolution: A Discipline of Software Engineering

In this section, we review Lehman's law of software evolution [31] and Perry's dimensions of software evolution environment [42]. These works are similar in the way that they are trying to build a theoretical foundation for software evolution research. Lehman proposed eight laws of software evolution based on his observations on the evolution of several industrial software systems over a long period of time. Perry focused on the context in which software system evolves. Factors in this context usually affect the way software evolves directly or indirectly.

2.1.1 Lehman: Laws of Software Evolution

Lehman has observed the evolution history of IBM OS/360 since 1968, and he has formulated eight generalized rules and hypotheses based on this system and others. He has called these rules the “laws of software evolution”. They include:

- *Law 1*: a software system that solves real-world problems must continually be adapted, otherwise it will become progressively less satisfactory.
- *Law 2*: as a program evolves, its complexity increases unless work is done to maintain or reduce it. As the need for adaptation arises and changes are successively implemented, interactions and dependencies between the system elements increase in an unstructured pattern and therefore led to an increase in maintenance costs.
- *Law 3*: the program evolution process is self regulating with close to a normalized distribution of measures of product and process attributes. This implies that after the system has stabilized through its early ages, software systems exhibit regular trends that we can measure and predict.
- *Law 4*: the average effective global activity rate on an evolving system is invariant over the product lifetime. Factors such as management, users, developers, support team, and the communication between them contribute to the stabilization of software size. As Brooks noted, simply adding more developers to the project will not improve the productivity proportionally [24].
- *Law 5*: during the active life of an evolving program, the content of successive release is statistically invariant. The rate of changes from release to release is constant.
- *Law 6*: functional content of a program must continually be increased to maintain user satisfaction over its life.
- *Law 7*: the quality of software program will decline unless it is constantly maintained to adapt to the changing environment. Program source code decays over time. Meanwhile, users’ expectation on the system keeps growing. Without proper maintenance, the quality of software systems perceived by users will decline over time.

- *Law 8*: software programming process is a multi-loop, multi-level feedback system that will self-stabilize over time. Lehman claims that this hypothesis has been validated on several commercial software systems [34]. However, different evolution characteristics have also been observed on applications from other domains and with alternative development processes [20].

Lehman’s laws of software evolution are mainly derived from studies of software applications that are developed in commercial environment using a closed development model. The first two laws state the general evolution characteristics of many software systems, irrelevant to the specific application domains. The rest laws are more specific to the particular application domains that were investigated by their empirical study, and the software process adopted by the developing organizations. Observations made by Godfrey and Tu indicate that these laws do not apply to many Open Source Software systems [20].

2.1.2 Perry: Dimensions of Software Evolution Environment

Perry introduced the concept of three dimensions of software evolution, and explained how the changes in these dimensions affect how a software system changes over time. These three dimensions are *domain*, *experience*, and *process*.

- *Domains* include the “real world” that encompass our application model, and theoretical sub-domains that provide infrastructure support for application systems. Any changes to the domain require corresponding changes to the software system. Domain changes are the fundamental and direct source of system evolution. As the “real world” and its model evolve, the specification of the software system must evolve as well, as does the actual implementation. On the other hand, software systems that do not solve “real world” problems directly, such as operating systems or programming language compilers, are influenced by the advances in computer science theories. Improvements in the areas such as algorithms or protocols bring new designs and implementations to these software systems.
- *Experience* is another dimension that affects software evolution. It does so by helping us better understand the software applications themselves, as well as the domain that

we are trying to model. Feedback from customers and developers provides insights into domain modelling, specification, and implementation of the software system. Experimentation is a systematic way of gaining experience. Scientific, statistical, and engineering experiments provide essential knowledge about aspects of software system and development process. The lessons we learned from experimentations help us to enhance the software system by improving the software development method and process. The evolution of our understanding and judgment caused by continued feedback, experimentation and learning is also one of the major reasons of software evolution.

- *Software process* includes mythologies, techniques, and tools that we use to develop and maintain software systems. When these technologies and processes evolve, it affects the ways we develop software systems, as well as the final software products that we create. New software development techniques such as modulation, abstract data type, object oriented analysis and design, design patterns and rapid prototyping had greatly influenced how software systems evolved over the past decades. Software process defines the way we develop and maintain software systems. Innovative development model such as “open source” development model and extreme programming model create software systems that are quite different from those developed with traditional process. Organizations define the culture and structure for software development process, as well as the final software products. Conway’s famous law hypothesizes the interesting relationship between organization structure and software systems. Bowman explored the ideal of using the organization of developers to recover software architecture [6].

2.2 Software Evolution Metrics

Evolution metrics measure how each version of system differs from its ancestors, descendants, and its many “cousins” in the evolution tree. Analyzing the trends of evolution metric values over time helps developers to manage the maintenance and continuous development of the system more effectively. Based on the past pattern of evolution metric values, several evolution models might be built to predict the fault rate of components in

future releases, to budget maintenance costs for various maintenance tasks, to discover undesired logical dependencies between components, or to schedule redesigns for components that are showing signs of decay.

There are many software artifacts from which we can collect evolution metrics. Researchers usually use three types of them to analyze software evolutionary histories: program source information, version control system database, and defect history database. History of source information provides the firsthand data on how the implementation of the software system has changed. Version control systems record the reasons for each code change, details of the change, components affected by the change, and sometime resources spent to implement the change. Defect database contains the description for each defects, causes of the defect, details of the fix and test cases of the verification for the fix. Researchers have built several evolution models that are based on evolution metric values collected from these program artifacts. We will introduce some representing models and the evolution metrics they used in the following sections.

2.2.1 Evolution Metrics From Program Source Information

Evolution metrics are divided into complexity metrics and change metrics. Complexity metrics include line of code (LOC), cyclomatic metric, fan-in, fan-out, etc. Change metrics measure how much code has been changed between two consecutive releases. They include number of lines of code added, deleted, or altered, the number of functions changed. In this section, we introduce some evolution models based on source metrics. Models are introduced together if they have similar purpose.

Understanding System Growth Patterns

Lehman *et al.* tracked the system growth history of IBM OS/360 and Logica FW system by measuring the number of modules in the systems for every release [32]. The purpose was to identify patterns in the system growth trend and to verify the “feedback system” mechanism as proposed in his laws of software evolution.

They developed an “inverse square growth” model to explain similar patterns found in the system growth curves of several industrial software systems. In this model, S_i is the

actual system size of release I measured as the number of modules, \widehat{S}_i is the predicted size, n is the total number of consecutive releases in the data set, and E is a model parameter. E is the average of individual E_i calculated as follows:

$$E_i = (S_i - S_{i-1})S_{i-1}^2$$

The model suggests that the rate of system growth tends to stabilize over releases. Lehman has used a “positive feedback” hypothesis to explain the fast growth rate towards the maximized “peak” of the curve, and “negative feedback” hypothesis to explain the declines of growth rate in more recent releases.

$$\begin{aligned}\widehat{S}_1 &= S_1 \\ \widehat{S}_i &= \widehat{S}_{i-1} + E/(\widehat{S}_{i-1})^2 (i = 2 \dots n)\end{aligned}$$

Godfrey and Tu examined the system growth and other evolution patterns of 96 releases of Linux kernel [20]. To measure the system size, they primarily used the uncommented lines of code for each release instead of the number of modules in the system as Lehman did, because they found modules in Linux kernel have great variation in LOC, and LOC tends to grow at the same pace with the number of modules. Other metrics used to reveal system growth patterns include the number of global functions, variables, and macros.

They observed that for Linux kernel, the growth rate of uncommented LOC fits well into a quadratic model, which does not agree with Lehman’s “inverse square” model. They therefore discussed the possible reasons for this disagreement. One of the most important reasons is the different development and maintenance process used by commercial software systems and open source software system.

They also found that the stable release stream of Linux kernel has different growth patterns from those exhibited by development release stream. When the growth pattern of each major subsystem of Linux kernel is analyzed, they noticed that unusually large size and high growth rate of the *driver* subsystem, and many instances of code cloning in this subsystem. They conclude that particular development processes, and unique application domains can greatly affect the growth rate and pattern of each software system.

Identifying Fault-Prone Program Components

Ohlsson and Wohlin developed a model based on structural complexity metrics to detect ageing and possible fault-prone program components [40]. The purpose of this model is to

help developers identify those components as early as possible, so that they can improve the quality of these components in the next release by using refactoring technique or redesigning the software architecture, before these aging components start to degrade the overall system maintainability. In this model, program components are labeled as green, yellow, and red, depend on their degree of decay. This classification is called G(reen)Y(ellow)R(ed) analysis.

Green Components - Components showing healthy maintainability and tractability.

Yellow Components - Decayed code that requires special attention to prevent possible defects in the future.

Red Components - Code that is hard to understand or contains potential defects. Red components need to be fixed immediately or risk disastrous consequences to the project.

Ohlsson *et al.* built the model by correlating the trends of a group of source code metrics with defect history of program components during past releases. They found that metrics that reveal program structural complexity such as cyclomatic metric, fan-in, and fan-out have the strongest correlation with the component defective rate. They also discovered that by incorporating “program state” metric and cyclomatic metric, this model produced the best prediction results, while using code size metrics alone produce the worst prediction. Using accumulated ranking, their predicting model shows strong correlation between the defect rates and components’ decay rankings.

Similar to the Ohlsson’s GYR model, Elbaum and Munson used a composite code measurement called “code churn” to predict program components’ future defects [27]. Besides complexity metrics, some researchers also suggested the use of coupling metrics as runtime-failure predictor [5].

Predicting Program Maintainability

Burd and Munro use “dominance tree” to model the system structure and call dependencies between program modules. They measure the specific characteristics of this “dominance tree” to predict the maintainability of program system [10].

Their “dominance tree” model is a static calling dependency graph of the program. Nodes in the tree are identified as either “direct dominated” or “strongly directly dominated”: if all the outgoing calls are made from one node to other nodes within the same branch of a subtree, we identify this node as a “directly dominated” node; on the other hand, if some outgoing calls are made to nodes outside its own branch, it is identified as a “strongly directly dominated” node.

The prediction model works in the following way. First, the portions of directly dominated nodes and strongly dominated nodes over the total number of nodes in the dominance tree are calculated. Their hypothesis is the larger the proportion of directly dominated nodes, the harder to maintain the system source code. On the other hand, the larger the proportion of strongly dominated nodes, the easier it is for maintenance. This is because when changes are made to directly dominated nodes, this will cause ripple effects to other call relations in the branch, and therefore it is much less desirable from the maintenance perspective. They applied this method to analyze the change of GCC from v2.7 to v2.8, and concluded that v2.8 is better maintained than v2.7.

2.2.2 Metrics From Version Control Management Information

Version control system (VCS) provides another valuable resource for the study of the evolution history of software systems. Compared to other artifacts generated during software development and maintenance, VCS information is more consistent because a software project usually sticks with one VCS system throughout its life. All the code-related history information is available from its database. The VCS database provides not only information about the text changes made to the project’s source code depot, it also records the date of the change, the number of lines affected by the change, developer who is responsible for the change, and a short description about the purpose and the nature of the change.

Many researchers have extracted valuable information from VCS database to benefit their understanding of software evolution. In this section, we review some of this work.

Determine The Readiness For Release

IEEE Standard 982.2 defines a Software Maturity Index (SMI). It is used to determine the readiness for release of a software system, when changes, additions, or deletions are made to the software systems comparing to previous releases. The history record of this index can also be used to study the impact of code changes. It is calculated as follows:

$$SMI = Mt - (Fa + Fc + Fd) / Mt$$

In this equation, Mt is the number of software functions/modules in the current release, Fc is the number of functions/modules that contain changes from the previous release, Fa is the number of functions/modules that contain additions to the previous release, and Fd is the number of functions/modules that are deleted from the previous release [1].

Predicting Future Fault Incidence

In the previous section, we have introduced some defect prediction models based on source code metrics. Many researchers discovered that change history extracted from version control system database is also effective in predicting future defects.

Graves *et al.* created a model that can predict the defect probability of a module by aggregating factors from the past changes (“deltas”) made to the modules [21]. The larger and more recent a “deltas” occurred, the more likely the component will have defects later. Their research is significant in that they reveal the close relations between past software development activities and future software quality.

They claimed that metrics based on change history information that is extracted from version control system database is more effective in predicting future defect than metrics based on program source code. Their statistic data shows that LOC is a weak indicator of defect rate. Since many complexity metrics are correlated with LOC, it implies that code metrics in general are not very effective predictors. Surprisingly, they also found there is no strong correlation between the number of developers involved in a change and the future defect rate of the module that is affected by the change.

In this model, two metrics are calculated from version control system database for each component of the system:

1. The number of deltas made to a module over the release history. History data shows

that this measure is proportional to the overall defect rate.

2. The average age of the module. This measure is calculated as a weighted average of the dates of the changes made to the module, weighted by the scope of the change.

A linear model that incorporates these two metrics provides the best prediction accuracy for future defects. An intuitive explanation is that components that have a long history and are seldom affected by changes are usually thoroughly tested and have fewer defects hidden inside.

They also introduced a more finely tuned, non-linear model called “weighted time damp model” that produces even better results. It summarizes contributions from all the “deltas” made to the component in the past, where old “delta” measures are down weighted by fifty percent per year. This improved model provides very satisfying results when they used it to analysis the error-prone components in the examined system.

The more recent study by Eick *et al.* used a similar methodology [13]. His purpose is to detect “decayed code” in legacy system. A section of program code is said to be “decayed” if it is much more difficult to change than it should be, that is, the cost and time required for the change increases, while code quality drops.

One direct reason for decayed code is past changes made to the software system. Other reasons include inappropriate architecture, violations of original design principles, imprecise requirements, time pressure, inadequate programming tools, etc. The main symptom of decayed code is quality degradation, which includes excessive code complexity, history of frequent changes, history of defects, widely dispersed changes, kludges in code, and excessive number of interfaces.

Eick *et al.* proposed several code decay indicators. These indicators are calculated from metrics based on version control management formation. Basic metrics include the number of deltas, the number of lines that are added or deleted in each delta, the date when a delta is complete, the interval to implement a delta, and the number of developers involved in a delta. Their model has six decay indicators: history of frequency changes (CHNG), span of changes (FILES), size (NCSL), age (AGE), fault potential (FPwtd, the weighted time model in [21] and FPglm, the linear model), and effort (EFF). Using these decay indicators, developers can easily identify decayed code in legacy system, so that

prevention can be taken before they become bottlenecks for the project. The effectiveness of these decay indicated is verified and proved work well on the change history data of Lucent Technologies' telephone switch system.

Revealing Hidden Dependencies Between Modules

Software maintainers often face a difficult task to identify hidden dependencies between modules in the software system. When one module is being changed, the ripple effect caused by hidden dependencies will impact some other modules. Such impacts are usually undocumented, and often cause expected problems.

Gall *et al.* investigated some change patterns exhibited by the change history of Product Release Database, and found that these patterns can help to reveal hidden “logical coupling” among program modules [16]. Their approach has two steps. The first is called *change sequence analysis* and the other is called *change report analysis*:

- *Change sequence analysis* identifies similar change sequence shown by program modules. Every past change made to a module in the system is labeled with the system release number. When these system release number are put together, it creates a “change sequence” for the module that records at which releases that this module was modified. Then they try to identify modules that show similar pattern in their change sequences. These common sequences reveal “logical coupling” between matched modules.
- *Change report analysis* verifies the logical coupling identified from *change sequence analysis* by examining archived “change report”. A change report records the reasons, defect class, amount, and type of every change. This step of analysis compares the “change report” of modules that are found to have similar change sequence. If the same change reason can be found in the change report for two “logical coupled” modules, for example, they both responded to the same bug report, then the “logical coupling” between these two modules are verified.

Similar work by von Mayrhauser *et al.* also attempts to detect hidden dependencies between modules in the system by exploiting information from history defect database [54].

Their approach is to build a *defect architecture* by identifying relationships between system components based on whether they are involved in the same defect report, and for how many times this situation happened in the past. The steps to build a defect architecture is described as follows:

1. First, they apply GYR analysis [40] to every components of the software system for all the history releases. All the decayed components (yellow and red) are identified and labelled. The source directory structure is used to as the framework for defect architecture, and those decayed components are attached as leaves.
2. Two or more decayed components (leaves) are linked, if they are related to the same defect report. This relation can be found at subsystem level as well, if two subsystems contain leaves that are already linked.
3. If a pair of components or subsystems exhibit persistent fault relations over many releases, it indicates that there are serious design flaws in the design of these components or subsystems.

In their case study, this method is able to pick three modules that all response to the same bug report and have the same change sequences. Their logical coupling are later verified in the description sections of the change reports.

2.2.3 Comparison of Evolution Metrics

All the models and the evolution metrics introduced here are summarized in Table 2.1. We list the name of the main author, the name of evolution metrics, the data source from where metric is measured, the purpose of the model, and some notes on the model.

2.3 Visualization of Software Evolution

Analyzing evolution metrics can help software managers and developers better understand the software development process, especially the maintenance and enhancement activities between releases, and more importantly, to plan and budget future development activities.

Author	Metrics	Aritfact	Purpose	Significance
Lehman (1997)	Number of modules in the system	Code	Understand system growth rate and pattern	Inverse square growth; feedback system
Godfrey (2000)	Lines of code, number of global functions, variables, and macros	Code	Understand system growth rate and pattern	Super linear growth; strong growth in particular subsystems of Linux kernel; counter example of Lehman's model
Ohlsson (1998)	McCabe and program states	Code	Identify aging and error-prone components	First to associate structural complexity with defect rate
Burd (1999)	Dominance tree	Code	Measure maintainability of components	Their case study is not very pervasive
Graves (2000)	Number of deltas, and average age	VCS log	Predict future defect rate	Claim to work better than code metrics
Eick (2001)	Various "delta" related metrics	VCS log	Identify decayed code	Six decay indictors based on "delta" metrics; verified with very large-scale and extensive case studies
Ramil (2000)	Module or subsystem changed, added or deleted	VCS log	To estimate evolution effort	Subsystem level change metrics are more effective than module level
Gall (1998)	Change occurred, and reason of changes	VCS log	Identify hidden dependencies	components change together tend to have logical dependencies; very novel approach
Mayrhauser (1999)	Defect report	VCS log	Identify hidden dependencies, and bad architecture design	Construction of "defect architecture" and "cumulated defect architecture"

Table 2.1: Research on Software Evolution Metrics

However, existing evolution metrics are not able to model structural changes made to the software system during its release history because many metric-based evolution models assume that the software system maintains a static architecture.

Information visualization has been demonstrated as valuable software comprehension tool for users to understand software architecture and its evolution. In this section, we review some of the work that applies visualization techniques in software evolution research.

A general introduction to software visualization and various visualization techniques are discussed by Ball and Eick in [3]. When a software project grows large, and has a complex architecture, it becomes impossible to recognize the high-level system structure and behaviors by analyzing only the source code. Software visualization tools help software developers deal with the complexity and increase their productivity for continuous development of the software system. Software visualization use graphical techniques to make software architecture visible by displaying program artifacts and behaviors. Ball and Eick demonstrated two tools that can visualize code differences between releases:

- Visualizing code version history. Data from version control system are visualized with special views for code age, age and bug fix, and fix-on-fix information.
- Visualizing code difference between releases. This tool displays the differences between source directories and file pairs simultaneously. Four colors are used to highlight the change status of code: red for deleted code, green for added lines, yellow for changed lines and gray for unchanged text.

Their tools focus mainly on the changes at source code level. They are not capable to represent changes at software architecture level.

2.3.1 GASE Tool and KAC System

GASE is a **G**raphical **A**nalyzer for **S**oftware **E**volution tool that takes the architectural facts of consecutive software releases, and generate visual diagrams that represent the architecture evolution[22]. The GASE tool consists of four components: a fact extractor, a change analyzer, a diagram generator, and a visualizer. The fact extractor parses the source code of select releases of target system, and extract the “facts” from the source

code that contains architecture information, such as the system containment hierarchy, and “call” and “include” relations between program entities. The change analyzer compares the difference between the “facts” of two releases. The diagram generator translates the difference of “facts” into colored box-and-arrow diagrams ready for visualization. Finally the viewer displays the architecture difference between selected releases with a navigation and query interface.

KAC system is another evolution visualization tool similar to GASE [25]. The difference is that it reuses many components from popular reverse engineering tools such as CIA for fact extraction and Rigi for visualization.

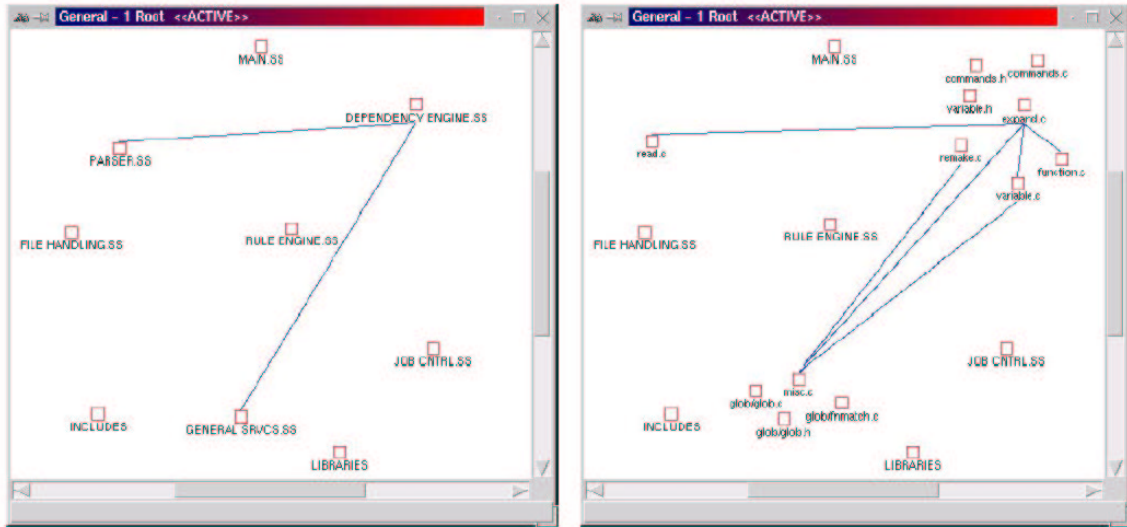


Figure 2.1: Screenshot of KAC System

In KAC, colors are also used to illustrate three different types of changes of architecture entities: preserved, added, and removed. The screenshot of KAC is shown in figure 2.1. The window on the left shows the visualization at the subsystem level. The window on the right shows the zoom-in effect, as modules and dependencies inside subsystem DEPENDENCY ENGINE are shown in greater detail.

2.3.2 Gall: Software Evolution in Color and 3-D

Gall *et al.* incorporated a three-dimension model and color histogram in the visualization of software release histories [18]. Different from GASE and KAC, which can only compare two releases at a time, this approach consolidates the entire change history of a software system into single 2-D or 3-D diagram. With a special VRML interface and a unique color scheme, users can visualize and navigate in the 3-D space to search interesting patterns and hidden relations between modules.

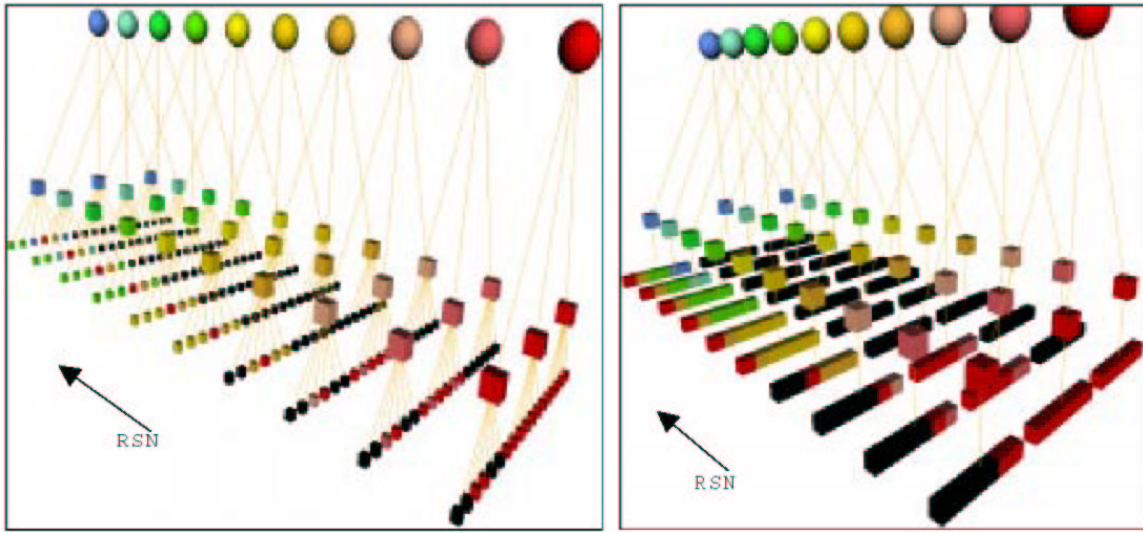


Figure 2.2: Screenshot of 3-D Evolution Diagrams

In the 3-D model, x and y dimensions show the software system architecture. The architecture is modeled as a layered tree with four levels: system, subsystem, module and program. The z dimensions represent time, labelled by the release number. Color with different saturations acts as an additional dimension that illustrates the evolution attribute of modules: the relative module age measured by the module release number (module release number is the system release number when this module was last modified). Figure 2.2 shows a sample 3-D evolution model. The diagram on the left visualizes the evolution of whole system with individual module shown as leaves of the system hierarchy

tree. The diagram on the right uses percentage bars to represent the proportion of modules of different age.

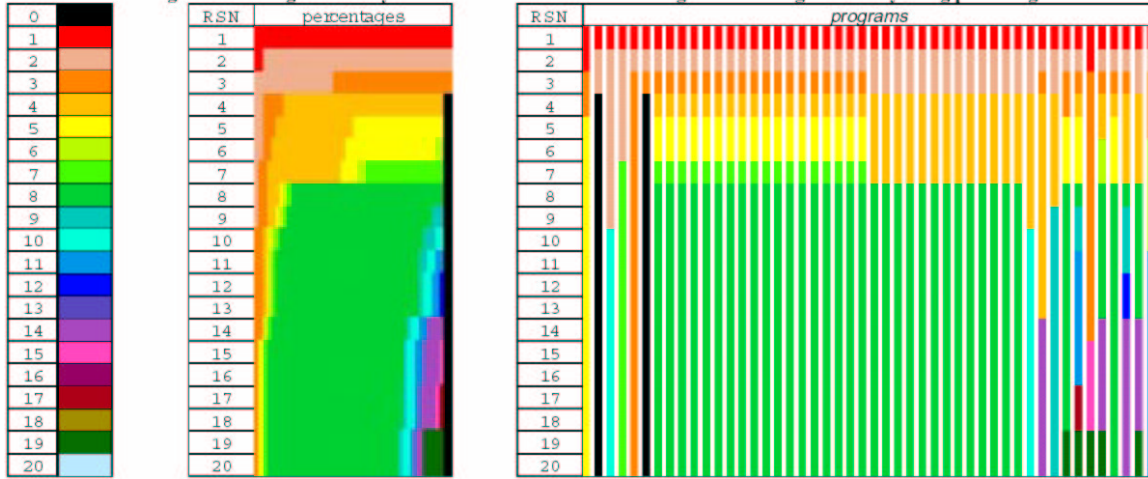


Figure 2.3: Screenshot of 2-D Evolution Diagrams

The 2-D model is a simplified model that does not contain any system structural information. It is a quick tool to overview the evolution history of all the modules in the system, as shown in Figure 2.3. On the left is the color scale that shows which color corresponds to which system release. The chart in the center shows the percentage of modules with different age at each release. The chart on the right shows the the age of each program module.

The advantage of this model is its simplicity. Using 3D diagram and 2D color histogram, user can get an instantaneous view of the system's evolution history. The disadvantage of this model is that the evolution information provided for each module is limited to its age. No furtherer information is provided for each module and its relations with other modules in the system.

2.3.3 Comparison of Evolution Visualization Techniques

All these visualization technologies provide some capabilities to visualize the evolution of software system structure. GASE and KAC adopt a reverse engineering approach to

compare extracted software architecture facts. The 3D software release history model is unique in that it provides the quick overview of the entire software revision history, and a powerful visual cue for identifying certain change patterns.

There are also limitations shared by these techniques. For example, users do not have much freedom in selecting multiple releases for architecture comparison; the query facilities are very limited and user cannot create arbitrary queries; the comparison engines capture limited types of architectural changes; and the navigation interface provides little information about individual program module.

2.4 Empirical Studies of Software Evolution

Empirical study helps us to understand how and why things work, and allows us to use the knowledge to materially change our practise and outcome. Empirical study has been applied widely in many other scientific research areas, but with limited success in software engineering, especially in software evolution [42, 26].

Perry discussed the difficulties in conducting the useful empirical studies in software engineering [42]. It is very hard to define and implement empirical study that could be relied on to change an organization's long-practiced development processes. The empirical study must be carefully designed, and the conclusion must be persuasive and general. Kemerer and Slaughter sited specific obstacles for software evolution studies such as difficulties in collecting historical data, and the lack of existing theory [26].

To conduct a successful and credible empirical study, we must maximize the accuracy of interpretation of data and observation, the relevance of our result to software engineering principles, and the impact on the software engineering practice. Perry proposed a six-component structure for a successful empirical study on software engineering. These six components are:

- Research context. The problem of focus is defined and its terminology is explained. Also the goal of study is linked to what is currently understood about the problem.
- Hypotheses. Hypotheses are essential to empirical study as they state the research questions we are asking. An example hypothesis for software evolution is “Does

software system always grows its system size when new releases come out?”

- **Study Design.** It is a detailed plan for creating the data that will be used to test its hypotheses. We need to design dependent and independent variables to link causes and effects, a plan to systematically manipulate independent variables to change predictably the way independent variables change, and the operational context of the study.
- **Threats to Validity.** These are influences that may limit our ability to interpret or draw conclusions from the study’s data and observations. Three types of validates must be protected in the empirical study:
 - **Construct Validity:** independent and dependent variables must accurately model the abstract hypotheses.
 - **Internal Validity:** changes in the dependent variables can be attributed to changes in the independent variables.
 - **External Validity:** results should be generalize to environment outside the study context.
- **Data Analysis and Presentation.** Quantitative analysis and qualitative analysis are two general approaches.
 - **Quantitative analysis** means comparing the numerical data. There are two tools commonly used. Hypothesis testing determines the confidence level at which the null hypotheses can be rejected. Power analysis determines the likelihood that the null hypothesis will be rejected when it really should be.
 - **Qualitative analysis** rely subjective data such as observations and interviews to understand human’s perspective of software process.
- **Results and Conclusion.** This is the weakest part of current empirical studies. In this section, we must explain the limits of the study, what the data says and how the data related to our initial problem.

2.4.1 Review of Software Evolution Empirical Studies

In this section, we review some empirical studies on software evolution. Each of them will be introduced and evaluated using Perry's empirical study principle.

Patterns of System Growth

As introduced in the section on code-based evolution metrics, Lehman and Godfrey have conducted studies on the system growth of long-lived software systems, as summarized in [33] and [19]. Lehman's early work was conducted on IBM OS/360 operating system and contributed to his "laws of software evolution". More recently, empirical data of industrial systems from ICL, Logica, BAE and Ministry of Defense have been studied. The result shows that many system growth curves fit into a "single parameter inverse square model".

Godfrey and Tu have examined the growth patterns of some emerging Open Source Systems. They studied the system growth of Linux kernel for over ninety releases, as well as many past releases of VIM and GNU C Compiler system. Both the overall system growth and the evolution patterns of individual subsystems are studied. They found that the system size of Linux kernel grows at a geometric rate (super linear). The authors attributed this result to the development characteristics of open source projects, such as the large number of developers contribute to the project in parallel, the distributed debugging process, and the centralized control over system architecture. They moreover found that common coding practices in open source project such as code cloning contribute to the unique growth patterns of many subsystems.

Evolution of Lucent PBX System

Mockus, Eick, and other researchers at Lucent Technologies conducted the large-scale empirical study on the change history of Lucent's main telephone switch system for over fifteen years [13]. The system is huge, consisting of 100 million line of code, another 100 million lines of header and Make files, organized into 50 subsystems, and 5000 modules, with 10 thousand developers involved.

They have extracted useful information from the database of change management systems, including Source Code Control System, Extended Configuration Management Sys-

tem, and Fault and Feature Tracking System. Their methodologies and analyzing tools are discussed in [37]. The goal of their research is to identify evidence of “code decay” in the system (“code decay” has been discussed previously in the section on evolution metrics). They collect several code decay indicators (CDI), and use them to diagnose the well being of the system. The analysis result shows some evidences of “code decay”, such as the increase over time in the number of files affected per change to the code and the decline in modularity of subsystems. The results also correlate factors such as the frequency and age of the change to the fault rate in modules, and the span and size of changes to the effort required to implement a change. Finally, the researchers concluded that there is no evidence of dramatic, widespread decay found in the system.

Evolution of Maintainability

Burd *et al.* performed an empirical study that compares the evolution history of four software systems [9]: a retail system over eight releases with a size of 10 KLOC, GNU C Computer system over thirteen releases sized at 300 KLOC, an operating system over four releases sized at 20 KLOC, and another retailed system over four releases sized at 40 KLOC. Their approach is to identify the increase of data complexity in the applications. Modules that had rapid increase of data complexity, but with relatively fewer changes to their call structures, are likely error-prone and may require maintenance.

They observed in one of the retailer system, there had been a significant increase in the data complexity in earlier releases. However, the increase had stopped when preventative maintenance had been performed in later releases. In the other retailer system and the operation system, a convergence of cumulative changes in call and data dependencies was identified, which suggested that a consistent preventative maintenance strategy has been applied. Finally, the GCC system exhibits a chaotic change history. They interviewed some developers of GCC, and they were told that a preventative maintenance approach was only attempted when time was permitted.

Different Change Characteristics Between System and Its Components

Gall *et al.* studied the product release history of a telecommunication application for twenty releases [17]. They compared the size growth of the whole system with individual

subsystems. In addition, they measured the changes of functionality in terms of modules added, removed or changed in the whole system and each major subsystem.

They observed some interesting growth characteristics at the system level. For example, the size of the system is growing linearly, which is very high for industrial system. In the beginning, the number of modules added into each new release is very large, and then it significantly decreases and becomes almost constant. Eventually the entire system evolved into a stabilized stage, where both the growth and change rates are decreasing.

Then they examined the evolution of one particular subsystem, because this subsystem has the highest growth rate and change rate. They found one module of this subsystem has many functions with similar names with slightly different endings. This indicates that many new functions are only slightly modified from existing ones. In fact, this module contains all the configuration information for the system. New configurations are often copied from existing functions with little modifications, and old configurations are seldom changed in new release. This explains the high growth rate, and low changing rate of this module. The other two modules of the subsystems both have a high growth and change rate, which means almost every other change in the system will affect these two modules.

Their conclusion of this empirical study is that there are significant differences in the evolution characteristics between the whole system, and individual subsystem or module.

2.4.2 Conclusion of Related Work

Many studies on software evolution emphasize the statistical changes of the software system by analyzing its evolution metrics. Beside some visualization tools, very little work has been done to help understanding the nature of the evolution of software architecture.

Another limitation of many empirical studies is the number of releases examined and the history of many archived data that is not long enough to generalize the results of the study as an evolution theory. However, the enormous amount of work required by large-scale empirical study makes it almost impossible without the application of dedicated tools and integrated environment, like the *SoftwareChange* environment created by Lucent Technologies when they conducted the imperial study discussed in [13]. Strong tool and environment support has been proven a key factor in conducting a successful empirical study on software evolution.

Our approach to study software evolution incorporates some of the research methods covered in this chapter, such as applying popular evolution metrics to measure the historical changes of system components, and using graphical diagrams to visualize the software change history. The main difference between our approach and existing methods is that we created an integrated environment that allow researchers to investigate the software change history from many aspects including evolution metrics, visualization graphs, and other analytical tools. We also introduce an analysis method that could track the structural changes of the software system when its source directory structural or file naming scheme was changed. In the next chapter, we will describe our integrated approach and structural analysis methods in detail.

Chapter 3

BEAGLE: An Integrated Platform for Studying Software Evolution

In chapter 2, we reviewed the two popular research approaches for studying software evolution. The first approach is to collect and analyze the historical trends of evolution metrics. The second approach visualizes the evolution of software organizations with graphical representations. We discussed the advantages and limitations of each of these two approaches, and then raised the issue that there has been limited research effort in studying evolution at the architectural level. Furthermore, we reviewed several empirical case studies on software evolution, and finally proposed an integrated environment with automated tools to assist the empirical study in this area.

To overcome the limitations in current research, we have proposed a new research method, which integrates evolution metrics as well as visualization techniques into one web-based research platform. In this environment, researchers can query the history data of the software system, compare the differences between releases, and investigate interesting evolution patterns from different perspectives. We also developed a technique called “origin analysis” to examine the structural change of software systems. The purpose of this technique is to find possible matches between renamed or relocated software components in the later releases with their “origins” from an earlier release.

In this chapter, we first introduce the research problems we are trying to solve, then we discuss the methodologies followed by the discussion of our methodologies, and finally

we describe a prototype implementation, an integrated environment called *BEAGLE* that we have developed to help researchers study software evolution.

3.1 Challenges to Software Evolution Research

We believe there are three major challenges that we must overcome in software evolution research. These obstacles limit our ability to understand the history of software systems using effective empirical study, thus prevent us from generalizing our observations into software evolution theory.

The first challenge is how to organize the enormous amount of historical data in a way that allow researchers to access them quickly and easily. Software systems with a long development history would generate many types of artifacts. We need to determine which artifacts should be collected as the data source for software evolution analysis.

The second challenge is how to incorporate different research techniques of software evolution into one integrated platform. We have reviewed several models that are based on software evolution metrics, and visualization techniques that display software history in graphical diagrams. Evolution metrics are precise, extendable, and can be used for numerical analysis. Visualization diagrams provide the overview of the evolution history and have visual appeal to the users. When used together, they are valuable tools for software evolution study.

The third challenge is how to analyze the structural changes of software systems. We have discussed in the previous chapter the needs for this analysis, and why traditional name-based comparison techniques are not effective to solve this problem. New research methods must be explored to solve this challenging problem.

3.2 Discussion of Methodologies

In this section, we introduce our answers to the three challenges listed above. Our approach includes a web-based research platform that integrates several essential techniques in studying software evolution, and a novel approach for analyzing software structural changes that is included in our research platform.

We first discuss how the data are selected and stored in the platform, followed by the discussion of the various analysis methods integrated in the platform, and how to apply them to solve problems in evolution research.

3.2.1 History Data Repository

Data Source

As a software system evolves, the various activities related to its evolution produce many new and changed artifacts. These artifacts include:

- Program source code, **Makefiles**, compiled binary libraries, and executables. These artifacts are the main products of software development activities.
- Artifacts related to the requirement specification and architecture design. They include feasibility study, functional and non-functional requirement specification, user manual, architecture design documents, user interface mockup, and prototype implementation.
- Artifacts related to testing and maintenance activities. They include defect reports, change logs, new feature requests, test suites, and automated quality assurance (QA) tools.

The archives of each of these artifacts reveal one or more aspects of the software evolution. Collecting and organizing these artifact archives are usually the first step in an empirical study.

In this thesis, our primary focus is the evolution characteristics of Open Source Software (OSS) systems. The reason is that most OSS projects maintain complete archives of program source code and version control database for history releases on their FTP sites, and free of charge. With the full access to program source code and version control database, we have been able to discover many details of the history of these software systems. We also have more freedom in our research without getting into complicated copyright or confidential issues as the case with commercial systems.

One characteristic of OSS development process is that developers often do not systematically achieve initial specification and design documents. Much of the documentation consists of sketches on scratch paper or drawing board, private emails between developers, or newsgroup discussions. This “bazaar” development style contributes to many factors that make OSS successful, such as short release cycle, quick adoption of new features, and prompt responses to bug reports [44]. However, the lack of well-archived documents make it difficult to investigate the evolution of OSS systems. For some OSS projects, the key documents that record original design decisions or reasons for important changes are either lost or difficult to retrieve.

On the other hand, OSS project usually has a complete archive of version control information, because the code “diff” is often distributed as patch to update the program source code from the earlier release to the current, where the end user can build the newest program binaries. The version control database also has a web interface for distributed development and debugging. However, typically version control information describes the text changes made to the source code at line-level, and sometime with a short description of the changes. It does not explain the context of the code change, such as the high-level modification to the software architecture, and its relations with other code changes. If the software developer did not document them explicitly, it is very hard for us to realize exactly what had been changes, and for what reasons. This makes version control database, when used alone, an ineffective resource of data for investigating software architecture evolution.

The *ad hoc* nature of OSS development process makes it very difficult to find archived documentation that describes in detail all the major changes made to the software architecture in the past. Fortunately, OSS projects usually maintain a complete source code base for every past releases. Furthermore, there are many software reverse engineering techniques that can extract and rebuild some of the software architecture information from the source code. As the result, we selected source code as the primary data source for studying the evolution of software architecture, and other program artifacts including version control database are used as complementary.

Many OSS projects publish the source code of past release on FTP sites. Table 3.1 lists three popular OSS systems, the archived releases, and their FTP addresses.

Project	Archived Releases	FTP Site
Linux Kernel	Linux 0.0.1 (9/17/1991) To Linux 2.4.9 (present)	ftp://ftp.kernel.org/pub/linux/
GCC	GCC 1.37.1 (2/21/1990) To GCC 2.95.2 (10/24/1999)	ftp://ftp.gnu.org/gnu/gcc/
VIM	Vim 3.0 (3/5/1996) To Vim 5.8 (5/31/2001)	ftp://ftp.vim.org/pub/vim/

Table 3.1: Release archives of three open source projects

Software Architecture Model

In this thesis, *software architecture* refers to the structure of system, emphasizing the organization of its components that make up the system and the relationships between these components. We apply reverse engineering techniques on the program source code to extract the most basic architecture facts including program components and their relationships, and then recreate the high-level software architecture using fact abstractors and relational calculators.

Depending on the abstraction level, we have four architecture models that describe the structure of the software systems using components and their relationships [51]. Each model describes the system structure with a different level of abstraction. By modeling the software system at several abstraction levels, researchers can not only study the overall organization of the system, but also be able to “drill down” the high-level component to further examine its internal structure. The four architecture models are:

1. *Entity-Level Model* This model describes the data and control flow dependencies between basic program entities, such as functions, non-local variables, types, and macros. It also describes the containment relations between these low-level program entities and their containing entities, which are program files.
2. *File-Level Model* This model describes the control flow and data flow dependencies between program files or modules. These higher-level entities and relations are “lifted up” from those in the function-level model using relational calculus.

3. *High-Level Model* This model also describes the dependencies between program files or modules. However, the dependencies are the abstractions of those presented in the file-level model. Related dependencies are grouped into three basic relation group: function call, data reference, and implementation relations between header files and implementation files. Instead of having more than ten different dependency types as in the file-level model, high-level model has only three basic dependency types.
4. *Architecture-Level Model* This model describes the software architecture at the highest abstraction level. Program entities models at this level are mainly subsystems and files. A subsystem is a group of related files or lower level subsystems that implements a major functionality of the system. The process of creating subsystems for a software system is mainly performed manually with assistances from the source directory structure, filename convention, and automatic module clustering tools. The relations between subsystems are described by the same three basic relation types as in the higher-level model.

Evolution Metrics

Code-based evolution metrics are valuable information to study the evolution attributes of individual program entities. Our integrated platform provides access to several evolution metrics in addition to the capability to compare software architectures of different releases.

The metrics we selected include basic metrics and composite metrics. Basic metrics include lines of code, lines of comments, cyclomatic complexity, code nesting, fan-in, fan-out, global variable access and update, number of function parameters, number of local variables, and the number of input/output statements. Composite metrics include S-complexity, D-complexity, Albrecht metric, and Kafura metric [28].

In addition to architecture facts extracted from program source code and evolution metrics that are also measured from source code, we need data that provides extra information about each past release. This information includes the release date, the full version number, the new feature list, and the bug fix list.

3.2.2 Navigation of Evolution Information

Incorporating Evolution Metrics with Software Visualization

Previous works in incorporating software metrics with visualization techniques in program comprehension have been discussed by Demeyer *et al.* [12] and Systa *et al.* [48]. Demeyer *et al.* proposed a hybrid reverse engineering model based on the combination of graph visualization and metrics. In their model, every node in a two-dimensional graph is rendered with several metrics at the same time. The values of selected metrics are represented by the size, position, and color of the node. Possible graph type includes tree, correlation, histogram, checker, and stapled graph. They also implemented a platform called *Code-Crawler* to experiment with various combinations of metrics and program visualization techniques. Systa *et al.* have developed a reverse engineering environment called *Shimba* for understanding Java programs. Shimba uses reverse engineering tools Rigi and SCED to analyze and then visualize the static structure and dynamic behavior of a software system. The nodes in each of these diagrams are annotated with metric attributes. These metrics measure the properties of the classes that are represented by the nodes, the inheritance hierarchy of the Java program, and different relations between classes.

Both approaches assist program comprehension by combining the immediate appeal of software visualization with the scalability and precision of metrics. We are proposing to adopt a similar approach in software evolution research, by creating an integrated platform that integrates evolution metrics, program visualization, software structural analysis, and source navigation capability into one environment.

The platform should provide at least two windows when showing the evolution information of software systems. The first window shows a visualization that models the history of the whole software system, or selected program entities or relations. When the user needs more detail about particular program entity, he can click on the graphical element that models the entity and the second window will be shown. This window contains a table showing the history of the evolution metric measurements of interested program entities.

Comparing Differences between Releases

As we have discussed in the previous chapter, there are two common approaches to visualizing software evolution. The first approach attempts to show the evolution information with one graph for all the history releases. The example is Gall’s colored 3D evolution graph. The other approach shows the architectural differences between two releases, as seen in GASE and KAC systems.

Our method is to display the two types of evolution visualization graph at the same time. First, we provide a tree-like diagram that shows the system structure of one of the release that is included in the comparison, usually the most recent one. We call it the *structure diagram*. The *structure diagram* models the system hierarchy as a tree with branches and leaves. The “branches” or internal nodes of the tree represent subsystems and program modules. The “leaves” of the tree represent functions defined in the program modules. User can click on a “branch” (a subsystem) to expand it to show the lower-level “branches” (modules), and further to “leaves” (functions). We also use colors and saturations to model the evolution status of each entity in the “tree”. Red, green, blue, and white are used to represent “new”, “changed”, “deleted”, and “unchanged” status respectively. To differentiate program entities that are all “new” (added to the system later than the first version in the comparison was released), different levels of red are used to represent their relative “ages”. Entity in vivid red came into the system most recently, and the darker red means the entity has been in the system for many releases. With the help of the tree diagram and a novel color schema, we can model the evolution of the system over several releases in a single graph.

The other kind of diagram is shown next to the tree diagram: it is designed to display and navigate the differences between two releases. We call it the *dependency diagram*. The *dependency diagram* is based on the landscape viewer used in PBS tools, and extends the schema by adding evolution related entities and relations.

If a user selects a group of releases and wants to visualize the change history, the tree graph usually shows the program structure of the most recent release in the group, with different colors to represent the different evolution status of its program entities inside. The software landscape graph will show the differences between the architecture of the earliest release and the most recent release, especially the structural difference of the program

entity that is selected in the tree diagram between the two releases. By showing the two types of visualization graphs together, user can examine the evolution from many different perspectives, and navigate from one diagram directly into another diagram. Figure 3.8 shows a prototype implementation of the ideals discussed above.

3.2.3 Analysis of Software Structural Changes

The method we have developed to analyze software structural change is called “Origin Analysis”. We use it to find the possible origin of a function or file that appears to be *new* to a later release of the software system, if it existed previously within the system in another location. Many re-architecting (high-level changes to the software architecture) and refactoring (low-level modification to the program structure) activities involve reorganizing the program source code by relocating functions or files to other locations, with little change actually made to the program entity. Meanwhile, their name may also be changed to reflect a new naming schema. As a result, many *new* entities that appear to be added to the newer release of the system are actually *old* entities in the *new* locations and/or with a new names.

We define “origin analysis” as the practice to relate program entities from the earlier release with the apparent *new* entities in the later releases. With “origin analysis”, the transition process from the previous program source structure to the new one could be better understood because we are able to unveil many hidden dependencies between the two architectures.

Why Origin Analysis?

Imagine we are given a task to analyze the software evolution of System X. System X has two releases so far, release v1.0 and release v2.0. Figure 3.1 shows the system structures of both v1.0 and v2.0.

After we compared the two architectures based on the names of program entities from both releases, we create a graph that shows all the new entities and relations in v2.0, and another graph that displays all the entities and relations that will be missing in v2.0. Provided with these two diagrams, questions such as “Where does file `D.c` in subsystem `S2` go

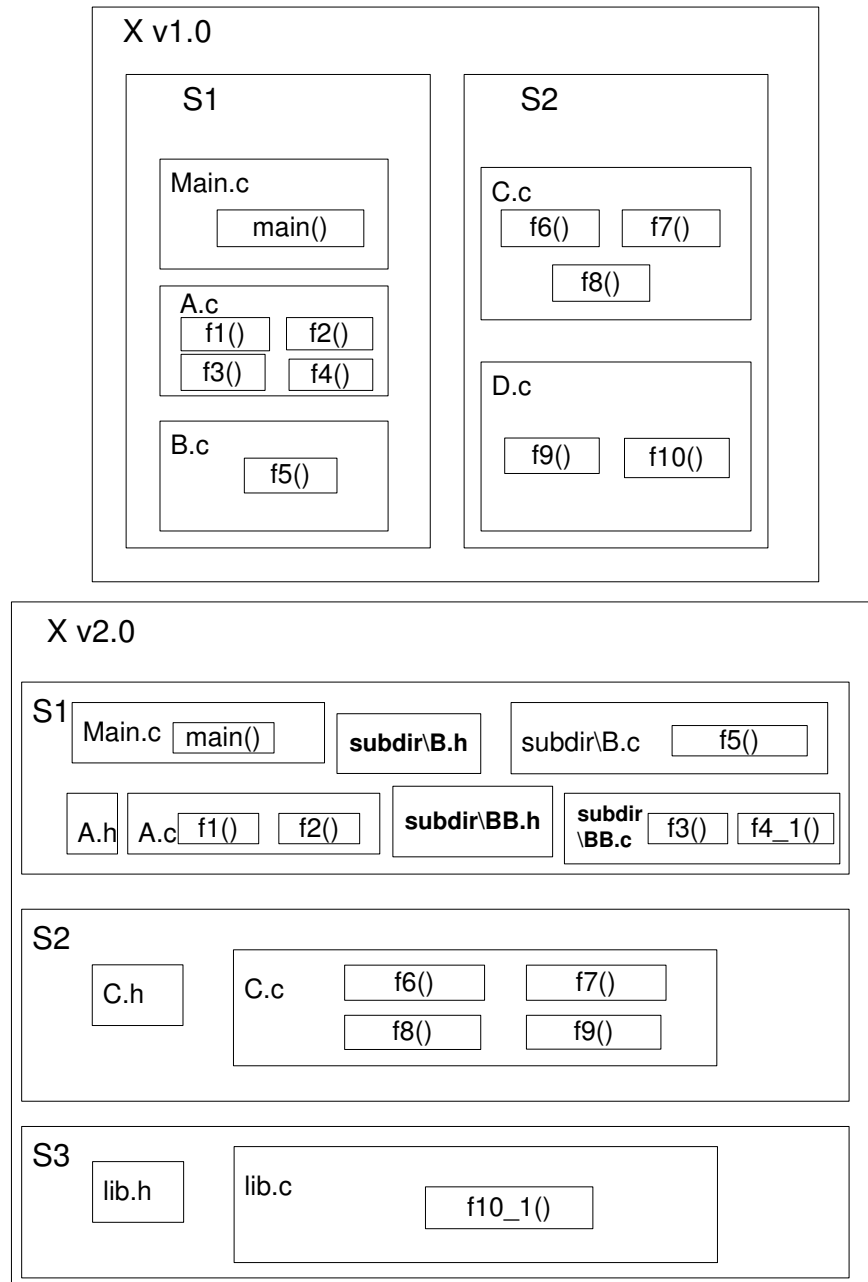


Figure 3.1: Example of Origin Analysis

in v2.0 and why?” or “Where does file `lib.c` in subsystem S3 come from in v2.0?” remains unanswered. We need more sophisticated analysis methods to answer these questions, or at least provide some clues if no conclusive answers can be provided.

In the next section, we will introduce two techniques that we have developed to implement *origin analysis*. The first technique is called *Bertillonage Analysis*. It uses code features to match similar program entities from different releases. The other technique is called *Dependency Analysis*. It exams the changes of relationship between selected program entity and those who are depended on it to find the possible match.

Bertillonage Analysis

Bertillonage analysis was originally used by police department in France in the 1800s to attempt to uniquely identify criminals by taking the measurements on various body parts like thumb length, arm length, and head size. This approach predates the use of fingerprints or DNA analysis as the primary forensic technique. We borrow this term to describe our approach to measure the similarity between new functions identified in a later release with those missing functions from the previous release, hoping to find a pair positive matches so that we can declare this “new” function has an “origin” in the previous release. We used the term “Bertillonage” as it is an approximate technique. Unlike more advanced techniques such as fingerprinting and DNA analysis that require more effort and take longer to conduct, “Bertillonage” is able to identity a small group of “suspects” easily and quickly from tens of thousands of population. We could use other advanced techniques that requires more computing power, or sometime even common sense, to filter out the real “suspect” from a much smaller population.

This approach was first used in *clone detection*, where the goal is to discover similar code segments within the same software release. We extend its application to software evolution, where we try to match similar functions from different releases to analyze structural changes. “Bertillonage” is a group of program metrics that represent the characteristics of a code segment. Kontogiannis proposes to use five standard software metrics to classify and represent a code fragment: S-Complexity, D-Complexity, Cyclomatic complexity, Albrecht, and Kafura [28].

We have pre-computed and stored these five measurements for every function in every

release of the system under consideration. Any two functions from consecutive releases with the closest distance between their measurement vectors in a 5-D space are potential candidates for a match. The rational is that, if a new function defined in the later release is not newly written, but rather an old function relocated from another part of the system in the previous release, then the “new” function and “old” function should share similar measuring metrics, thus they should have the closest Euclidean distance between their Bertillonage measurements. The matching algorithm is described as follows:

1. As the result of an architectural comparison, a function in the reference release is identified as “new”, which means a function with the same name in the same file does not exist in the previous release.
2. Compile a “missing” list that contains functions that existed in the immediate previous release, but do not exist in the current release.
3. Match the Bertillonage measurement vector of the “new” function with that of every function in the “disappeared” function list. Sort their Euclidean distance in ascending order.
4. Select the five best matches.
5. Among the five best matches, compare their function name with the “new” function being matched by matching the common substring in their names. Choose the one whose function name is the most similar to the “new” function, which means it has the longest common substring with the function name that we are matching with.

The last step of comparing function name works as a filter to discard mismatched functions, since there are chances that two irrelevant functions happen to have very similar Bertillonage measurements. Here is an example from the case study of GCC that illustrates why it is necessary. In GCC 2.0, there is a new function `build_binary_op_nodefault` defined in file `cp-typeck.c` in subsystem `Semantic Analyzer`. When applying Bertillonage analysis, we get the following five best matches. The distance `d` is calculated as the Euclidean distance between the two file-element vectors that represent the code features of selected functions.

- | | |
|--|--------------|
| 1. combine from fold-const.c: | d=1005745.47 |
| 2. recog_4 from insn-recog.c: | d=2496769.23 |
| 3. insn-recog.c from recog_5: | d=7294066.05 |
| 4. fprop from hard-params.c: | d=8444858.78 |
| 5. build_binary_op_noddefault from c-typeck.c: | d=8928753.44 |

The obvious choice should be match number 5, which has the exact filename as the “new” function. The only difference between these two functions is the files in which they are defined. However, they do not have the closest distance, as match 1 to 4 are much closer to the “new” function than the correct “origin” function. The explanation could be that this function has somewhat changed its internal structure (control flow and data flow) in v2.0, so it measured as distant in the 5-D vector space. However, since these two functions are expected to implement the same functionality in both releases, we can still pick them up with Bertillonage matching algorithm enhanced with function name filter.

Dependency Analysis

We use the following analogy to explain the basic idea behind the *Dependency Analysis*: imagine a company that manufactures office furniture has decided to move from Toronto to Waterloo. This event will affect both its business suppliers and customers. Its supplier, say a factory that provides building material to the company, must update its customer database by deleting the old shipping address in Toronto, and then adding a shipping entry to reflect the new address in Waterloo. The customer, for example, Office Depot, also needs to update their supplier database to delete the old Toronto address and update it with the new Waterloo address. If we do not know the fact that the new office furniture company that just registered with City of Waterloo is actually the old company with many years of operation history in Toronto, we can compare the changes of the customer database of its suppliers, and the supplier database of its customers to discover this move.

The same type of analysis can also be used for analyzing software architectural changes. In this case, we are trying to identify a particular change pattern on call dependency. Here is a description of how the dependency analysis is performed to track function movements:

1. Identify the “new” function in the reference release.
2. Analyze the caller functions:
 - (a) Find all the caller functions of this “new” function.
 - (b) For every caller function that also exists in the previous release, compare the differences of the function lists that it calls in both releases. Select those functions that were being called in the previous release, but no more in the reference release.
 - (c) Any functions that are selected more than once are candidates for the origin of the “new” function.
3. Analyze the callee functions:
 - (a) Find all the functions that this “new” functions calls in the reference release.
 - (b) For every callee function that also appear in the previous release, compare the difference of the list of functions that call it in both releases. Select those functions that were calling it in the previous release, but no more in the reference release.
 - (c) Any functions that are selected more than once are candidates for the origin of the “new” function.
4. By combining the results from previous two steps, we might find the “origin” for the “new” function, if it is not really newly written, but an “old” function being moved to the current location.

Figure 3.2 shows an example that we can verify our dependency analysis. Function A in release v2.0 is “new” to the system. Now we need to find out if there it has an origin in the previous release v1.0.

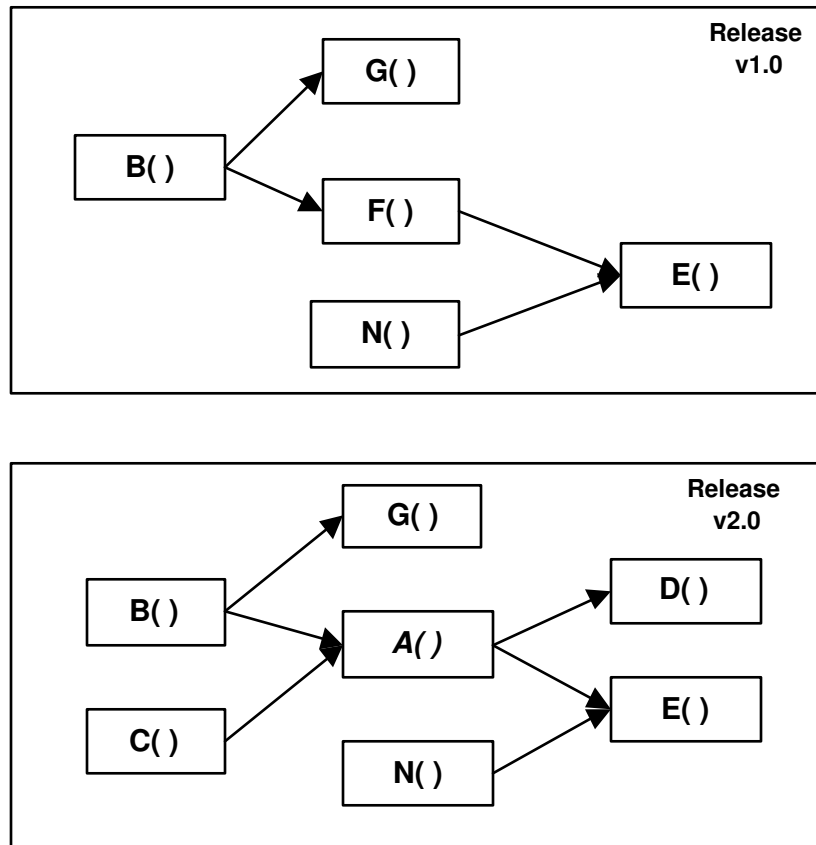


Figure 3.2: Example of Call-Relation Change Analysis

- *Caller Analysis*: Function A is called by Function B and C in v2.0. However, only B exists in both v2.0 and v1.0. So we will see how the callee list of B has been changed: B used to call G and F in v1.0, but in v2.0, it calls G and A. The difference is function F in v1.0 and we put this function in the candidate list.
- *Callee Analysis*: Function A calls function D and E in v2.0. Because D was not in v1.0, we only need to study E: E used to be called by F and N in v1.0, but it is called by A and N in v2.0. The difference is function F again, which agrees with the result from caller analysis.

After applying both caller analysis and callee analysis, we believe that the “new” function A in v2.0 has very close tie with an “old” function in v1.0, if they are not the same function at all.

3.3 BEAGLE: An Integrated Environment

To validate the research techniques we have just discussed, we have built an research platform called *BEAGLE*, that integrates several research methods for studying software evolution, including the use of evolution metrics, program visualization, and origin analysis for structural changes.

BEAGLE has a distributed architecture that reassembles a three-tier web application. Figure 3.3 illustrates the conceptual architecture of BEAGLE. At the backend, the evolution data repository stores history information of the software system. The data repository, together with the query-processing interface, forms the database tier. In the logic tier, comparison engine retrieves information from the database tier, and compare the differences between the selected releases from various perspectives. The origin analysis component performs the task to reveal the hidden relations between the program structures of difference releases. The visualization component generates the graphical representation of the software evolution data. The components in the logic tier receive user queries and send back query results through the user interface application running on clients’ machines, which forms the user tier. Users can also navigate the evolution data using tools from this tier.

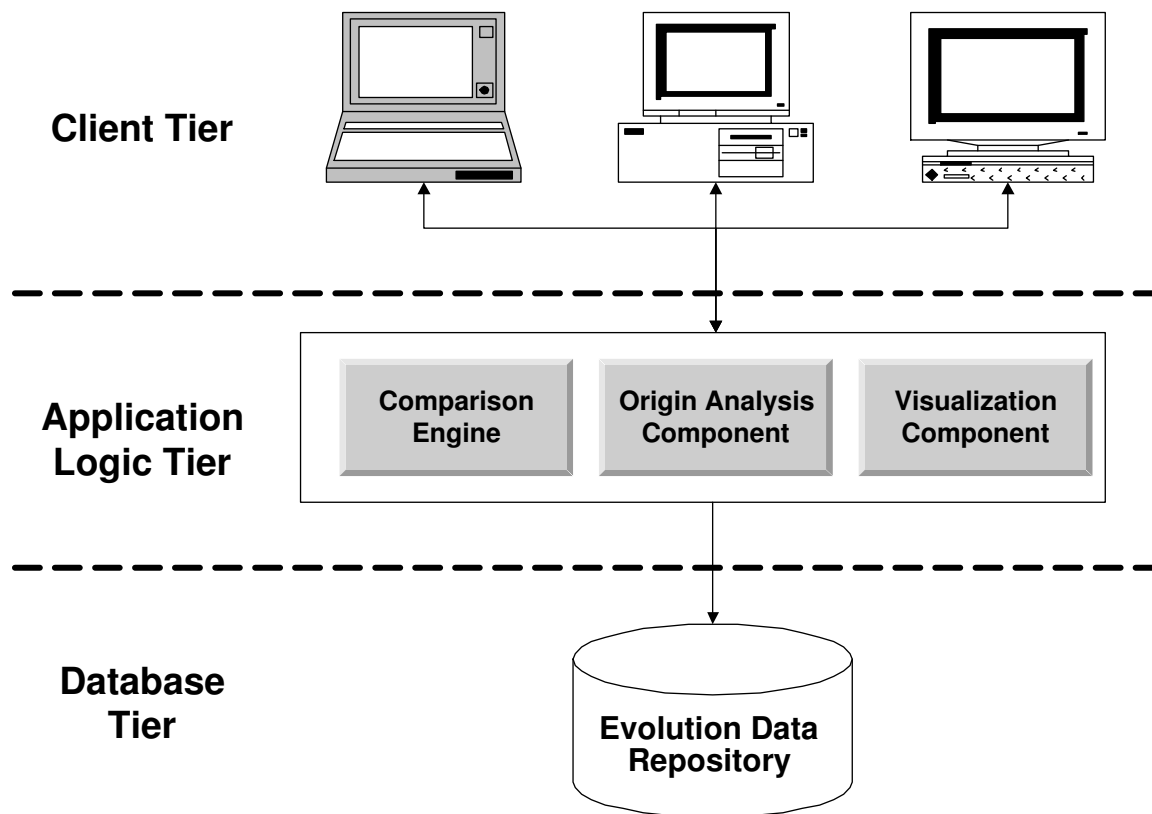


Figure 3.3: Conceptual Architecture of BEAGLE Environment

3.3.1 Database Tier

Like many information retrieval systems, BEAGLE is supported by a data repository that is implemented as a relational database. In the database, software architectural information of past releases, as well as metrics that describe the attributes of program entities are stored in the database, organized according to a *star schema*, which are described below.

Functional components in the logic tier access the information stored in the data repository through a query interface. In BEAGLE, the query interfaces are written in SQL, the standard relational database query language.

Data Repository Schema

In the repository, relational tables are organized according to a *star schema*. The *star schema* is a popular data model in database warehouse systems and multi-dimensional database systems. It is a query-centric model designed for static databases that store large amount of historical data, and supports time series analysis to discover historical patterns presented by the data and to forecast future trends.

In *star schema*, tables are arranged in the following ways. A central “fact” table is connected to a set of “dimension” tables, one per dimension. The name “star” comes from the usual diagrammatic depiction of this schema with the fact table in the center and each dimension table shown surrounding it [50].

The BEAGLE data repository has four fact tables. They model the system structure and relations between program entities at various abstract levels. The four levels of abstraction are: entity, file, high, and architecture. Each level of architecture fact is stored in its own table for all the history releases. Besides the different abstraction level, all four fact-tables have very similar structure.

1. *Entity-Level Facts* - A entity-level fact table stores the lowest level of architecture information that we model in BEAGLE: the dependencies between functions, global variables, and macros. It also stores the containment relations between basic program entities and files. We have used the source code extractor `cfx` to pull out such information from the source code in our examples.

2. *File-Level Facts* - This table is the abstraction of entity-level facts: it stores the relations between source files. File-level facts are induced from entity-level facts using relational calculus formulas defined in `grok` scripts in PBS. Ten types of relations are stored in this table: `call_body`, `call_ifc`, `call_lifc`, `call_noifc`, `dep_other`, `impl_proc`, `impl_var`, `ref_body`, `ref_ifc`, and `ref_lifc`.
3. *High-Level Facts* - Information stored in the table is further abstracted from file-level facts. Even though the main entities modeled in this table are still files, the relations between files are a set of higher-level relations that are merges from the intermediate relations modeled by file-level facts. We call these facts high-level to differentiate them from the file-level facts. The abstraction of relations between files removed cluster of dependencies by concentrating only on three simple dependency relations: *userproc*, *usevar* and *implementby*.
4. *Architecture-Level Facts*- The architecture-level fact table contains not only relations between program files, but also higher level architecture facts between file and subsystem, subsystem and subsystem, and also containment relations between files, low-level subsystems, and high-level subsystem. Subsystem is a group of related program files working together to provide a major functionality of the system. In BEAGLE, the grouping of program files into subsystems is performed manually with the help of source directory structure, domain knowledge, and design documents.

Figure 3.4 shows the relations between fact tables and six dimension tables, as well as the schema of each table.

Besides the fact tables, there are six dimension tables. They provide additional information for entities and dependencies modeled in the fact tables. Here is a list that explains each dimension table in detail:

1. The *version number* table stores the breakdown of the version number of each history release. For example, GCC 2.7.2.3 is broken into major release as two, minor release as seven, major bug-fix release as two, and minor bug-fix release as three. The series column is used to distinguish between the stable release stream and the experimental

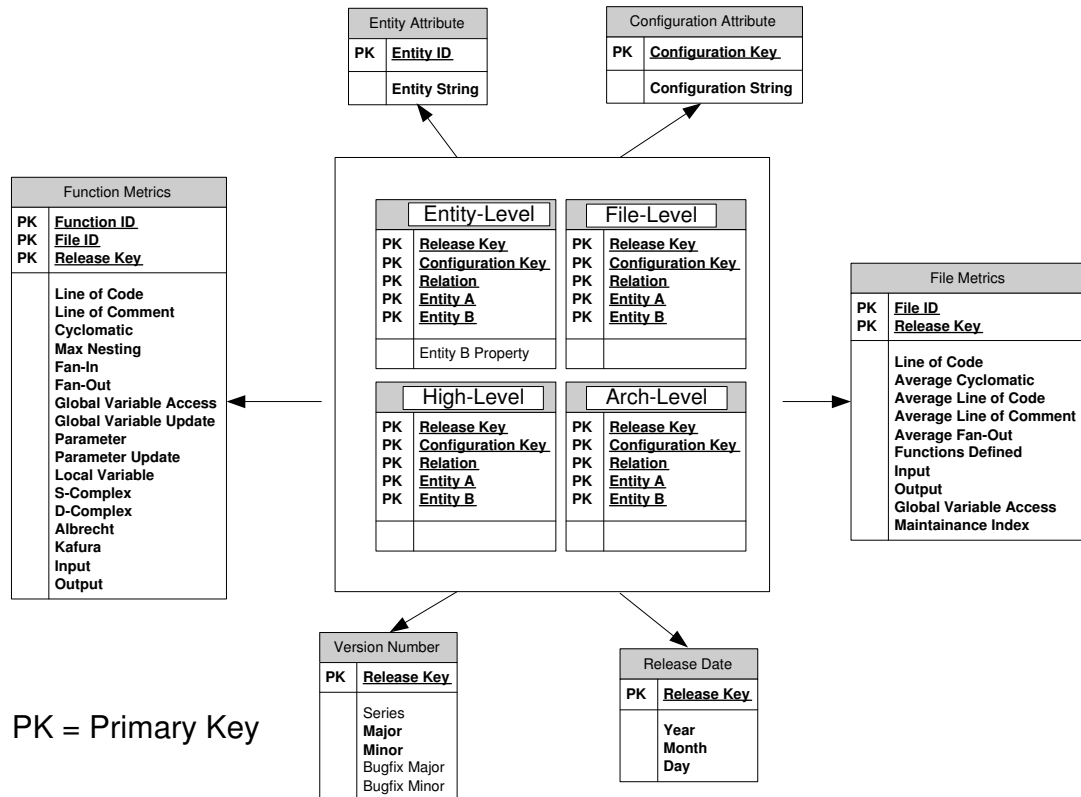


Figure 3.4: Schema of BEAGLE Data Repository

release stream. In GCC project, *GCC* is reserved for production releases, and *EGCS* is for experimental releases. In Linux kernel, the middle digit (minor release number) indicates whether the release is a production release (if even) or an experimental development release (if odd).

2. The *release date* table stores the release date of each history releases. It includes three columns: year, month, and day. The release date is used to calculate the time interval between consecutive releases, which we use as a rough indicator of development effort.
3. The *entity attribute* table maps the name of entities stored in fact tables to an integer value to save storage space, and improve the comparison performance. Applications can easily retrieve the real name of program entitles back by doing a lookup on this table.
4. The *configuration attribute* table extends the *configuration* column in fact tables. Many software systems support flexible building configurations. For example, GCC supports C, C++, Objective C, Chill, Fortran, and Java. It provides users an option to choose which compiler to be included in the build. In our case study, we build each release of GCC with two build options: *ONLY* for building a c only compiler, and *ALL* for building GCC compiler suite with all supported programming languages.
5. The *function complexity* table contains a select of code metric measurements targeted at the function level. Measured metrics include LOC, McCabe’s cyclomatic complexity, fan-in and fan-out. We also pre-compute and store four composite metrics: S-Complexity, D-Complexity, Albrecht, and Kafura [28]. We will use this metric information to act as a kind of “fingerprint” for the functions in “origin analysis”.
6. The *file complexity* table contains a set of metrics at the file level. Most metrics included in this table are basic complexity metrics. The last metric, *maintenance index*, measures the maintainability of a program source file as introduced in [38].

History Data Collection and Processing

The data in the fact tables are collected using PBS tools. The metric measurements are collected using a source code analyzing tool called *Understand for C++*, which is a reverse

engineering, documentation, and metrics tool for C and C++ source code [15]. The outputs from both tools are processed by a series of transformers we have written to transform them into the formats that conform to the BEAGLE repository schema.

The PBS outputs follow RSF (three-element tuple) and TA [23] formats. Both formats are very close to fact table schema, so the conversion process is straightforward. The situation is different for Understand for C++. Because Understand for C++ has its own internal data storage schema, much work needs to be done to translate the data generated by Understand for C++ from its own schema into BEAGLE repository schema.

Understand C++ generates two types of analysis reports. One is *Metrics Report*, which shows basic metric information for functions and files such as LOC, Cyclomatic complexity, Fan In, Fan Out, etc. Another report is the *Cross Reference Report*, which contains the following information:

- The *Object Cross Reference Report* lists all C/C++ objects, such as variables, parameter and macros along with declaration or usage references.
- The *Class or Type Cross Reference Report* lists all declared classes and types along with their declaration or usage information.
- The *Function Cross Reference Report* lists all C/C++ functions along with parameter list, return type, and reference information.

We can use Metrics Report directly to populate the two metric tables in BEAGLE repository for basic code metrics. For more complex metrics such as Albrecht and Kafura, we have to parse the Cross Reference Report output to rebuild the internal cross-reference database in memory. By walking through the internal cross-reference database, we can calculate all kinds of required interactions between functions and files to calculate composite code metrics. We have to do so because PBS does not provide detailed information at sub-function level. To build a complete architecture fact repository for all the history releases, we need to repeat the data collecting procedures for every archived release. Additional information regarding the release events, such as release data and release version number are also collected and used to populate various dimension tables in data repository. Figure 3.5 is a flowchart that illustrates data collection procedure.

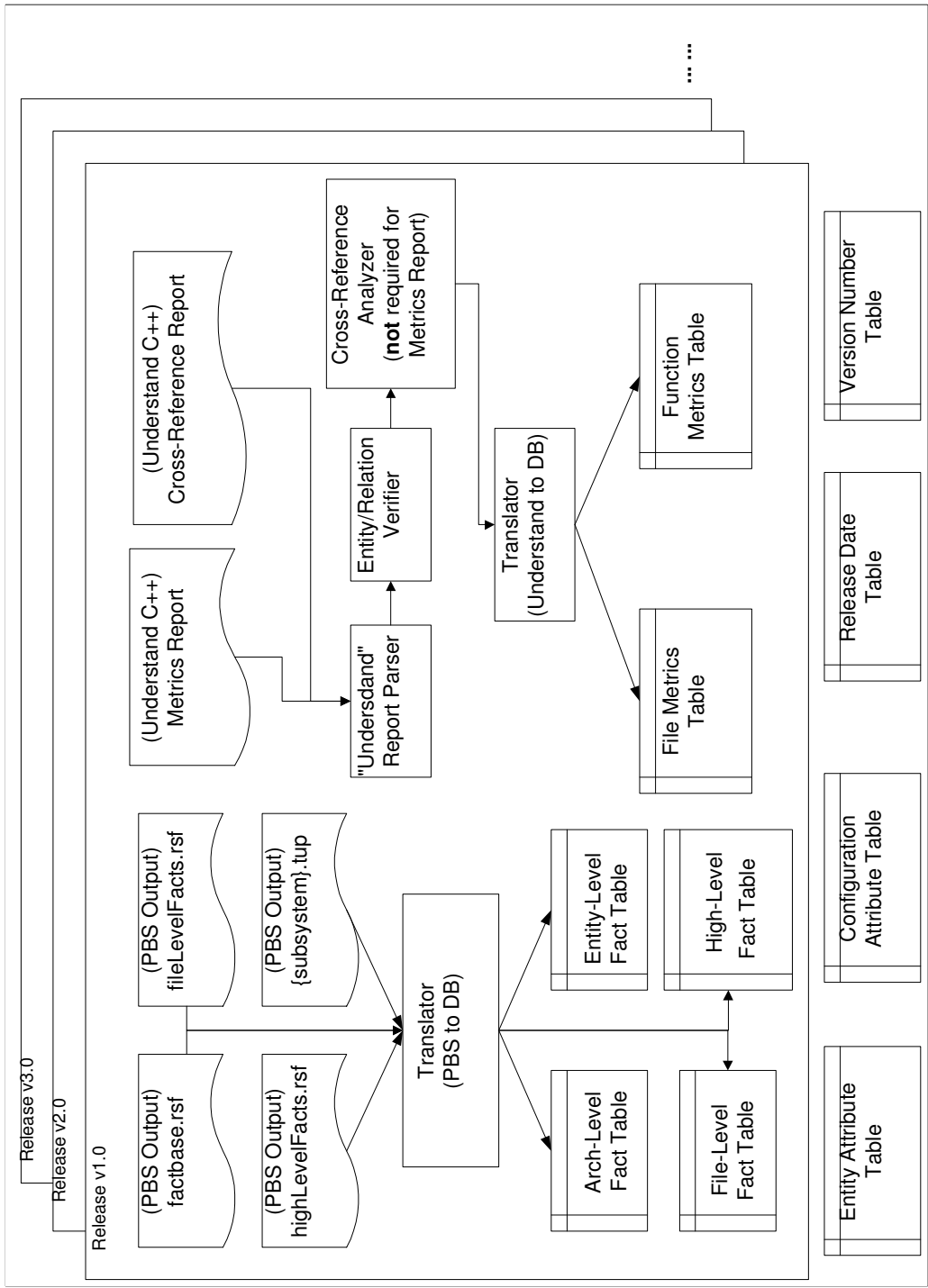


Figure 3.5: Procedures to Build Data Repository

Repository Access Interface and Comparison Query

Having all the history data in a relational database is the first step in building the data tier of BEAGLE. We must also provide a query facility for the repository so that all the functional components in the logic tier can access the data repository effectively, and to “slice and dice” the history data stored in the repository to investigate the patterns of software architecture evolution.

One of the benefits of choosing a relational database for implementing BEAGLE data repository over some proprietary data storage is that RDBM provides SQL (Structured Query Language) as the standard query interface for easy and flexible data access. SQL is a powerful query language that is able to express almost all the queries that users want to issue to the history data repository, for analyzing software architecture evolution attributes and patterns.

We present two example queries to illustrate the querying interface and working mechanism of BEAGLE comparison engine.

Example Query: Change of Architecture Entities

Our first task is to find the set of all functions that were newly defined in version v2 (*i.e.* were not present in version v1). We also want to find out all the files in which new functions are defined, as well as the LOC and Kafura metrics for all the new functions. Here is the SQL statement that carry implement this task:

```
SELECT Func_Name.entity_string AS Function,
       File_Name.entity_string AS File,
       Metrics.line_of_code AS LOC,
       Metrics.Kafura AS Kafura
FROM Entity_Attribute AS Func_Name,
     Entity_Attribute AS File_Name,
     Function_metrics AS Metrics
WHERE Func_Name.entity_id = Metrics.function_id
      and File_Name.entity_id = Metrics.file_id
      and Metrics.release_key = v2
      and (Metrics.function_id, Metrics.file_id) IN (
```

```

        SELECT function_id, file_id
        FROM Function_Metrics
        WHERE release_key = v2
    EXCEPT
        SELECT function_id, file_id
        FROM Function_Metrics
        WHERE release_key = v1 )

```

This SQL statement uses two data tables from the repository: **Function Metrics** and **Entity Attribute**. It selects those rows in the **Function Metrics** table with release key equals to **v2**, plus condition that the function key and file key exists in version **v2**, but not in version **v1**. Then it refers to the **Entity Attribute** table to convert the integer key back to entity name string.

Figure 3.6 shows a section of the output of the above query. We are comparing GCC 2.7.2.3 and GCC 2.8.0.

Example Query: Change of Architecture Relations

Our second task is to compare version **v3** with **v2** (both under build configuration **c1**), and show all the new relations between files within subsystem **s1**, where the caller is 'old' (exists in both **v2** and **v3**), but the file being called is new (only exists in **v3**, not in **v2**). Here is the SQL statement to carry out the query:

```

SELECT Caller_File.entity_string AS Caller,
       Callee_File.entity_string AS Callee
FROM SS_Fact,
     Entity_Attribute AS Caller_File,
     Entity_Attribute AS Callee_File
WHERE SS_Fact.relation = "useproc"
      and SS_Fact.entity_a = Caller_File.entity_id
      and SS_Fact.entity_b = Callee_File.entity_id
      and SS_Fact.release_key = v3
      and SS_Fact.configuration_key = c1
      and SS_Fact.entity_a IN (

```

FUNCTION	FILE	LOC	KAFURA
save_constants_in_decl_trees	integrate.c	11	10
insn_cuid	combine.c	11	88
rtl_equal_for_field_assignment_p	combine.c	33	120
sets_function_arg_p	combine.c	27	12
merge_assigned_reloads	reload1.c	45	40
reload_cse_check_clobber	reload1.c	8	4
reload_cse_invalidate_mem	reload1.c	23	12
reload_cse_invalidate_regno	reload1.c	55	66
reload_cse_invalidate_rtx	reload1.c	15	21
reload_cse_mem_conflict_p	reload1.c	49	27
reload_cse_noop_set_p	reload1.c	63	120
reload_cse_record_set	reload1.c	101	360
reload_cse_regno_equal_p	reload1.c	22	9
reload_cse_simplify_set	reload1.c	40	100
reload_cse_regs	reload1.c	119	527
reload_cse_delete_death_notes	reload1.c	24	36
reload_cse_no_longer_dead	reload1.c	12	7
reload_cse_simplify_operands	reload1.c	156	480
free_regset_vector	flow.c	9	7
print_rtl_with_bb	flow.c	86	50
find_valid_class	reload.c	24	0
exception_section	varasm.c	16	1
mark_constant_pool	varasm.c	18	9
mark_constants	varasm.c	50	36
output_after_function_constants	varasm.c	12	6
asm_output_aligned_bss	varasm.c	18	55
bss_section	varasm.c	19	2
eh_frame_section	varasm.c	9	1
in_data_section	varasm.c	5	0

Figure 3.6: Result of the first example query

```
        SELECT entity_b
        FROM SS_Fact
        WHERE entity_a = s1
        AND relation = "contain"
        AND release_key = v3
        AND configuration_key = c1
    INTERSECT
        SELECT entity_b
        FROM SS_Fact
        WHERE entity_a = s1
        AND relation = "contain"
        AND release_key = v2
        AND configuration_key = c1 )
and SS_Fact.entity_b IN (
    SELECT entity_b
    FROM SS_Fact
    WHERE entity_a = s1
    AND relation = "contain"
    AND release_key = v3
    AND configuration_key = c1
EXCEPT
    SELECT entity_b
    FROM SS_Fact
    WHERE entity_a = s1
    AND relation = "contain"
    AND release_key = v2
    AND configuration_key = c1 )
AND (SS_Fact.entity_a, SS_Fact.entity_b) IN (
    SELECT entity_a, entity_b
    FROM SS_Fact
    WHERE relation = "useproc"
    AND release_key = v3
    AND configuration_key = C1
```

```

EXCEPT
    SELECT entity_a, entity_b from SS_Fact
    WHERE relation = "useproc"
    AND release_key = v2
    AND configuration_key = C1 )

```

This example is more complicated than the previous one, so we will explain in more detail. Since we do not need any metric information, so only **subsystem-level fact** table and **entity attribute** tables are accessed in the query. As introduced in the previous section, subsystem-level fact table contains facts related to function call relations, data reference relations, and implementation relations between files and subsystems.

The SQL statement first selects rows from the subsystem-level fact table where the release key is v3, configuration key is c1, and most important, the relation between two entities must be “useproc”, which means **call relation** between file or subsystem entities. Then it continues to specify the caller file, callee file and the call relations with **SELECT** clause.

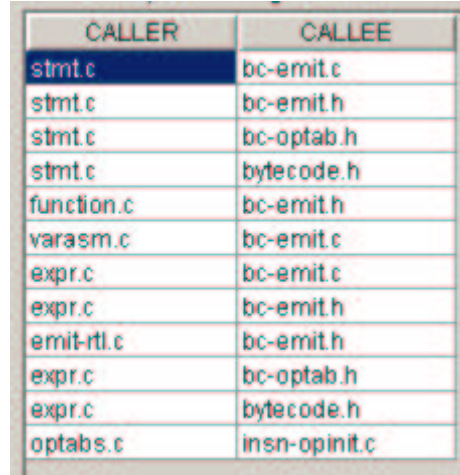
The first sub-clause in the SQL statement puts constraint on the caller file. In the first **SELECT** statement of the sub-clause, it selects those files contained in subsystem s1 in release v3. The second **SELECT** statement chooses those files contained in subsystem s1, but in release v2. The **INTERSECT** operator makes sure that the selected caller files exists in both release, so that they are qualified for being “old” callers.

The second sub-clause constrains the file being called. In the first **SELECT** statement of the sub-clause, it selects those files contained in subsystem s1 in release v3. The second **SELECT** statement chooses those files contained also in subsystem s1, but in release V2. The **EXCEPT** operator ensure that the select callee files exists only in release v3, but not in v2, so that they are qualified for being “new” callees.

The last sub-clause limit the call relations to be “new”. It again uses the **EXCEPT** operator to choose those “useproc” relations that exist in release v3, but not in v2.

Similar to the SQL statement in the first example, the original file name strings are converted back from integer keys by referring to the **entity attribute** table. The result will be a list of (caller file, callee file) relation pairs that satisfy the query criteria.

Figure 3.7 shows the output of this query. The two releases compared by the query are GCC 2.8.0 and GCC 2.3.3. The focused subsystem is “RTL Generator”.



CALLER	CALLEE
stmt.c	bc-emit.c
stmt.c	bc-emit.h
stmt.c	bc-optab.h
stmt.c	bytecode.h
function.c	bc-emit.h
varasm.c	bc-emit.c
expr.c	bc-emit.c
expr.c	bc-emit.h
emit-rtl.c	bc-emit.h
expr.c	bc-optab.h
expr.c	bytecode.h
optabs.c	insn-opinit.c

Figure 3.7: Result of the second example query

3.3.2 Application Logic Tier

The core functionalities of BEAGLE are provided by components in the application logic tier. They are *version comparison engine*, *origin analysis component* and *evolution visualization component*.

Version Comparison and Evolution Visualization

In BEAGLE, we adopt a novel approach to visualize the difference between various releases. Figure 3.8 shows the screen shot of BEAGLE visualizing the architecture differences between GCC version 2.0 and GCC version 2.7.2.

The tree structure in the left panel of the window shows the system structure of GCC version 2.7.2. Items shown in *folder* icons are subsystems. It contains files, which is shown in *Document* icon. Under file, there are items that represent functions defined within the source file. Functions are shown in *block* icons. User can click on an icon, and the system

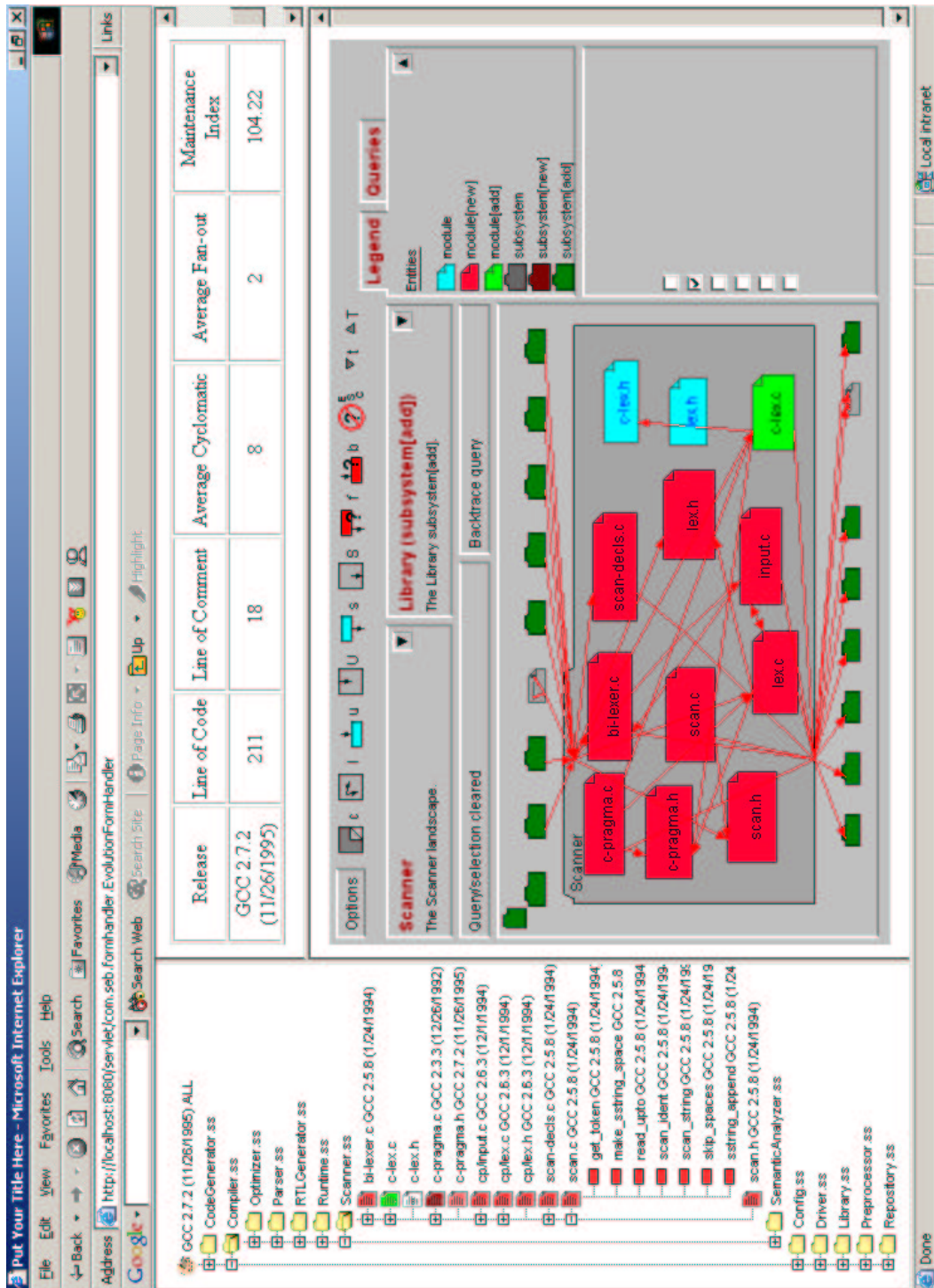


Figure 3.8: Screen Shot of BEAGLE Architecture Comparison: GCC 2.0 vs GCC 2.7.2

structure tree will automatically expand to show entities under the selected subsystem or file.

In BEAGLE’s evolution visualization, colors are used extensively to model the evolution status of individual program entities:

- *Red* represents entities that are “new” to the release. Since we chose to visualize the architecture differences between GCC v2.0 and v2.7.2, any entities including subsystems, files, or functions in v2.7.2, but were not in v2.0 are treated as “new”, thus are tagged with red icons.
- *Blue* indicates program entities that were originally in v2.0, but are missing from v2.7.2.
- *Green* indicates that parent-level entities, such as subsystems and files, contain either “new” entities or have entities deleted from them. We choose green color because it presents life and changes. If the none of the contained entitles ever changed, this will be indicated by *white*.
- *Cyan* icons are for functions that exist in both version 2.0 and version 2.7.2.

For program entities that are “new” to GCC version 2.7.2, different “reds” with various levels of saturation are used to differentiate their “tenure” within the system. An entity in vivid red came into the system relatively late, while darker red means that entity has been in the system for several releases.

At the left bottom of figure 3.8 we can see nine new files under “Scanner” subsystem. `c-pragma.c` first appeared in GCC at version 2.3.3. It is one of the oldest among all nine files, so its red is the darkest. `c-pragma.h` first appears in GCC at version 2.7.2, which means it is the youngest. Thus its color is very fresh red. File `cp/Input.c` first seen in GCC at version 2.6.3. It is later than `c-pragma.c` but earlier than `c-pragma.h`. As the result, its icon has a red color with saturation somewhere in the middle.

The frame on the right side of figure 3.8 shows another style of software evolution visualization. It is based on the landscape viewer used in PBS. It extends PBS’s schema by adding evolution related entities and relations. Six new entities are added to model *new subsystem*, *delete subsystem*, *changed subsystem*, *new file*, *delete file* and *changed file*.

Also there are six new relations: *new call*, *delete call*, *new reference*, *delete reference*, *new implemented-by*, and *delete implemented-by*.

If user chooses a newer release as the reference release, and want to see how the software architecture has been changed since a specified earlier release, then the evolution visualizer will display all the “new” entities, “changed” entities and “unchanged” entities, along with “new” relations and “unchanged” relations. “New” means the entity only exists in the newer reference release. “Changed” means the subsystem or file exists in both the reference release and earlier release, but it contains “new” modules or functions within it. Since function is the most basic program entity in BEAGLE, it only has two evolutions status: “new”, or “unchanged”.

If user chooses an older release as the reference release, and want to compare its software architecture with a newer release, then the evolution visualization will display all deleted, changed, and unchanged entities. It also shows delete, changed, and unchanged relations. “Deleted” means the entity exists only in the earlier release, but not in the newer release. “Changed” means that subsystem or file contains modules or functions that are no longer in the newer release. Since function is the most basic program entity in BEAGLE, it only has two evolutions status: “unchanged”, or “deleted”.

Origin Analysis

In BEAGLE, we apply both Bertillonage analysis and dependency analysis to exam every “new” functions in the selected release with its immediate previous release to find out its “origin”, and exam all the “delete” functions with its immediate next release to find out its “destination”. Under some cases, source files will be moved to new locations in the later releases, most time to new directories, as a maintenance effort to reorganize the source directory structure. In other cases, related source files are given common prefix or suffix in their file names for easier understanding of their responsibility in the system. Even though the file content does not change, many files will have a new name after the new naming scheme is adopted.

These types of changes to file path and file name make traditional architectural comparison tools such as GASE and KAC ineffective, because they treat a file with a different path or name a very different file. The result will be too many “new” files identified in the

newer release. Our solution to avoid this kind of chaos is to apply Bertillonage analysis on every function defined in the “new” file. If the majority of the functions have “origin” functions that are from the same file in the previous release, we can imply that this file is the “origin” file of the selected “new” file. Another solution is to perform call dependency analysis at file level. Instead of checking the “callee list” change of caller functions and “caller list” change of callee functions, as in the call dependency analysis performed at function level, we examine the “callee list” change of files that have call dependencies with this “new” file, or the “callee list” changes of those files that this “new” files has call dependencies with. The result is the potential “origin” file for the selected “new” file.

3.3.3 User Tier

Users interact with BEAGLE through *user tier* components. These components handle user input and submit queries to the *logic tier*, then organize and display the results on the screen. Here use a simple example to illustrate the interaction between BEAGLE user interface and a user. The software system under investigation is GNU C Compiler.

Please choose two or more GCC/EGCS historical releases to compare.

☐ Compare **Two Releases**: First Release Second Release

☒ Compare **Multiple Releases**:

Choose A Build Configuration:

Show the Software Architecture of ☒ **Newest Release** ☐ **Oldest Release**

Figure 3.9: User Interface for Entering Query Options

Initially, a list of history releases of GCC are displayed in a web page, along with short description for each release, such as the full release number and release date. A user can select any two releases or a group of consecutive releases over a period, and then request an architecture comparison, as shown in Figure 3.9. The user interface component will respond to the user's request by sending a message to *version comparison engine* in the *logic tier*. When the comparison is finished, the results are passed to the *evolution visualization component*, where the difference between the two software architectures are converted to graphical diagrams along with other detailed change information about individual program entities. Finally, the diagrams and other attributes are send back to the *landscape viewer* component in the *user tier* for display and further navigation, as shown in Figure 3.8.

3.4 Conclusion

We have introduced an interactive, web-based integrated approach to study software evolution, especially architectural and structural changes. The data source we selected for study is the architecture facts extracted from program source code, with additional information on evolution metrics, release details, and revision control data. All the history data is stored in a relations database and organized according to star schema. Queries to the evolution data are implemented in SQL statements. The query results are displayed in a web browser as visualized evolution graphs and tables of evolution metrics. Users can navigate the evolution data as usual WWW pages. The evolution of software structure is studied using origin analysis methods. The purpose of this analysis is to reveal the hidden relationships between program entities in the more recent release with those from the earlier release as the results of system re-architecture. We present two methods for origin analysis. One method compares the feature sets of functions from both releases to find the possible match. The other method analyzes the changes of call relations between suspected functions and their dependents. When used together, these two methods are able to provide plausible results.

In the next chapter, we verify the effectiveness of our approach by examining the evolution history of GCC, a large open source system with a long development history using *BEAGLE*.

Chapter 4

Architectural Evolution of GCC: A Case Study

In the previous chapter, we discussed the main ideas we have developed to browse and analyze software evolution, with particular emphasis on the evolution at the architecture level. We also described an integrated platform *BEAGLE* that implemented these techniques. In this chapter, we will use the evolution history of the GNU Compiler Collection (GCC) project as an example to demonstrate how one may use BEAGLE to explore the evolutionary history of a large software system.

This chapter begins with a brief description of GCC and its development history. Then we demonstrate how BEAGLE can aid in answering various detailed questions about its evolution. We have chosen questions that a new developer might ask in trying to come to an understanding to the software architecture of GCC and its evolution.

4.1 Background and History of GCC Project

4.1.1 Origin of GCC

The GNU Compiler Collection (GCC) was originally developed as a compiler for the C language (`gcc`) by Richard Stallman, the founder of the GNU and Free Software Foundation. The first version of GCC was released in June 1987. It consisted of 110,000 lines of C code,

initially supporting two target platforms: VAX and Sun3 Workstation. It compiled only C code at that time. The original design goal of GCC was to create a portable optimizing compiler that supported diverse CPU architectures and multiple programming languages [47], and one that has remained throughout its lifespan. GCC is flexible to be extended to support other programming language and platform¹.

GCC version 1.x was developed and maintained by Richard Stallman and a few enthusiasts from the GNU project [11]. The software is copyrighted and distributed under GNU GPL (General Public License), which requires the redistribution of the compiler and its source code to be free.

4.1.2 GCC 2.0 and Cygnus

In the early 1990, GCC was facing a major challenge. While GCC version 1.x performed well on CISC machines such as DEC VAX and Intel i386, extra optimization effort was needed to support newly emerging RISC platforms, which require much more complex instruction scheduling mechanisms. Michael Tiemann wrote [11], “With the world transitioning from CISC to RISC, we went from having hands-down the best compiler in almost every regard to a more complex set of tradeoffs the customer would have to evaluate. It was no longer a simple, straightforward sell.”

Another challenge came from supporting the C++ language. The GNU C++ compiler started as a separate project in the fall of 1987. Although its code was originally based on GCC, the development of GNU C++ fell behind GCC in terms of stability, as C++ is a much more complex language than C. Furthermore, the design of the C++ language was still evolving throughout late 1980s and most of the 1990s. New and complex features, such as templates, were continually being introduced to the “draft” standard. It became obvious that the old development model (*i.e.*, maintenance by a small group of enthusiasts) of GNU was not practical.

To keep the GNU C and C++ compiler projects moving forward and competitive, various changes were made to the development model of GCC. Cygnus, which used to profit by distributing GCC software and providing porting services, teamed with FSF to

¹Someone was able to port GCC to a new CPU (the 32032 from National Semiconductor) in just two weeks, and still got performance that was 20 percent faster than NS’s proprietary compiler [11]

develop GCC version 2. FSF still kept the “steering wheel” of GCC, which controls the direction in which GCC should go and how it should be built. In the pre-web age, this was an effective development model for open source software to obtain necessary resources and commitment. Cygnus contributed most of the key developers of GCC, and in the meanwhile making money by selling a value-added product line based on GCC and other GNU tools, as well as providing porting and maintenance service.

GCC version 2.x, which was released in February 1992, bundled compilers for C, C++ and Objective-C into one package. GNU C++ was no longer a separate project: as it fully merged within GCC. GCC 2.0 was able to generate object code for 19 different CPU architectures, compared to only 13 in GCC 1.42. Most newly supported architectures were for RISC machines, such as the HP 9000/800x series and IBM RS6000. The new version also had more effective optimization and scheduling algorithms.

4.1.3 EGCS and Web-based Software Development

GCC 2.x is not perfect. Its support for “templates”, as introduced in 1998 ANSI C++ standard, was very poor both in completeness of functionality and efficiency, due to the limitation in its software architecture design. The STL (standard template library) implementation that was based on the design from HP is inferior to the one from SGI (another popular STL implementation). “Exceptions” in C++ are implemented in g++ without much optimization. Many innovations in instruction scheduler design and code optimization algorithms that emerged during the 1990s have yet to be incorporated into GCC source code.

Along with the technical issues, there was also tension between various GCC developers. Traditionally, GCC had been tightly controlled by FSF with respect to issues such as which new features should be added and how the architecture should be modified for the next release. FSF tended to be conservative about adding new features. Because GCC is the system compiler for all GNU projects, stability was top priority for them. FSF’s conservative altitude towards the evolution of GCC also resulted in the long product release cycles, an average one year for each new release (even for bug-fix releases)

On the other hand, with the increasing popularity of GCC and web-based cooperative development models (pioneered by Linux kernel), more and more people with diverse inter-

ests have become involved in the development of GCC. Each group has different interests in the direction of GCC development. Some focus on the optimization for a particular architecture, such as the Pentium; some wish to include a Fortran front-end or a new C++ library into GCC; some want to port GCC onto embedded devices; and some just want to try out the newest instruction scheduler from the IBM research lab. The diverse collection of streams of GCC development slowed down the overall process and caused tension between zealous developers and the conservative “steering committee”.

To handle the situation, a group of developers decided to start a more experimental development project, based on GCC but running as a parallel development system to the traditional GCC project. This project was named EGCS (Experimental GNU Compiler Systems). The development model for EGCS is more “open” and collaborative. It allows developers all over the world to have an opportunity to contribute to the project [11]. Similar to Linux kernel, EGCS has a very short release cycle.

The benefit of having two projects active at the same time is obvious. New features and improved hardware architecture support could be tested in EGCS without hurting the stability of GCC. When a feature is debugged thoroughly in EGCS and proven stable enough, or a bug found in the old GCC code base, they are passed to the GCC maintainer at FSF immediately, and vice versa. Linux kernel project has similar development model that maintains the stable releases and development releases in parallel.

The EGCS project made its first release, EGCS 1.0, in August 1997. Until March 1999, seven versions had been released (1.0.x and 1.1.x). During the same time, GCC published their 2.7.x and 2.8.x releases.

After EGCS releases had been widely accepted by the software development community for over two years and proven to be a reliable system, a historical moment occurred in April 1999. The Free Software Foundation officially halted development on the GCC 2.8.x compiler and appointed the EGCS project as the official GCC maintainers. Also the meaning of GCC is changed to be the abbreviation of “GNU Compiler Collection”. The most up-to-date GCC version at the data collection time of this thesis was 2.95.2 released on October 24, 1999)².

²Version 3.0 was released on June 2001, after the work for this thesis had been completed

4.2 Common Software Architecture of GCC Releases

4.2.1 Reference Architecture of Compilers

A compiler is a program that processes a set of statements written in a particular source programming language, and translates it into machine language that a computer processor can execute. A compiler is comprised of four essential components: a scanner, a parser, a semantic analyzer, a code generator and optimizer [2].

Conceptually, a compiler operates in phases, each of which transforms the source program from one form of representation into another. Those phases as shown in figure 4.1, are often grouped into a “front-end” and a “back-end”. The front-end consists of phases that depend primarily on the source language and are largely independent of the target machine. This includes lexical and syntactic analysis, the creation of the symbol table, semantic analysis, and the generation of intermediate code. The front-end can perform a certain amount of code optimization as well. The front end also includes the error handling functionality that goes along with each of these phases.

The back-end includes the phases that depend heavily on the hardware architecture of the target machine. Generally, the back-end does not depend on the source language, but instead on the specification of the intermediate language and the architecture of the target hardware. The back-end normally includes code optimization and code generation, together with necessary error handling and symbol table operations.

It has been a common practice to take the same front-end of a compiler and rewrite its associated back end to create new compilers that runs on different machines. For example, IBM VisualAge Smalltalk product family contains versions for many platforms including Windows, OS2, AIX, Solaris, Netware, HP-UX, and Linux. It is also popular to compile several different languages into one common intermediate language, and then reuse the same back-end for the particular target. Software architects of a compiler system need to exercise careful design to balance the interfaces and dependencies between front-end and back-end to ease compiler porting

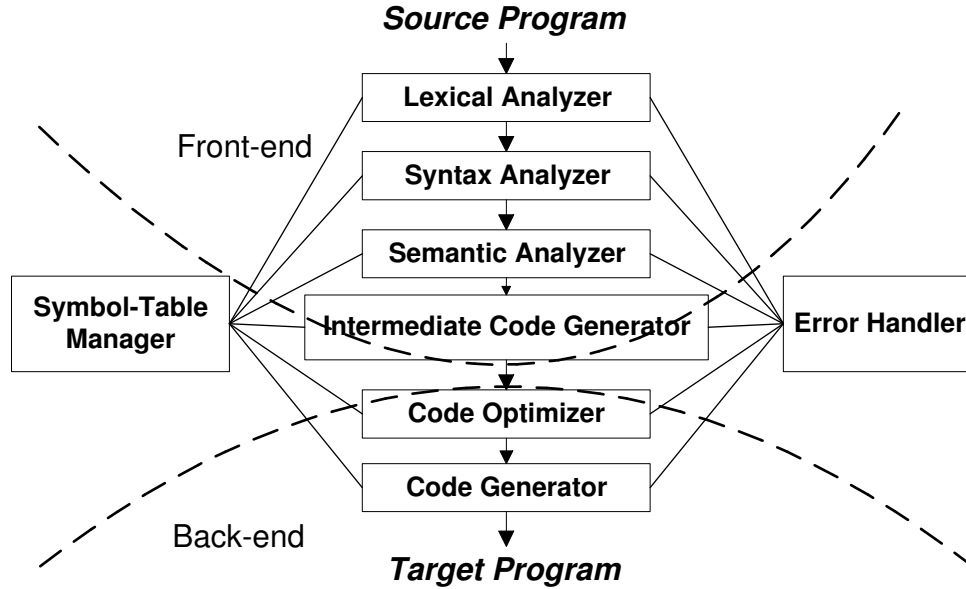


Figure 4.1: Components and Phases of Compiler

4.2.2 GCC Conceptual Architecture

The conceptual architecture of a software system is the software designer's mental model of the overall system structure, including the decomposition of the system into subsystems, and the dependencies between subsystems. The conceptual architecture provides a suggested or idealized system structure to help us understand the enormous information provided by the program source and extracted low level architectural facts [7]. Conceptual models are usually created using the following information: directory structure and grouping of file names, graph layout, related documentation, build process, organization structure of project group, and source code comments.

We used the discussion of modern programming language compiler [2] and existing GCC documentation [47] to create the following conceptual architecture for a modern portable multi-target multi-language compiler, in figure 4.2. The major components of our GCC conceptual architecture include:

- *Driver* is mainly an interface between GCC and the user. It also coordinates the

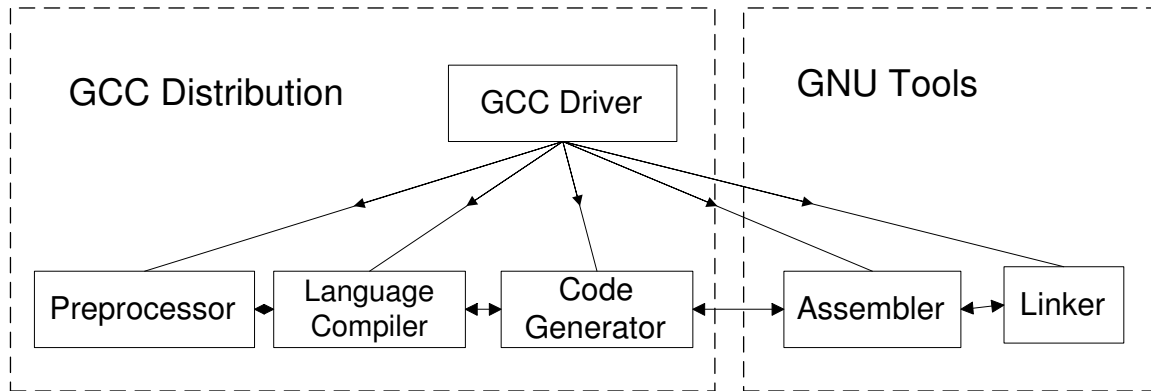


Figure 4.2: Conceptual Architecture of GCC and its Components

execution of various compilation phases within GCC (and later, outside GCC). It performs the following tasks:

- Interprets the command line parameters.
 - Determines language type based on file name suffixes, then chooses appropriate language compiler and utility program to run, the parameters to run with, and initiates execution of the compile.
 - Converts command line parameters according to a formal specification language called “specs”. The “specs” language defines rules such as: if gcc is called with option ‘-x’, then call the compiler or utility program with option ‘-y’. This effectively creates a unified entry point for all language compilers in GCC family.
- *Preprocessor* implements the preprocessor directives, such as `include` and `macro`. It also removes comments. The result is clean source code with line-numbering directives, which the rest of GCC subsystems may use in warning and error messages.
 - *Language Compiler* includes both the language front-ends and a part of the target machine back-end. It performs lexical analysis, syntactic analysis, semantic analysis, generating intermediate code as well as some optimizations at the RTL level.

- *Code Generator* translates the intermediate code into assembly code for the target machine.
- *Assembler* is not part of the GCC distribution, but is used by GCC driver “gcc”. It produces relocatable machine code that can be passed directly to linker.
- *Linker* is also not part of the GCC distribution, but is used by GCC driver “gcc”. It specifies all object files, the location of libraries and links program.

GCC is a large software system with half a million lines of commented code (version 2.7.2.3). In this section, we concentrate on the architecture of Language Compiler subsystem, whose conceptual architecture is shown in figure 4.3.

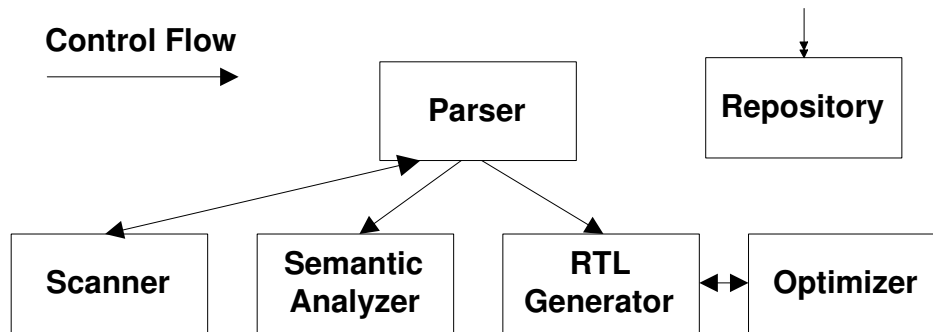


Figure 4.3: Conceptual Architecture of Language Compiler

There are six lower-level subsystems inside the Language Compiler subsystem. The scanner reads the input file from the preprocessor as a string of characters, and recognizes a stream of words and symbols, called tokens. The tokens output by the scanner are input to the parser, which recognizes the phrase structure of the source language and builds an abstract syntax tree (AST) to pass on to the semantic analyzer. Semantic analyzer adds attributes to the AST nodes according to the semantic analysis result. Then the AST is passed on to the RTL generator. The intermediate language represented in RTL format will go through various level of optimization by the optimizer before finally beginning emitted from Language Compiler subsystem to Code Generator subsystem. Data

structures and related operations that implement token, AST, and RTL are put in the repository subsystem.

In this chapter, we refer to software architecture as the organization of software system with program entities such as subsystem, file, and function. It also describes the dependencies (control flow, data reference, and function declaration and definition) between these entities.

4.2.3 Concrete Architecture of GCC

The Concrete architecture shows the implementation model of the system structure provided from software reverse engineering tools and human interpretation.

The concrete architecture of GCC is created in the following steps. First we extract architecture facts from the source code. Then we abstract the lower level facts to the architecture level. Eventually the implementation model is mapped to the conceptual model so that we can compare the similarities and differences between the designer’s mental model and the actual system implementation.

The concrete architecture of GCC version 2.7.2.3 shown in figure 4.4 was generated using the software comprehension tool suite PBS. It shows the “calls” relations and “data reference” relations between the system components of GCC in its implementation.

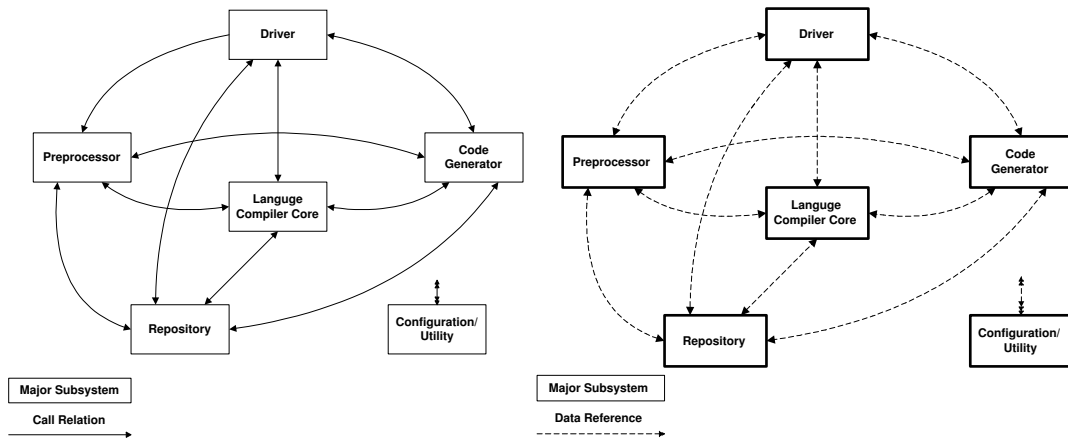


Figure 4.4: Concrete Architecture of GCC

The concrete architecture of the Language Compiler subsystem is shown as a call relation graph in figure 4.5, and a data reference relation graph in 4.6.

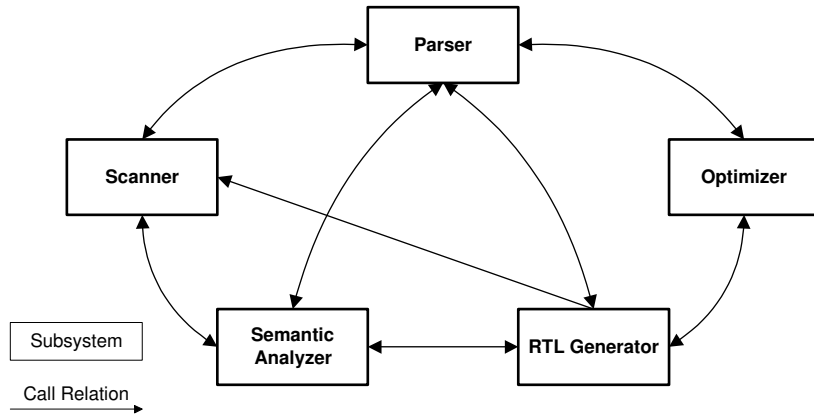


Figure 4.5: Concrete Architecture of Language Compiler - Call Relation

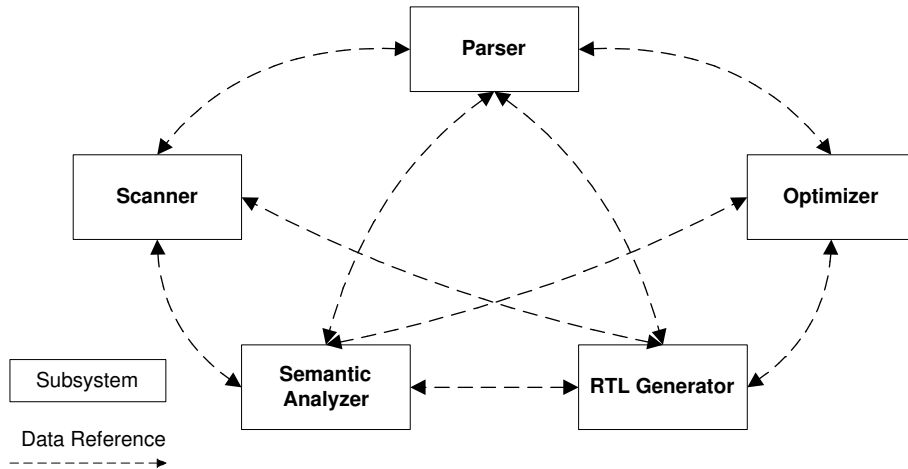


Figure 4.6: Concrete Architecture of Language Compiler - Data Reference

4.3 Related Research Work on GCC

GCC has been used as a case study in several research papers, especially by the SWAG group at the University of Waterloo. In this section, we will summarize some of this research, which covers the GCC source code size growth, its build-time behavior, and its maintenance evolution using dominance tree. We discuss these research works as they aid in understanding how GCC has evolved over time.

4.3.1 GCC System Size Growth

Godfrey and Tu studies the system growth history of GCC over 10 years of releases [52]. The major result is that the growth of GCC is increased by steps as shown in Figure 4.7. Within the same project branch, the growth is smooth and slow. However, between project branches, for example, between GCC 1.x and GCC 2.x, or between Gcc 2.x and EGCS 1.x, the size increased dramatically. There are also releases from different branches that overlap in their releases date. For example, the last GCC 1.x release, 1.42 was released several months after the official release of GCC 2.0. GCC 2.8.x was also released at the same time as EGCS releases.

This means that several GCC releases were developed at the same time. This finding corresponds to our review of GCC development history. Release are maintained mainly for stability and bug fixing within release branches, while new architecture are experienced by creating a new release branch such as GCC 2.x and EGCS.

This finding also contrasts with the faster growth rate of some other open source systems such as Linux kernel and VIM text editor. The development of Linux kernel and VIM adopt a more de-centralized collaborative approach, where one person acts as the project coordinator (Linus Torvalds for Linux kernel and Bram Moolenaar for VIM), and many other developers from all over the world are contributing code to the system continuously. As the result, the time intervals between new releases for these two projects are very short. On the contrary, GCC has adopted a more conservative development model. The key developers of GCC are all from Cygnus, and the project is coordinated by FSF. Outside development concentrates mostly on bug finding and compiler porting. This conservative approach attributes to the slower system growth rate of GCC within the same release

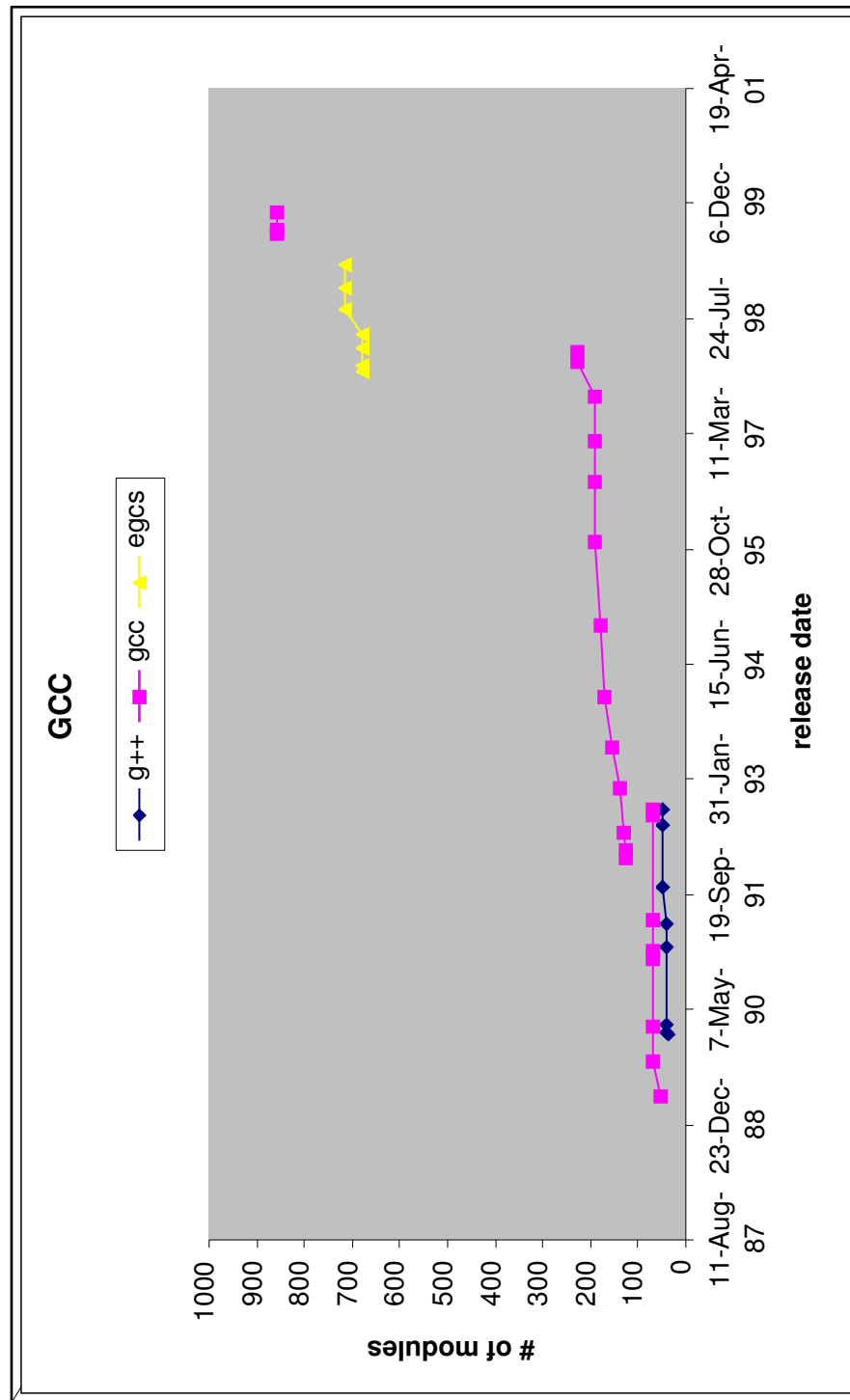


Figure 4.7: System Growth of GCC Releases

stream. On the other hand, the parallel developing nature of open source system attributes to the sudden growth of GCC system size between different release streams.

4.3.2 GCC Build-Time Behaviors

GCC exhibits interesting building behavior, including bootstrapping and build-time code generation, as discussed by Tu and Godfrey [53].

Bootstrapping

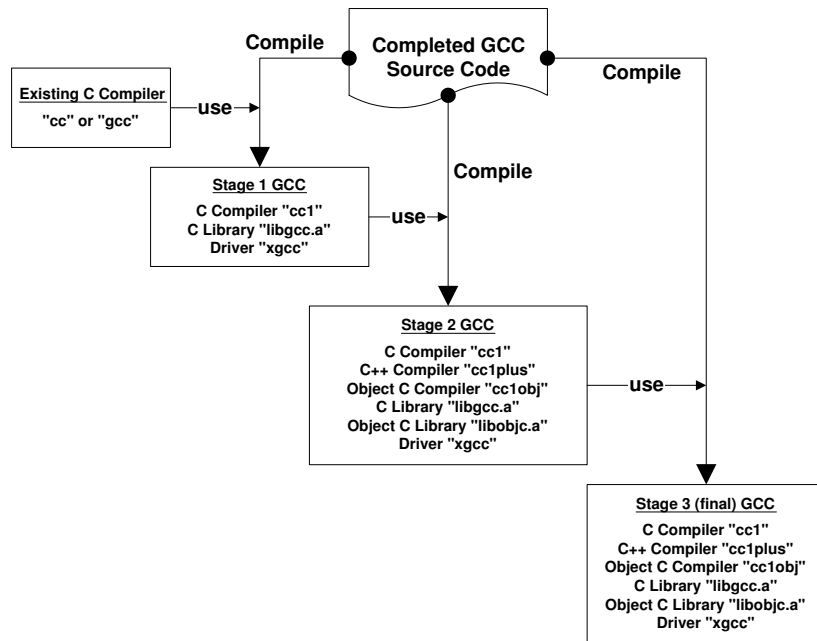


Figure 4.8: GCC Bootstrapping Build

The build-time behaviour of GCC during bootstrapping is shown in Fig 4.8. During the bootstrapping process, three different GCC compilers are built. The first one is built by the default system C compiler, and the remaining two are built by GCC itself. In all three

builds, the same source files are compiled. Three copies of the GCC compiler executables GCC are created at different time and each but the last is immediately used to compile for the next phase.

Build-time Code Generation

In GCC, the Register Transfer Language (RTL) is an intermediate representation used to represent the target system's code after parsing, similar to Java byte code. However, unlike Java byte code, RTL is hardware dependent. The specification of RTL and the portion of source code that operates on RTL are generated at build time, using machine description information and collected system parameters from the GNU `configure`. The main benefit of having a target-dependent RTL representation is that we can immediately generate the target machine language (assuming an infinite number of registers), but in a way that the compiler can understand and manipulate. Hardware-dependent optimizations also operate on this intermediate format, and only valid instructions for the particular machine are generated as the result of all passes of transformation, for RTL has built-in knowledge of target CPU architecture.

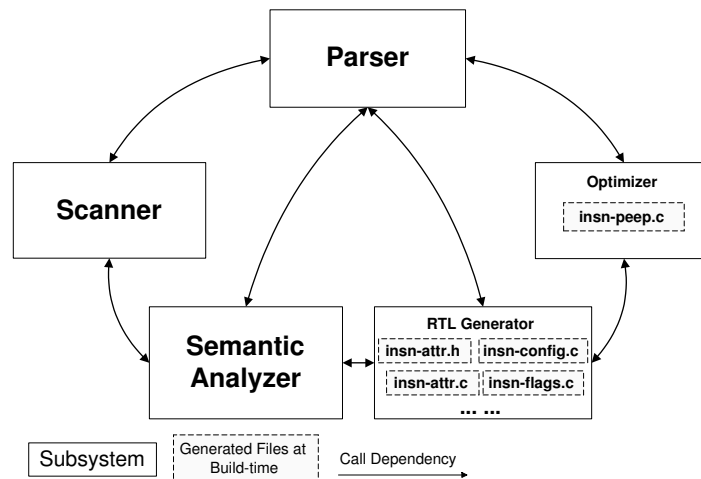


Figure 4.9: GCC Build-Time Code Generation - Generated Code

Figure 4.9 shows a portion of the code architecture view of GCC 2.7.2.3 with “holes” (dashed boxes) that represent the missing source code files. Both the core compiler subsystem and code generator subsystem contain the RTL manipulation code that is missing from the distribution. The internal code architecture of the core compiler subsystem is illustrated in Fig. 4.9.

The missing files in the core compiler subsystem are generated at build-time from code templates by source code generators.³ The procedure is explained here and illustrated in Fig. 4.10 with a build view architecture diagram.

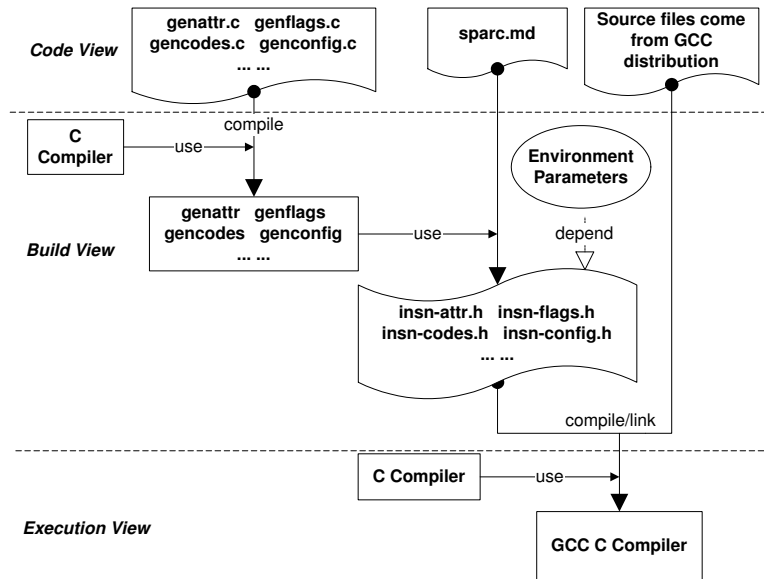


Figure 4.10: GCC Build-Time Code Generation - Code Generation Procedure

1. First, the (build-time) source code generators are compiled; the source code for these generators are contained within files whose names begin with `gen`. The result is a set of executable programs.

³The build-time source code generators shipped with the GCC source should not be confused with the object code generator subsystem of GCC, which is a standard component of any compiler.

2. Next, these code generators are executed in sequence. They take machine description files for the target machine as input. The machine description files are picked by `configure`. The output is a collection of C source files. These generated files have names that begin with `insn`. These C files are used to fill the “holes” in the code view of core compiler subsystem.
3. Finally, the source files to build a working GCC are all available. We now compile the code from the source distribution together with build-time generated code, and link them together to create the GCC compiler system.

Thus, the build-time architecture shows how the GCC system “fills in the gaps” of the code view of the shipped source code that were (intentionally) left by the GCC developers.

4.3.3 Dominance Tree Analysis of GCC Evolution

Burd and Munro use dominance trees [10] as discussed in Chapter 2 to track the change in maintainability from GCC v2.5.4 to GCC 2.8.0. They use the percentage cumulative change of strong to the direct dominance relations against direct to the strong dominance relations as the code maintainability index. They discovered that the dominance relation index dropped to negative from v2.5.5 to v2.7.0, and continuing through to version 2.7.2. Then, the index jumped dramatically to positive at version 2.8.0. They hypothesized that GCC 2.7.0 added many new features, thus decreasing the maintainability as developers were busier writing new code than finding bugs in existing code. On the other hand, GCC 2.8.0 was planned as a maintenance release, which means few features are added, as most activity involved fixing existing bugs. They later on confirmed these hypothesis from the interviews with the main developers of GCC.

4.4 Elaboration of Research Questions On GCC Evolution

After we reviewed the GCC project history, and its software architecture, we have some questions about how the software architecture of GCC has evolved during its long history

of 15 years. New programmers who want to contribute to the development of GCC might also face these questions. In this section, we elaborate on these questions, and demonstrate how BEAGLE can be used to explore the answers.

- The EGCS project played an important role in the life of GCC project. In EGCS, a brand new architecture was designed. One of the immediately perceivable results of such changes is that EGCS has a totally different source directory structure and naming scheme for source files. Traditional architectural comparison methods that detect new or deleted program entities or relations across releases will fail because they treat every entity and relation in EGCS releases as new. Since there is no common program structure between classic GCC and EGCS, they lose track of what has not been changed from GCC to EGCS. So our first research question is to find out how different architecturally EGCS really is from GCC, that is, how much of what appears to be new is actually just a reformulation or renaming of preexisting program elements.
- *Rearchitecting* activity involves changes to the system structure at the subsystem level and program level. On the other hand, *refactoring* is about restructuring program source code at the file level and function level. Given the experimental nature of EGCS project, we are interested in knowing how much the EGCS software architecture has changed during its project development period. We are also interested in comparing this result with that of stable GCC releases, such as GCC 1.x releases or GCC 2.x releases. Since EGCS is an experimental project, one would expect that it would have a different change characteristics from those of production releases, such as the extend of change for each new release, which subsystems receive most of the changes and what types of changes.
- During the long history of GCC development, there had been many efforts to rearchitect the system, and at lower level, to refactor modules. So our next question is to discover those undocumented rearchitecting or refactoring actives. Refactoring is one common practice of “perfective maintenance” at the low level, where modules are restructured for easier maintenance and comprehension. At the higher level, rearchitecting efforts reorganize the software structure at the subsystem level to add

important new features, or to satisfy other design concerns. In many Open Source Software projects, these activities are often neglected in the release document, which places more emphasis on new features and bug fix.

- We discussed previously that different build configurations will affect the software architecture that is created by the build processes. Since GCC supports many program languages, we would like to know how much of these compilers share command code modules, and how the GCC front-end system is organized to support each language. To be specific, we would like to compare the architecture of GCC with only the C compiler built-in, with those GCC architectures that include support for all of the GCC compilers (C, C++, Objective C, Java, Chill, and Fortran as in GCC 2.95.2).
- We are also interested in the distribution of the development effort among different subsystems for each releases, and for a particular subsystem, the distribution of maintenance effort across releases. This information assists us to understand the development planning of successful software project in the past, so that we could apply what we learned in planning and budgeting future releases.

In the following sections, we will adopt a tutorial style of presentation to show how BEAGLE can be used to answer the above questions. At the same time, we will present some discoveries about the evolution of GCC.

4.4.1 From GCC 1.0 To GCC 2.0

As we discussed in the sections on background and history of GCC, it was a significant improvement of GCC to evolve from version 1.x to 2.x, as a C++ compiler was integrated with the system, a new back-end that can target more hardware platforms, and many other Improvements were made. In this section, we will demonstrate how to use the basic architecture comparison facility provide by BEAGLE to find changes made to the GCC architecture when it evolved to a new major release.

Select Comparison Options

First, we click on the **Architecture Evolutions** link in the menu frame on the left of the screen to enter the **Comparison Option** window, where we would select which GCC releases to investigate, what is build configuration of these releases, and which release should be used as the reference (base point of all the comparisons).

In this example, we select GCC release 1.39 as the representative release for GCC 1.x, and GCC release 2.0 as the first GCC 2.x releases. To fully understand the architecture differences between these two releases, we need to perform two comparisons in BEAGLE. In the first comparison, we set the newer version 2.0 as the reference, so that we could observe all the new entities and relations that are added new to GCC 2.x architecture. In the second comparison, we set the earlier version 1.39 as the reference, so that we can observe all the entities and relations that are to be discarded from the older architecture.

Figure 4.11 shows the screen where we perform the first comparison. We choose GCC 2.0 as the reference release, and the “ALL” configuration option that includes all the supported GCC compilers. For the second comparison, we only need to change the selection of **show the software architecture of** from **newest release** to **oldest release**, and we will have GCC 1.39 as the reference release.

Compare Architecture: Overall System

After clicking on the **Submit** button, we examine the comparison result screen. It shows the architecture differences between the two selected releases at the subsystem level. The diagram on the top of figure 4.12 shows results for the comparison with GCC 2.0 as the reference. Red entities and relation arrows are unique to GCC 2.0, which means they were added to the newer architecture. Green entities means the components themselves exist in both releases, and they contain newly added entities inside. Cyan color entities are those that remain unchanged in GCC 2.0.

The diagram at the bottom of figure 4.12 shows the query result when we selected GCC 1.39 as the reference release. The blue entities and relation arrows are unique to GCC 1.39, which means they will be deleted from the newer GCC 2.x architecture. Green entities exist in both releases, and they contain subcomponents that are also deleted later in GCC 2.0 too. Cyan colored entities are those that remain unchanged.

Please choose two or more GCC/EGCS historical releases to compare.

☒ Compare **Two Releases**: First Release Second Release

☐ Compare **Multiple Releases**:

Choose A Build Configuration:

Show the Software Architecture of ☒ **Newest Release** ☐ **Oldest Release**

Figure 4.11: Architecture of GCC 2.0 Comparing to GCC 1.39 - Selection Screen

By comparing the two diagrams, we get the initial impression that GCC 2.0 added many more new program entities and relations comparing to GCC 1.39 than the entities and relations that were removed. This finding agrees with the general belief that software always grows larger in size. GCC 1.39 contains 70 source files, while GCC 2.0 has 126 source files. The system almost doubled in size.

Comparing Subsystem Architectures: Parser

Now we want to zoom into the compiler subsystem to see how its architecture has changed. Click on the `Compiler.ss` icon to expand its branch in the system structure tree, all the second level subsystems contained in compiler subsystem will be shown, along with their change status. The architecture landscape frame also updates to display the current subsystem. Because the compiler subsystem does not directly contain any source files, we continue to zoom into the Parser subsystem underneath to see how it has changed.

The screenshot on the top of figure 4.13 shows the architecture landscape of GCC 2.0 comparing to the reference release v1.39. It highlights the components and relations that are new to the architecture, and those containing new sub-components.

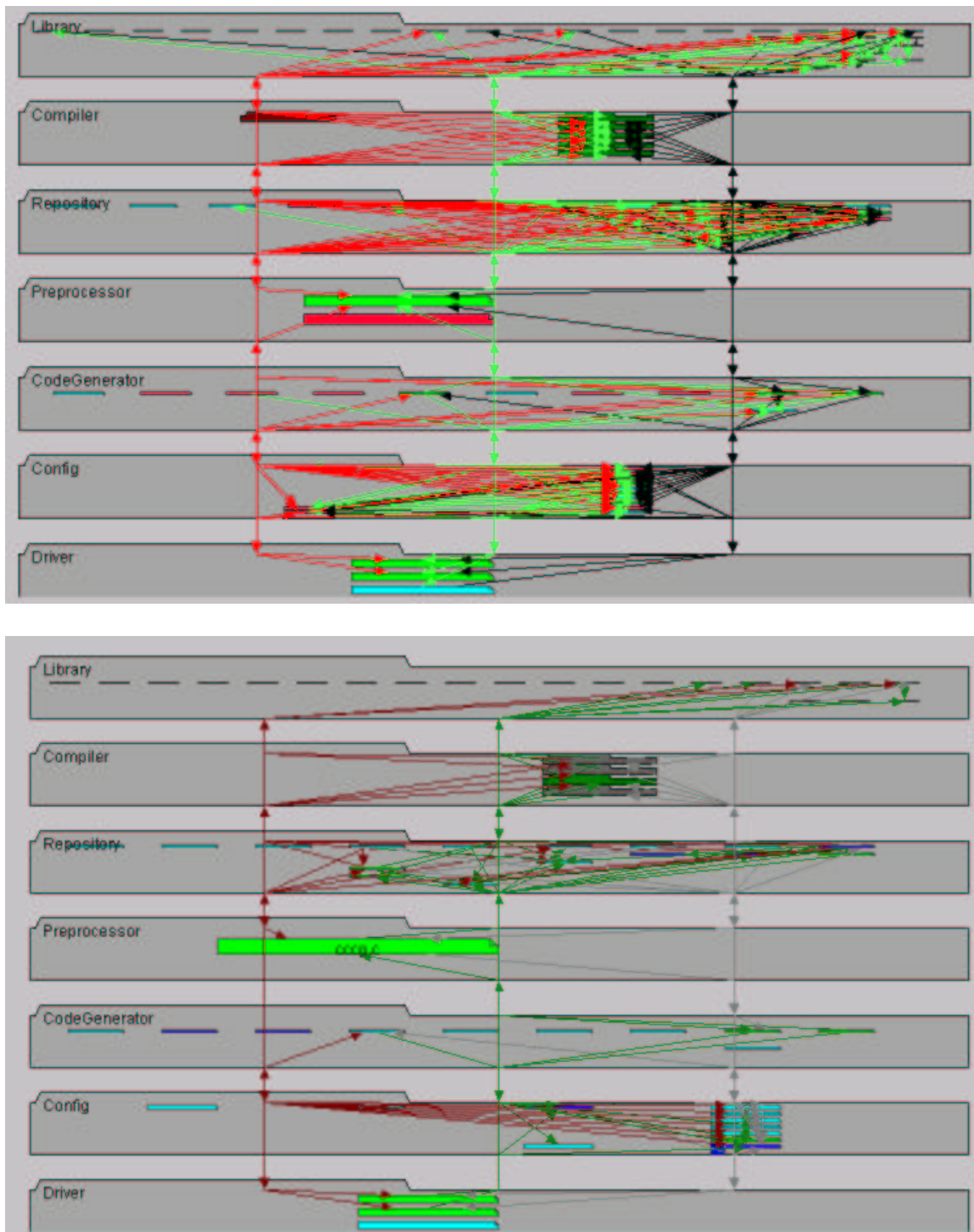


Figure 4.12: Architecture Comparison of GCC 2.0 and 1.39 - Top Subsystems

From the summary displayed in the information frame, we can see out of 21 files contained in Parser subsystem, 17 of them are new. From the structure tree frame on the left, we can observe that most “red” files have “c”, “cp”, or “objc” as suffix. Recall the history of GCC, we know that GCC 2.0 is the first release that integrated three language compilers — C, C++ and Objective C — into one GCC distribution. As a consequence, GCC developers designed a new Parser subsystem, where modules that handle different languages are differentiated by the suffix in their file names.

The screenshot at the bottom of figure 4.13 shows the architecture of GCC 1.39 compared to reference release v2.0. It highlights the components and relations that were deleted from the new GCC 2.x architecture, and those contain sub-components that became obsolete. Only one file, `c-parse.tab.c` will be deleted from parser subsystem. Two other files, `c-decl.c` and `fold-const.c` also have functions that no longer exist in the newer release.

Comparing Subsystem Architectures: RTL Generator

The RTLGenerator subsystem belongs to the Compiler subsystem. In theory, it should be the last stage of compiler front-end, as the output of this stage of compilation should be the intermediate representation in RTL format. However, in practice, this subsystem also contains a small portion of the compiler back-end, because the RTL format used by GCC is partially CPU architecture dependent. As mentioned before, all three parsers in GCC (C, C++, and Objective C) generate their intermediate code in RTL format, so we expect very little change will be made to the old GCC 1.x code that generates and manipulates RTL code from the parse tree. On the other hand, because GCC 2.x is designed to provide better support for RISC CPUs, we also expect some new code to be added to this subsystem so that RTL code can be generated and then optimized at early stage by considering the special characteristics of RISC architecture.

The screenshot at the top of figure 4.14 shows what has been added to GCC 2.0. As we expected, few completely new files were added. Some of them are specific to C++ compiler, such as `cp-expr.c` and `cp-init.c`. The majority of the additions occurred at the function level, as many new functions are introduced. We believe these new functions extends the way RTL code is generated, so that GCC 2.x can generate code for a much

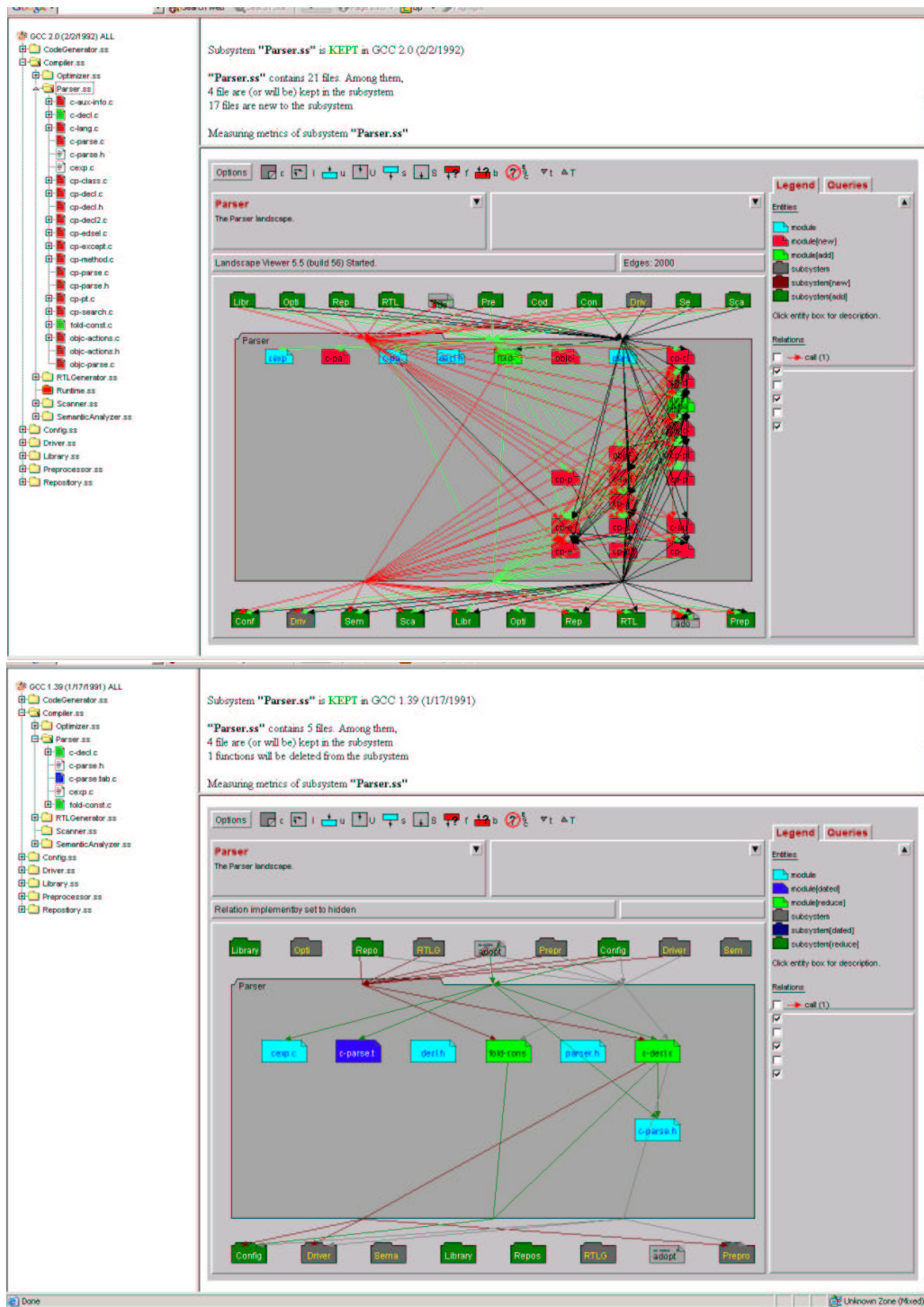


Figure 4.13: Comparison of GCC 2.0 and 1.39 - Parser

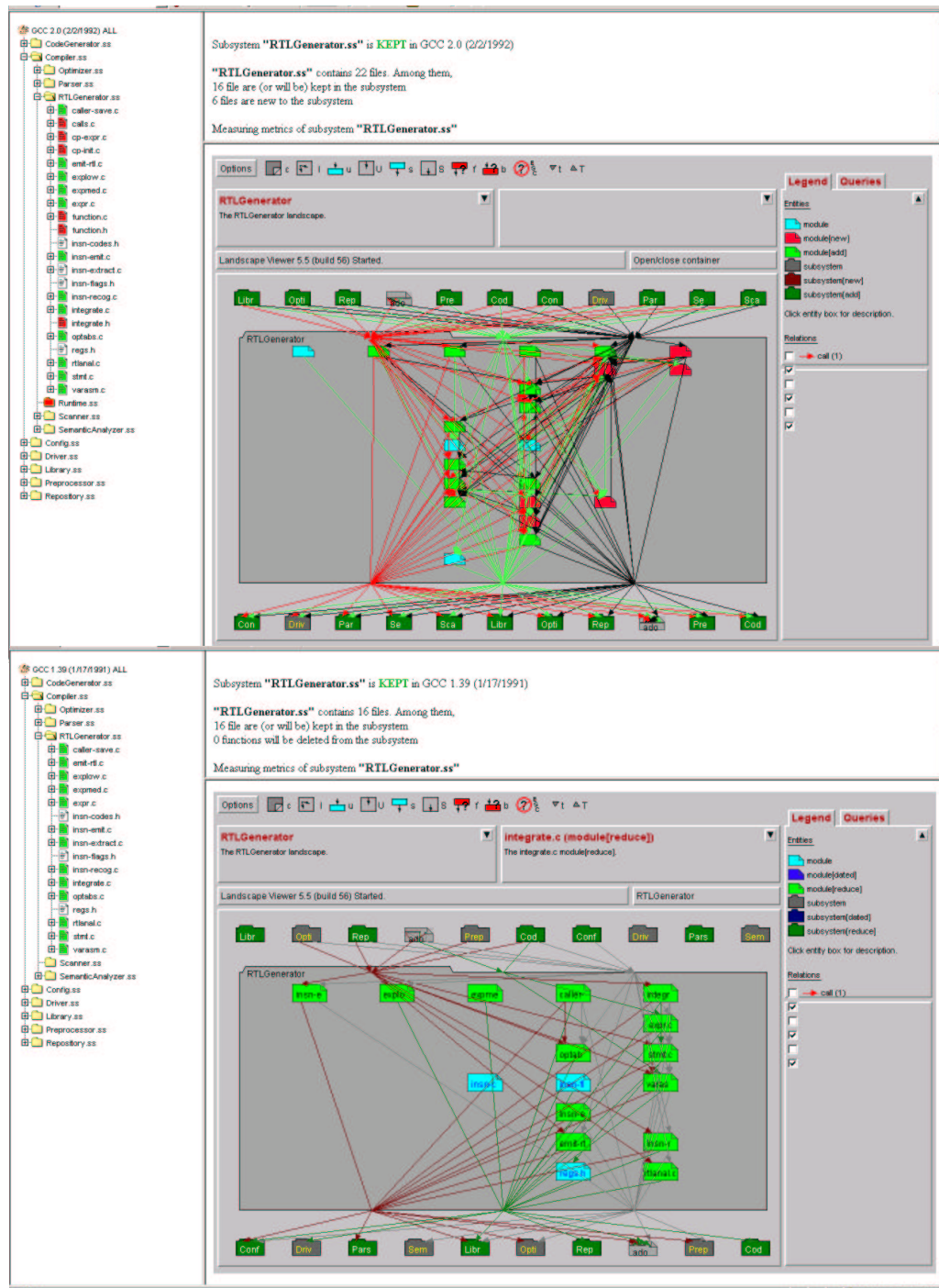


Figure 4.14: Comparison of GCC 2.0 and 1.39 - RTL Generator

broader range of CPU platform.

The screenshot at the bottom of figure 4.14 shows all source files within the RTLGenerator subsystem from older GCC 1.x are still kept in GCC 2.0. However, many files have obsolete functions deleted from the older architecture.

Comparing Subsystem Architecture: Code Generator

The Code Generator subsystem is one of the most important back-end subsystems. Assembler code that is specific to target CPU is emitted at this stage. Since GCC 2.0 supports many target CPUs, we expect this subsystem in GCC 2.0 architecture will have many new files. Figure 4.15 verified our expectation, as the number of files in this subsystem almost doubled in GCC 2.0, and only two files `gnulib.c` and `gnulib2.c` are moved out of the system. Our investigation suggested that the removal of these two files is related to the new way in which GCC 2.0 handles command libraries. It does not mean the removal of major features from the subsystem.

4.4.2 From GCC 2.x To EGCS 1.x

There are many differences between GCC releases and EGCS releases. For example, EGCS releases reorganized their source directory structure, and also adopted a new naming convention for the source files. These changes make conventional architecture comparison methods, which identify changed and unchanged program entities by comparing their names and directory location in both releases, no longer applicable. These methods first choose one release as reference, and then all entities in the other release that have different name and location in the source directory are treated as changes compare to the reference. If the two releases under comparison have very different source structure, conventional methods will treat everything in the later release as new entities.

Figure 4.16 shows the comparison results by selecting **Architecture Evolution** from the main menu, which is a “name and location based” comparison method as used in our previous comparison of GCC 2.x and GCC 1.x. As we can see, every program entity (file and function) in the system structure tree and landscape diagram is red, which means they are falsely identified as “new” in EGCS 1.0.

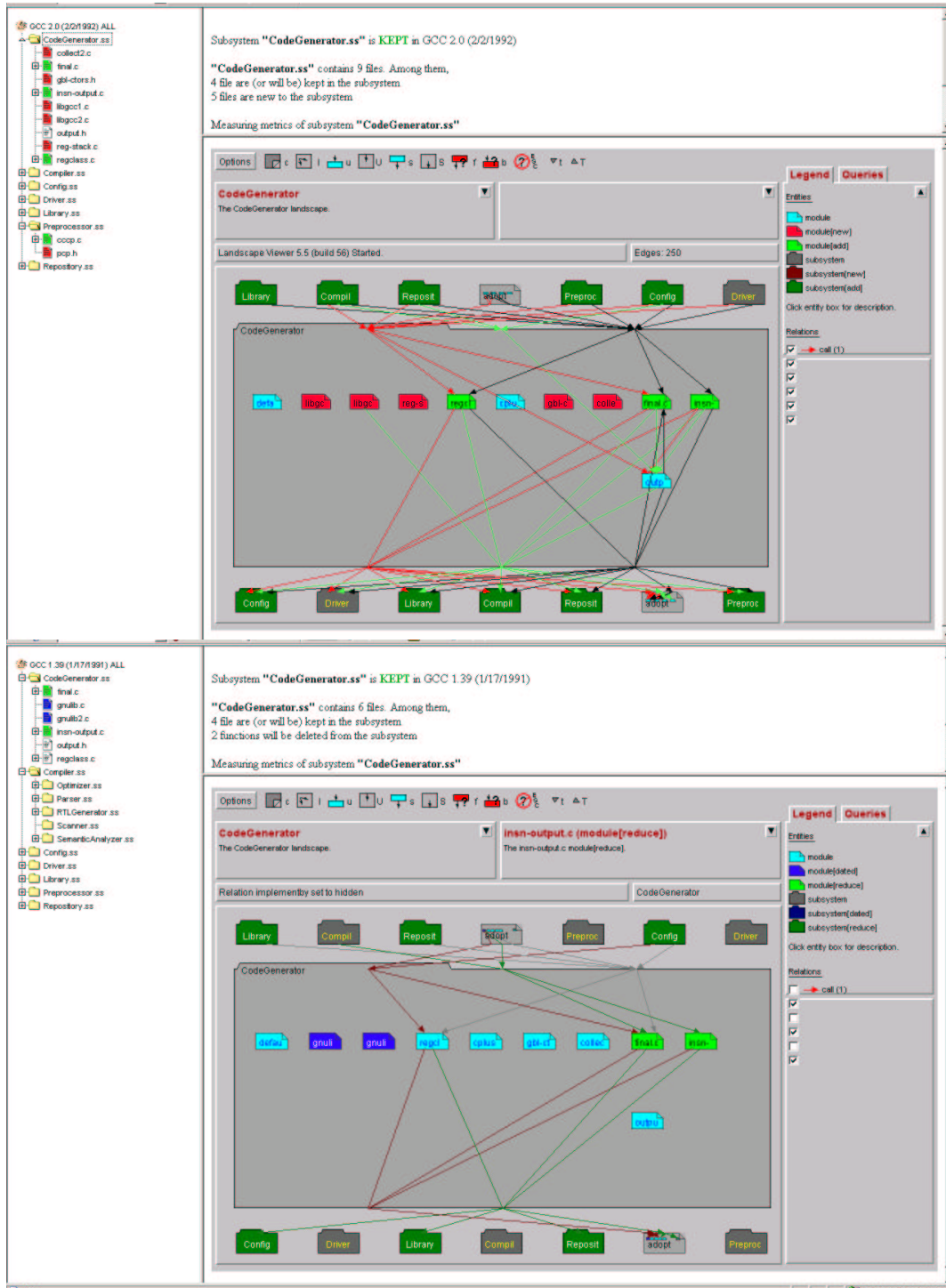


Figure 4.15: Comparison of GCC 2.0 and 1.39 - Code Generator



On the other hand, with *Origin Analysis* we can analyze whether a particular function has a correspondent from the previous release, or it is newly written for the later release. If there a corresponding function is found in the previous release that either has similar features or similar dependencies with this function, we can related these two functions, as it demonstrates how this particular function has been moved inside the program structure as the software architecture evolves into the new release.

We can also perform *Origin Analysis* at the file level. This will examine every function defined in a given file, then count how many functions already exist in the previous release, as GCC 2.7.2.3 in this example, and how many functions are new in EGCS. If majority of the functions came from a single file in GCC 2.7.2.3, we can conclude that this new file in EGCS 1.0 are inherited from that file in GCC 2.7.2.3.

Figure 4.17 shows the result of a sample Origin Analysis request on file `gcc/c-decl.c`. Among 70 files defined in this file, 41 functions can be traced back to their origin functions defined in the previous GCC release by using *Bertillonage Analysis*. With no exceptions, all the original functions were defined in file `c-decl.c` of GCC 2.7.2.3.

Starting from EGCS 1.0, many source files that directly contribute to the building of the C/C++ compiler were moved to a new subdirectory called `/gcc`. To analyze the architecture change at this magnitude (from GCC to EGCS), *Bertillonage Analysis* has been demonstrated to be more effective than *Dependency Analysis*. *Dependency Analysis* assumes that when we analyze a “new” function, its callers and callees from both current release and the previous release should be relatively stable, which means most of the functions that have dependencies on this particular function should not also renamed or related in the newer release. However, this is not the case when completely different software architecture is adopted in EGCS 1.0 comparing that of GCC 2.7.2.3, and most of the files and functions are either renamed or related in the directory structure.

In our case study, we have performed *Origin Analysis* on every source file of EGCS 1.0, which attempts to located possible “origins” in its immediate previous release of GCC 2.7.2.3. The goal is to understand what portions of the old GCC architecture is carried over to EGCS, and what portion of EGCS architecture represents the new design. This test takes 3 days to run on a Dual Pentium III 1GHz workstation. Here we present the result for two representing subsystems of GCC: Parser and Code Generator. One is from

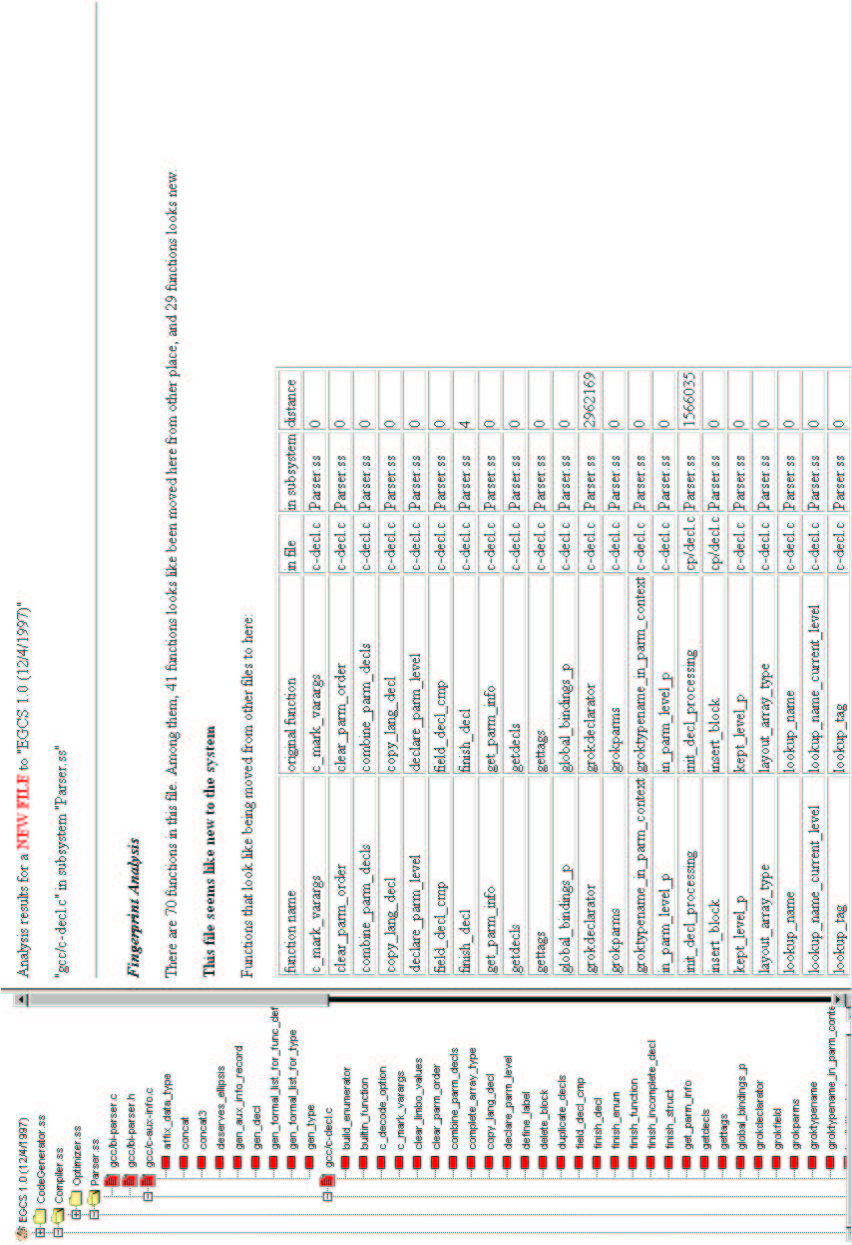


Figure 4.17: Origin Analysis on EGCS 1.0

compiler front-end, and another from the back-end. Both of them are essential to the EGCS software architecture, so their evolution story is representation of entire EGCS system.

There are 30 files in the **parser** subsystem. Half of them are header files, or very short C files that defined macros. We will not consider these files in the analysis. Of the remaining 15 files, we have three files considered to be old GCC file carried over from v2.7.2.3. We say files are “old” if more than 2/3 of functions defined in the file have “origin” in the previous release, on the other hand, “new” files should have less than 1/3 of carried over functions. In the parser subsystem, we have seven of such “new” files. All the other files are considered as “half-new, half-old”, which numbered five. Overall, out of 848 functions defined in the parser subsystem of EGCS 1.0, 460 are considered “new”, and 388 are considered “old”. The “new” functions counted as 56 percent of total functions. For a new release of compiler software, this percentage of newly designed code is really high, esp. for a subsystem that is based on mature techniques such programming language parser. Table 4.1 lists the complete result.

File Name	Total Func	New	Old	Change Type	Origin
gcc/c-aux-info.c	9	0	9	Mostly Old	c-aux-info.c
gcc/fold-const.c	44	15	29	Mostly Old	fold-const.c
gcc/objc/objc-act.c	167	17	150	Mostly Old	objc-act.c
gcc/c-lang.c	16	14	2	Mostly New	c-lang.c
gcc/cp/decl2.c	57	50	7	Mostly New	cp/decl2.c
gcc/cp/errfn.c	9	9	0	Mostly New	None
gcc/cp/except.c	25	20	5	Mostly New	cp/except.c
gcc/cp/method.c	30	26	4	Mostly New	cp/method.c
gcc/cp/pt.c	59	57	2	Mostly New	cp/pt.c
gcc/except.c	55	52	3	Mostly New	cp/except.c
gcc/c-decl.c	70	29	41	Half-Half	c-decl.c
gcc/cp/class.c	61	31	30	Half-Half	cp/class.c
gcc/cp/decl.c	134	84	50	Half-Half	cp/decl.c
gcc/cp/error.c	31	16	15	Half-Half	cp/error.c
gcc/cp/search.c	81	40	41	Half-Half	cp/search.c

Table 4.1: Origin analysis results on EGCS 1.0 — parser

Table 4.2 lists the result of the Origin Analysis on the Code Generator subsystem.

Out of the five files that contain function definitions, three files are considered “new” by the analysis, and the rest two are considered “old”. Overall, 84 percent of the functions defined in Code Generator subsystem are newly written. This percentage is much higher than the Parser subsystem. From these results, we can conclude that EGCS 1.0 has a significant portion of the source code that were newly developed comparing to the GCC release that it is suppose to replace, especially for back-end subsystems.

File Name	Total Func	New	Old	Change Type	Origin
<code>gcc/cplus-dem.c</code>	36	36	0	Mostly New	None
<code>gcc/crtstuff.c</code>	5	5	0	Mostly New	None
<code>gcc/insn-output.c</code>	107	95	12	Mostly New	<code>input-output.c</code>
<code>gcc/final.c</code>	33	20	13	Half-Half	<code>final.c</code>
<code>gcc/regclass.c</code>	20	12	8	Half-Half	<code>regclass.c</code>

Table 4.2: Origin analysis results on EGCS 1.0 — code generator

4.4.3 Stable Releases vs. Development Releases

GCC Evolution During EGCS 1.x Releases

To compare the different evolution patterns of GCC stable releases and EGCS experimental releases, we first issue a query to BEAGLE as shows in Figure 4.18.

From the main menu, we choose the selection of **Architecture Evolution**, and then from the query frame on the right, we select the radio button **Compare Multiple Releases**. Since we are going to view the evolution history of the EGCS project, we select all of the EGCS releases from EGCS 1.0 down to EGCS 1.1.2, which contains seven releases in total. We want to see how entities and relations have been added to the system, so we select the option **Show the landscape of newest release**. Then click on the **Submit** button to execute the query.

Figure 4.19 and Figure 4.20 show the query results. Figure 4.19 presents the EGCS evolution history for Optimizer subsystem and Code Generator subsystem, both of which are essential components for GCC compiler back-end. Figure 4.20 shows the evolution

Please choose two or more GCC/EGCS historical releases to compare.

☐ Compare **Two Releases**: First Release Second Release

☒ Compare **Multiple Releases**:

Choose A Build Configuration:

Show the Software Architecture of ☒ **Newest Release** ☐ Oldest Release

Figure 4.18: Evolution of EGCS Releases - Make Query

patterns for Parser subsystem, which is one of the most important subsystem of compiler front-end.

In figure 4.19, we can observe that in the Code Generator subsystem, there was only one new file `gcc/frame.h` (in red icon) added at EGCS 1.0.1 during the entire EGCS project, and there are four files (green icon) having new functions defined within it. In the Optimizer subsystem, there are three new files added at various releases, as `gcc/gcse.h` added at EGCS 1.1, `gcc/global.c` added at EGCS 1.0.1, and `gcc/haifa-sched.c` in EGCS 1.1. Many other files contain new functions inside of them, as we can see eight files are shown in green icon in the subsystem.

From the files branched out from Optimizer subsystem, we can observe that there were many new files added during the life of EGCS. This finding is expected, as EGCS project emphasized trying out many state-of-art optimization and machine instruction scheduling algorithms, thus this will certainly introduce new files and new functions to this subsystem.

Figure 4.20 shows the detailed evolution information for the Parser subsystem. Comparing to the evolution history of Optimizer and Code Generator subsystems, there are no new files added to the Parser subsystem during the life of EGCS project. However, many of the files had new functions defined within it.



The reason for this finding is obvious if we understand the history background of GCC and EGCS. When EGCS project started, the language front-end of GCC compiler had been very mature. The extra features needed at language front-end included support for new ANSI C++ features, which can be achieved easily by defining new functions inside existing source files. This explains why in the Parser subsystem, the number of new functions dominates the number of new files. However, this is not the case for compiler back-end, such as Code Generator subsystem, and Optimizer subsystem. New optimizing techniques and new scheduler algorithms were used extensively in EGCS releases. This means extra optimization cycles are needed at the back-end. With pipe-and-filer architecture style, one processing step always corresponds to one code module [46]. As the result, new source code modules (files) are added at the back-end subsystems to support the extra optimizing cycles.

BEAGLE also allows us to investigate the change history of individual files or functions. In the information frame, if we scroll the page down, we can see the comparison metrics for selection file or function. If we click on the link **view file history** or **view function history**, BEAGLE will show a table listing the entire history of selected file or function in term of its major metrics. By observing the change pattern of its metrics, we will have some idea about how this file or function has been evolved during its entire life.

Table 4.3 shows the evolution history of file `gcc/combine.c` from Optimizer subsystem during EGCS releases. From the table, we can observe that the size of this file increased three times during EGCS project at 1.0.1, 1.1, and 1.1.2. The reason for second size increase at EGCS 1.1 is caused by new functions defined within this file. The functions within the file had not changed much structurally, as the average cyclomatic complexity and average fan-out metric for all defined function remained constant.

Table 4.4 shows the evolution history of a selected function "add_method" in EGCS releases. This function is defined in file `class.c` and part of the C++ compiler. Its job is to add a `method` to the `type` that defined by the current class. We can observe that this function kept the same during EGCS 1.0.x releases, then there is a major change at EGCS 1.1, and later remain unchanged throughout EGCS 1.1.x releases. The change itself is very interesting, as the length of the function decrease by 16 lines, as well as all other complexity metrics decrease in value. After we have investigated the source code of this

EGCS Release	Code	Comment	Func	Avg. Cycl.	Avg. Fan-out	MI
1.0	7279	2740	58	28	23	-30
1.0.1	7281	2742	58	28	23	-30
1.0.2	7281	2742	58	28	23	-30
1.0.3	7281	2742	58	28	23	-30
1.1	7532	2799	59	28	23	-30
1.1.1	7532	2799	59	28	23	-30
1.1.2	7569	2814	59	28	23	-31

Table 4.3: Change history of file `gcc/combine.c`

function by following the links provided in the web page, we found out that in EGCS 1.0.x, one of the function parameter `method` (defined as a pointer to data type `tree`) is not used directly in the function body. At the beginning of the function, more than 10 lines of code is consumed to make a copy of data structure pointed by `method` to `decl`. According to the comment, the purpose of this step is to “be sure that we have exclusive title to this method’s `DECL_CHAIN`”. After this point, all the reference to `method` is replaced by `decl`. In EGCS 1.1.x, this extra copy procedure is eliminated, and all the references to `decl` are changed back directly to the function parameter `method`. We suspected it was caused by changes made to other parts of the system, especially to the `tree` data structure. This explains the shrinkage in the code size of this function in EGCS 1.1x comparing to EGCS 1.0.x. Also because references to local variable and function parameter generate different result in the calculation of many composite complexity metrics, we have different S-Complex, Albrecht and Kafura metric values for these two function definitions.

GCC Evolution During GCC 2.x Releases

Now that we have examined the evolution of experimental EGCS releases, let us compare it with those of stable production release of GCC from 2.0 to 2.7.2. Our goal is to compare the different evolution strategy adopted by the project planner, and how they treat experimental releases and stable releases differently.

Figure 4.21 shows the evolution history of GCC from 2.0 to 2.7.2. In this particular screen, only new entities are presented. For Optimizer subsystem, we observe that only

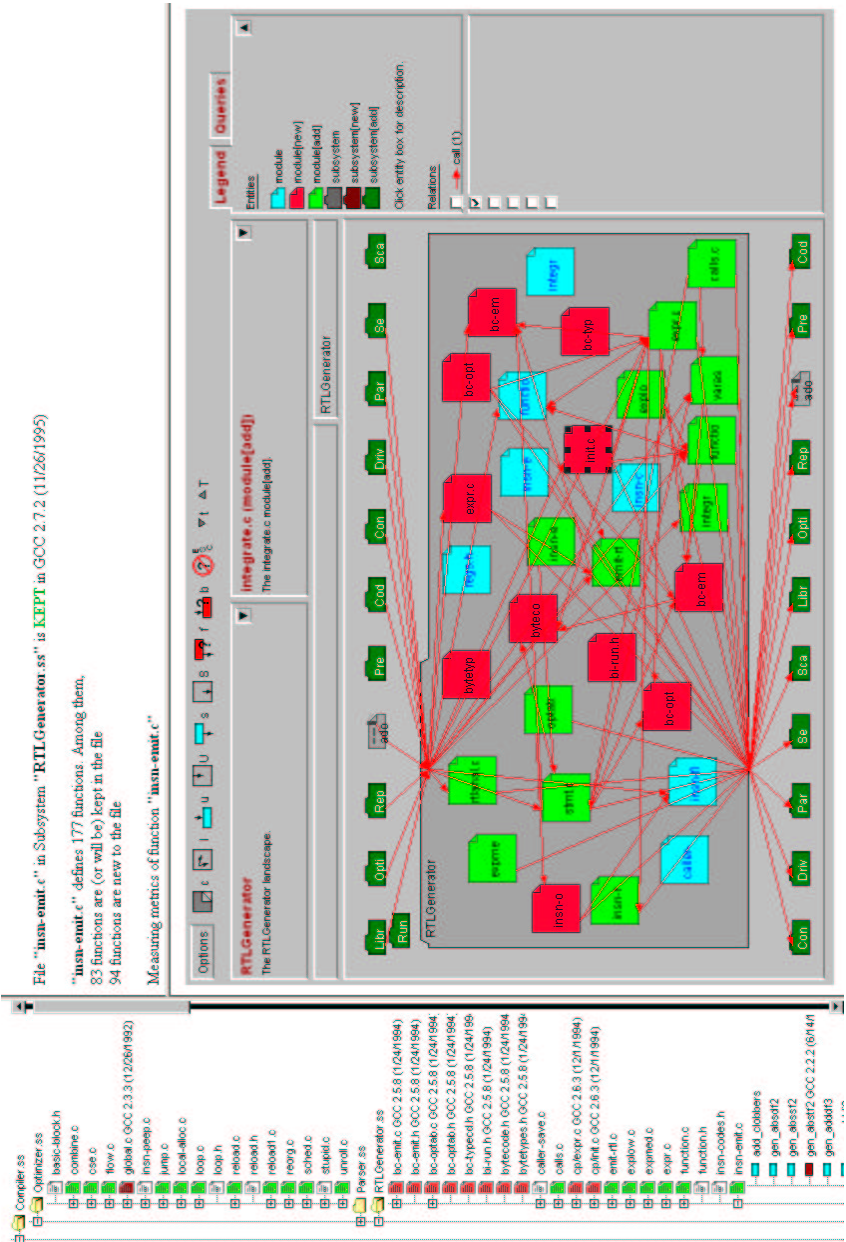


Figure 4.21: Evolution of GCC 2.x Releases - Code Generator and Optimizer

EGCS Release	Code	Comment	Cycl.	SComplex	DComplex	Albrecht	Kafura
1.0	106	16	11	144	5.4	408	24336
1.0.1	106	16	11	144	5.4	408	24336
1.0.2	106	16	11	144	5.4	408	24336
1.0.3	106	16	11	144	5.4	408	24336
1.1	90	13	10	121	5.5	381	20736
1.1.1	90	13	10	121	5.5	381	20736
1.1.2	90	13	10	121	5.5	381	20736

Table 4.4: Change history of function `add_method`

one file was introduced new during the long release period. All other addition happened beneath file level. It contrasts to the pattern of EGCS releases that we just investigated, where many new files are introduced.

For the RTL Generator subsystem, the situation is a little different. Many new files were introduced at different releases in this period. When we pay closer attention to the filenames of these new files, we discover that many of them are header files, such as `bytetype.h`, `bi-run.h`, and `bc-typecd.h` which define several macros and data structures such as `C struct` and `union`.

Figure 4.22 shows the evolution history of the Parser subsystem during GCC 2.x stable release period. We can see many new files were introduced, especially those files related to C++ compiler and Objective C compiler. We can attribute these additions to the continuous effort of GCC development to support three languages from the C family in one system, and have them work together harmoniously. Also during this long period, the language standards themselves are kept evolving, so it is natural to continually change the language front-end to keep the parser up-to-date. When the EGCS project started, all three languages were finally standardized, so there was no more need to change parser element dramatically.

By comparing the evolution pattern of GCC stable releases with EGCS experimental releases, we conclude that except some other factors, most new code is added to the stable releases in the form of new functions within existing files, while for experimental releases, they tend to be in the form of new files.

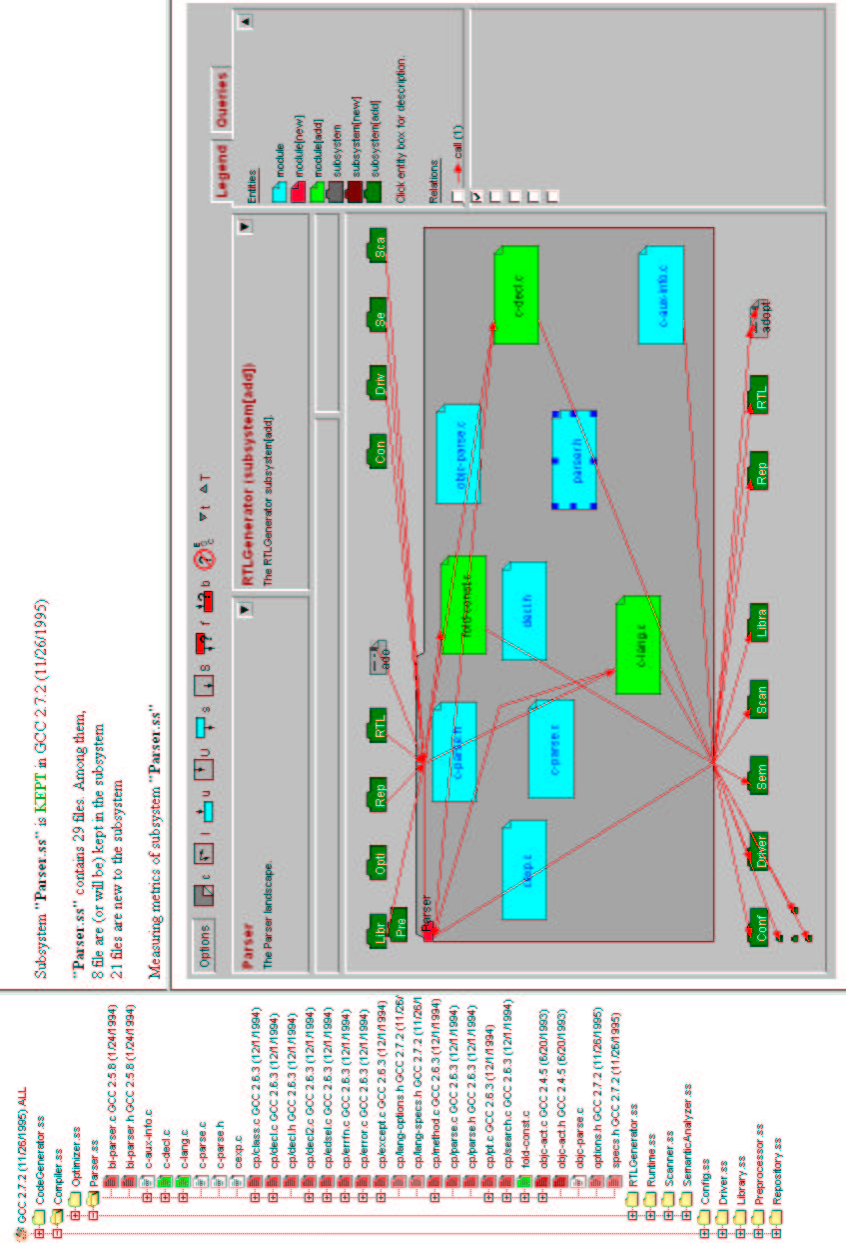


Figure 4.22: Evolution of GCC 2.x Releases - Parser

4.4.4 Build Configurations and GCC Architecture

In this section, we compare the architecture of GCC 2.7.2 under two different build configurations. The first one built only the essential C compiler, and the other configuration will build all the compilers that GCC supports.

In GCC, every language compiler generates its intermediate code in RTL format, so they share the same back-end subsystems including RTL Generator, Code Generator, and Optimizer. At the front-end, they each have their own scanner and parser subsystem. Each parser creates a syntax tree directly into the tree format, and later is processed by the semantic analyzer. This tree format is identical for all the program language supported in GCC. The partial trees are then passed from time to time to routines that belongs to the RTL Generator subsystem to convert the syntax tree to the RTL format. So even though each language compiler has its own scanner and parser, all the parsers create parse tree using the same data structure as defined in the `tree.h`, `tree.c`, and `tree.def`. Joachim Nadler and Tim Josling thoroughly discussed the procedures to write a new compiler front-end for GCC that reuses the standard parser tree data structure and back-end components of GCC in [39].

Figure 4.23 shows the difference between two builds of the Parser subsystem. All red icons represent files unique to the complete build comparing to the C only build. The purpose of these files is to support programming languages other than C. In GCC 2.7.2, they are C++ and Objective C. By examine the filenames of these extra file, we find out that they all contain special suffix. For example, `texttttcp` are common suffix for C++ compiler files, and `textttobjc` are suffix for Objective C compiler. All other files that do not have language specific suffix such as `options.h` and `fold-const.c` and core files for C compiler such as `c-decl.c` and `c-parse` are shared between the two build configurations.

For the back-end subsystems, such source sharing is much more common, as compiler back-ends are usually program language independent. In figure 4.24 that shows the Optimizer subsystem under two different build configurations, all the files are shared between the two build configurations (white color file icons). The only differences are dependency links that are initiated from modules in other subsystems.

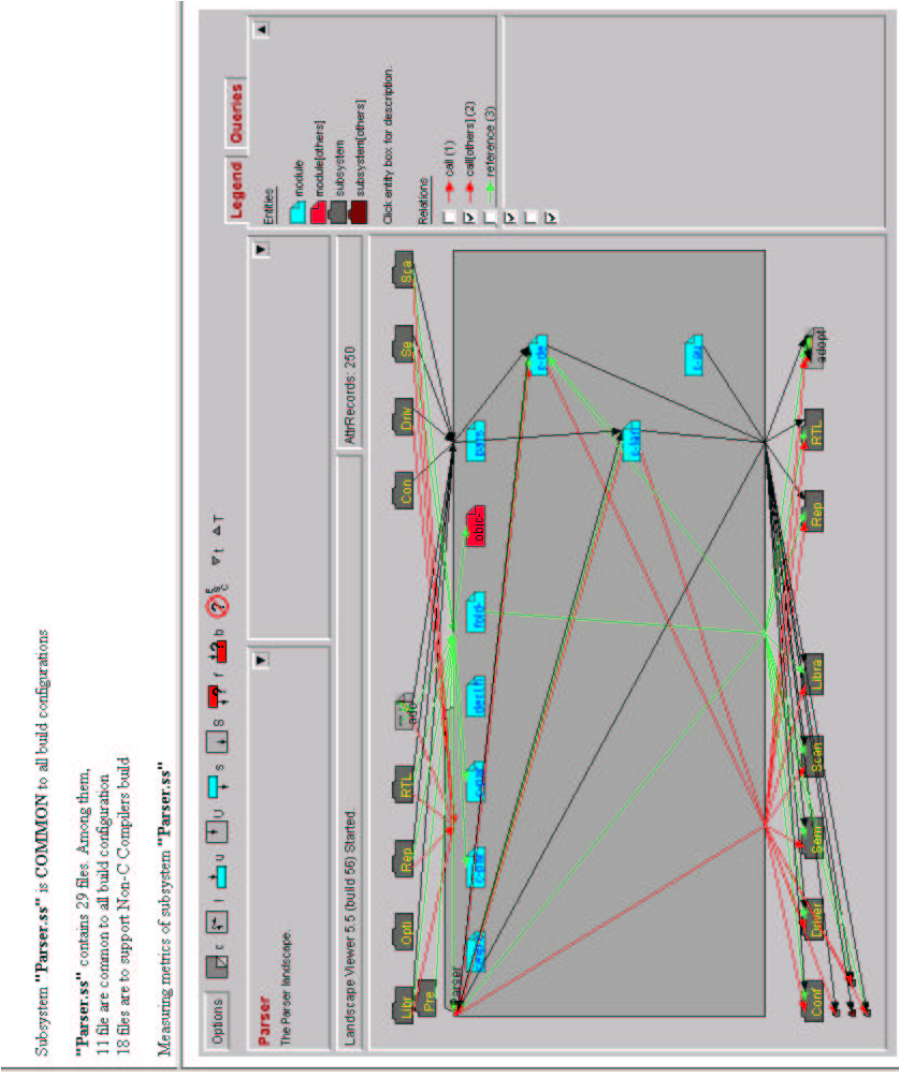


Figure 4.23: Compare C-Only and All Build Configuration - Parser

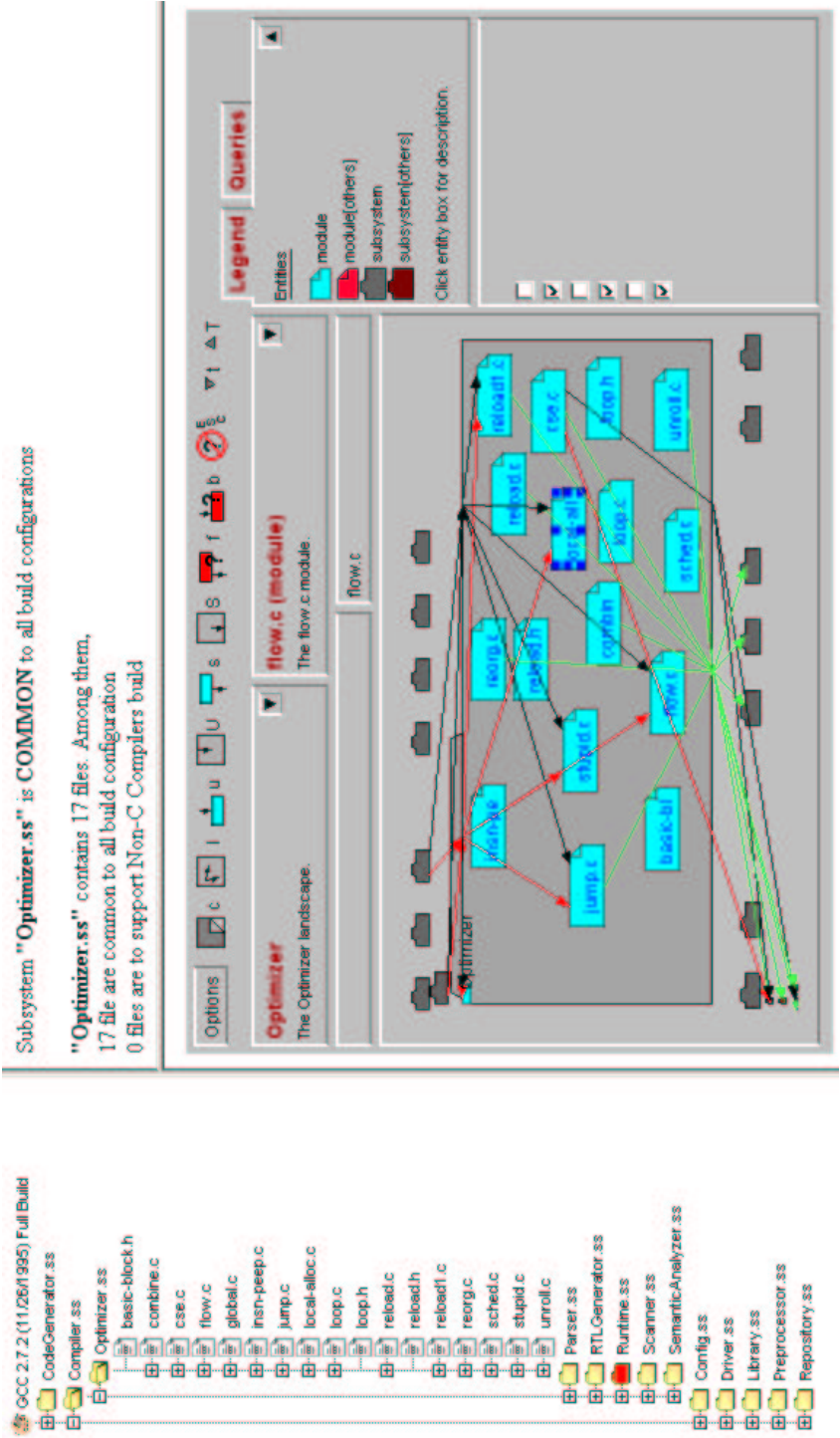


Figure 4.24: Compare C-Only and All Build Configuration - Optimizer

4.4.5 Refactoring and Rearchitecting

In this section, we will demonstrate how to use *Origin Analysis* as supported by BEAGLE to discover undocumented refactoring and rearchitecting activity in GCC 2.0.

First we click on link **Analyze Changes** in the main menu, then we choose GCC 2.0 from the pull-down list, and then click on **Submit** button. The origin analysis is performed at both the file level and the function level. If the user selects a file, BEAGLE will perform Bertillonage analysis on all functions that are defined within that file. It will also apply dependency analysis on the source file to detect its possible origin file from the previous release. If the user selects a function, both Bertillonage analysis and dependency analysis are performed on the selected function.

In Figure 4.25, we select the file `function.c` from the system tree of GCC 2.0, and the analysis results are displayed in the frame on the right. This screen tells us that there are 49 functions defined in this file; Bertillonage analysis returns seven positive matches, and no matches for the remaining 42 functions. This result reveals that there are even instances of refactoring efforts put into this file, where functions are moved from their original location to this file. In this particular example, they are all from same file `stmt.c` in GCC 1.42.

Tables 4.5 shows the result of Bertillonage analysis on function `assign_parms` that is defined in file `function.c`.

	Function	File	Subsystem
1	<code>build_binary_op_nodfault</code>	<code>c-typeck.c</code>	Semantic Analyzer
2	<code>assign_parms</code>	<code>stmt.c</code>	RTLGenerator
3	<code>recog_5</code>	<code>insn-recog.c</code>	RTLGenerator
4	<code>store_one_arg</code>	<code>expr.c</code>	RTLGenerator
5	<code>gen_muls3</code>	<code>insn-emit.c</code>	RTLGenerator

Table 4.5: Bertillonage analysis on function `assign_parms`

Bertillonage analysis correctly finds the original function. The origin function has the same name as the “new” function, and it ranks second in Bertillonage distance. For dependency analysis, we are given no result by caller analysis. However, we get good results from callee analysis, as shown in table 4.6.

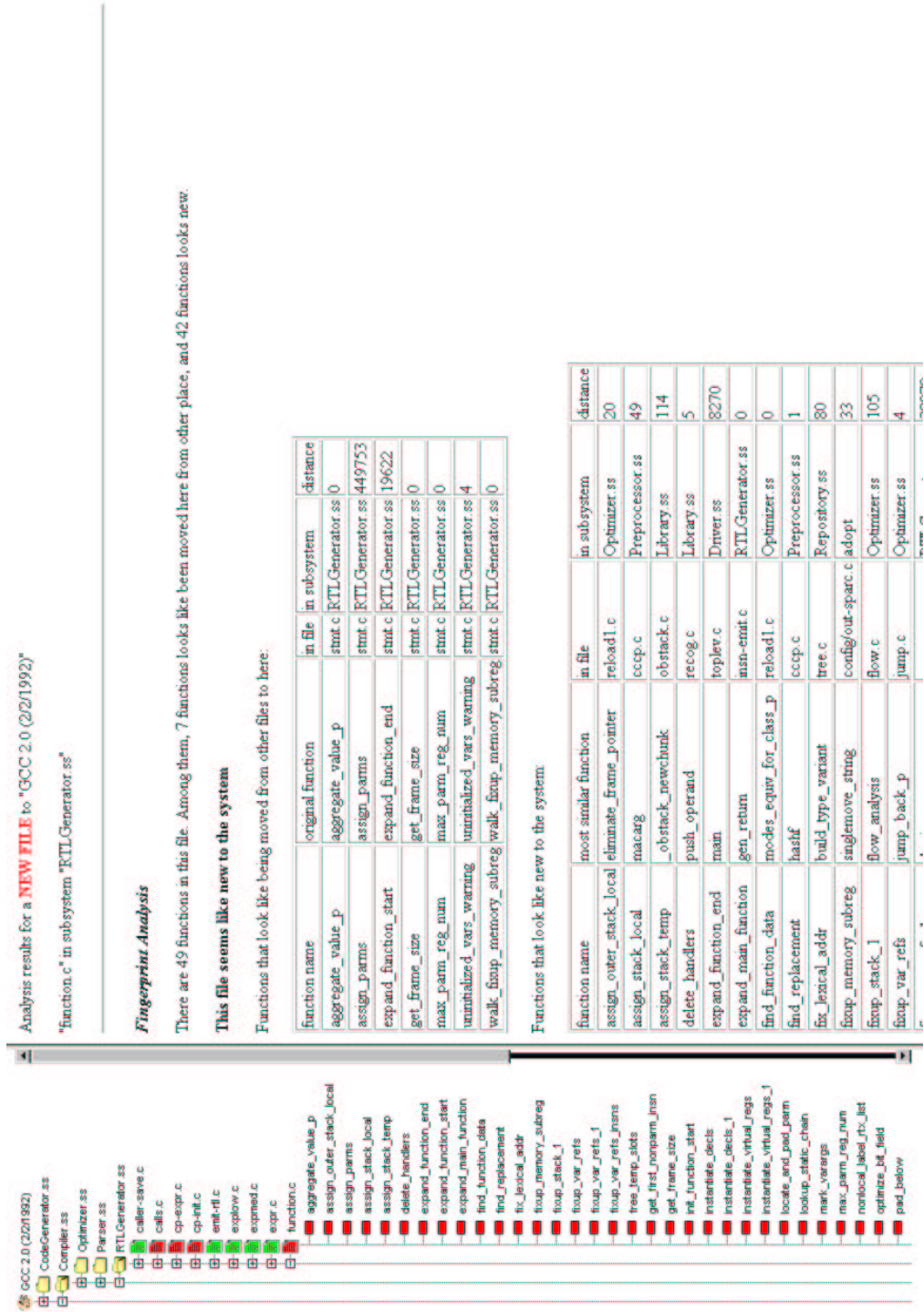


Figure 4.25: File Level Origin Analysis Example One

The best candidate function from table 4.6 is "assign_parms" in file `stmt.c` from subsystem RTL Generator. In this example, the Bertillonage analysis and dependency analysis provide consistent results. Although this is not the case for every analysis, we still can observe many positive results on different new functions.

Callee Function	Caller Functions Unique In Previous Version
<code>build_pointer_type</code>	<code>Datatype</code>
<code>max_reg_num</code>	<code>assign_parms, save_for_inline</code>
<code>move_block_from_reg</code>	<code>assign_parms</code>
<code>Bzero</code>	<code>strength_reduce, assign_parms, cse_main</code>
<code>reg_mentioned_p</code>	<code>copy_rtx_and_substitute</code>
<code>int_size_in_bytes</code>	<code>assign_parms</code>
<code>convert_to_mode</code>	<code>expand_inline_function</code>
<code>gen_rtx</code>	<code>wipe_dead_reg, assign_parms</code>
<code>tree_last</code>	<code>assign_parms</code>
<code>get_last_insn</code>	<code>emit_jump_if_reachable</code>
<code>Bcopy</code>	<code>save_string, assign_parms</code>
<code>expand_expr</code>	<code>assign_parms, expand_asm_operands</code>
<code>list_length</code>	<code>Commontype, symout_types</code>
<code>emit_move_insn</code>	<code>emit_unop_insn</code>
<code>build_decl</code>	<code>implicitly_declare</code>

Table 4.6: Callee analysis on function `assign_parms`

Figure 4.26 is another example that gives good results. We look at a "new" file "enquire.c" in the Configuration subsystem. Bertillonage analysis on all the functions defined in this file reveals that among 62 functions, 13 functions have been found to have origins in file `hard-params.c` from the previous release, such as function `bitpattern`, `ceil_log` and `efprop`. Function `fprop` is very interesting. Even though the code feature distance between this function and its origin from file `hard-params.c` is as large as 12519800, our detection algorithm is still capable to pick them up as perfect match. The remaining 49 functions such as `basic` and `check_defines` do not have origin functions that provide good match results. It means these functions are possible newly written for this release.

For dependency-based analysis, the caller analysis does not provide correct result. This is understandable since only 13 out of 62 functions are from another file. The percentage of

relocated functions are not large enough to generate the usual dependency change pattern to its caller functions that we can use to detect its origin file. However, when we analyze its callees, we get very good result as shown in table 4.7: two out of five callee files of `enquire.c`, which are `regclass.c` and `rtl.c`, list `hard-params.c` as one of the missing files that no longer have call dependency with them. This finding agrees with the result from the previous Bertillonage analysis.

Callee file	Caller Files Unique In Previous Version
"regclass.c"	"hard-params.c"
"cccp.c"	"integrate.c", "combine.c"
"rtl.c"	"flow.c", "loop.c", "print-tree.c", "hard-params.c"
"toplev.c"	"symout.c", "c-parse.y"
"gcc.c"	"flow.c", "c-decl.c"

Table 4.7: Call analysis on file `enquire.c`

4.4.6 Distribution of Evolution Effort

In chapter 3, we introduced the technique of using code feature distance (first proposed by Kontogiannis to detect function cloning) in our Bertillonage analysis to match similar function from the previous release to “new” function in the current release as its “origin”. In this chapter, we introduce the experiment results of using the same technique in measuring the quantified difference between consecutive releases of GCC, and the distribution of the quantified differences among different subsystems of GCC, especially the distribution between front-end components and back-end components. We begin with an introduction of the idea, followed by the experiment result by measuring 29 releases of GCC for over 10 years. Finally, we discuss the feasibility of this approach and possible improvements.

The code feature distance between two consecutive releases of GCC is measured in the following manners:

- If a function (identified by its containing file directory location, filename, and function name) exists in both releases, the code feature distance between the two version of the function is calculated as the Euclidean distance between their fingerprint metrics.



- If a function (identified by its containing file directory location, filename, and function name) exists in only one release, the code feature distance is calculated as the absolute Euclidean distance between its fingerprint metrics and a imaginary function, whose fingerprint metrics are all zero.
- If a file (identified by its directory location and filename) exists in both releases, the code feature distance between the two versions is the sum of all the distance values between the functions defined in the file.
- If a file only exists in one release, the code feature distance is calculated as the sum of all functions defined in this file, which means they are all compared to a null file with null functions defined.
- The overall code feature distance between two releases of GCC is the sum of all distance values of their source files.

The code feature distance give us a rough idea about how much difference exists between the source code of two consecutive releases of GCC using a single numerical value. Here we present some results as we applied this measure to GCC history releases. The measurement is performed in two dimensions.

The first dimension is the distribution of code distance among different subsystems. Which subsystem are changed most for a particular release? This question is commonly asked by software project leader to budget his limited resources for developing the next release of software system.

Figure 4.27 is a code feature distance chart for GCC release 2.0. From the chart, we can see at this evolutionary release, subsystems such as RTL Generator, Parser, and Optimizer are most different from those peers in the previous release. Major changes to RTL Generator and Optimizer correspond to one of the two major new features introduced in GCC 2.0: better RISC CPU support. A much changed Parser is the direct result of the other new feature of GCC 2.0, which is the integration of support for C, C++ and Objective C into one compiler system.

GCC 2.8.0 is mainly a maintenance release. At the time it was released, the EGCS project, which was based on the GCC 2.7.2.3, had already started, so no new features

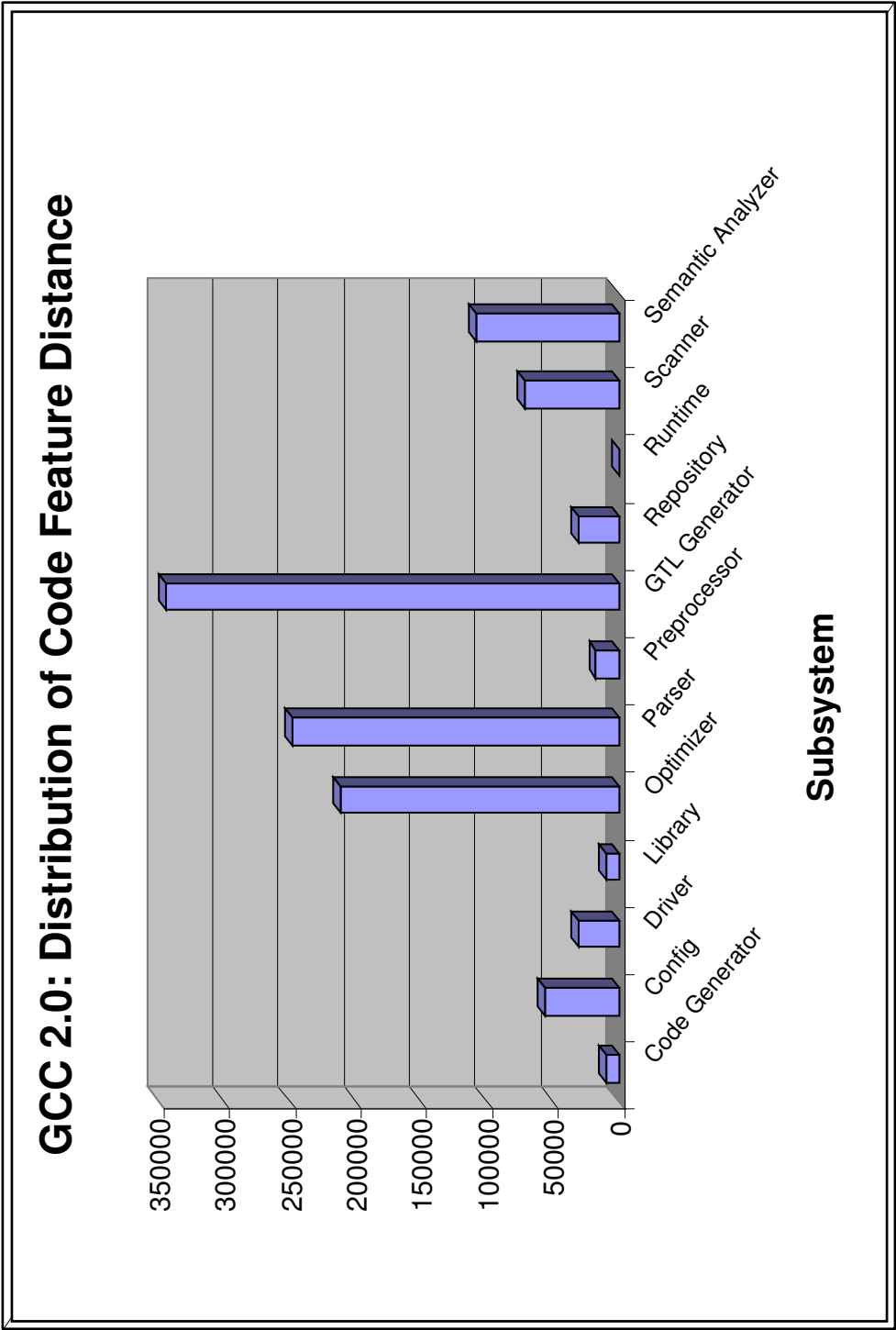


Figure 4.27: Distribution of Code Distance Across Subsystems - GCC 2.0

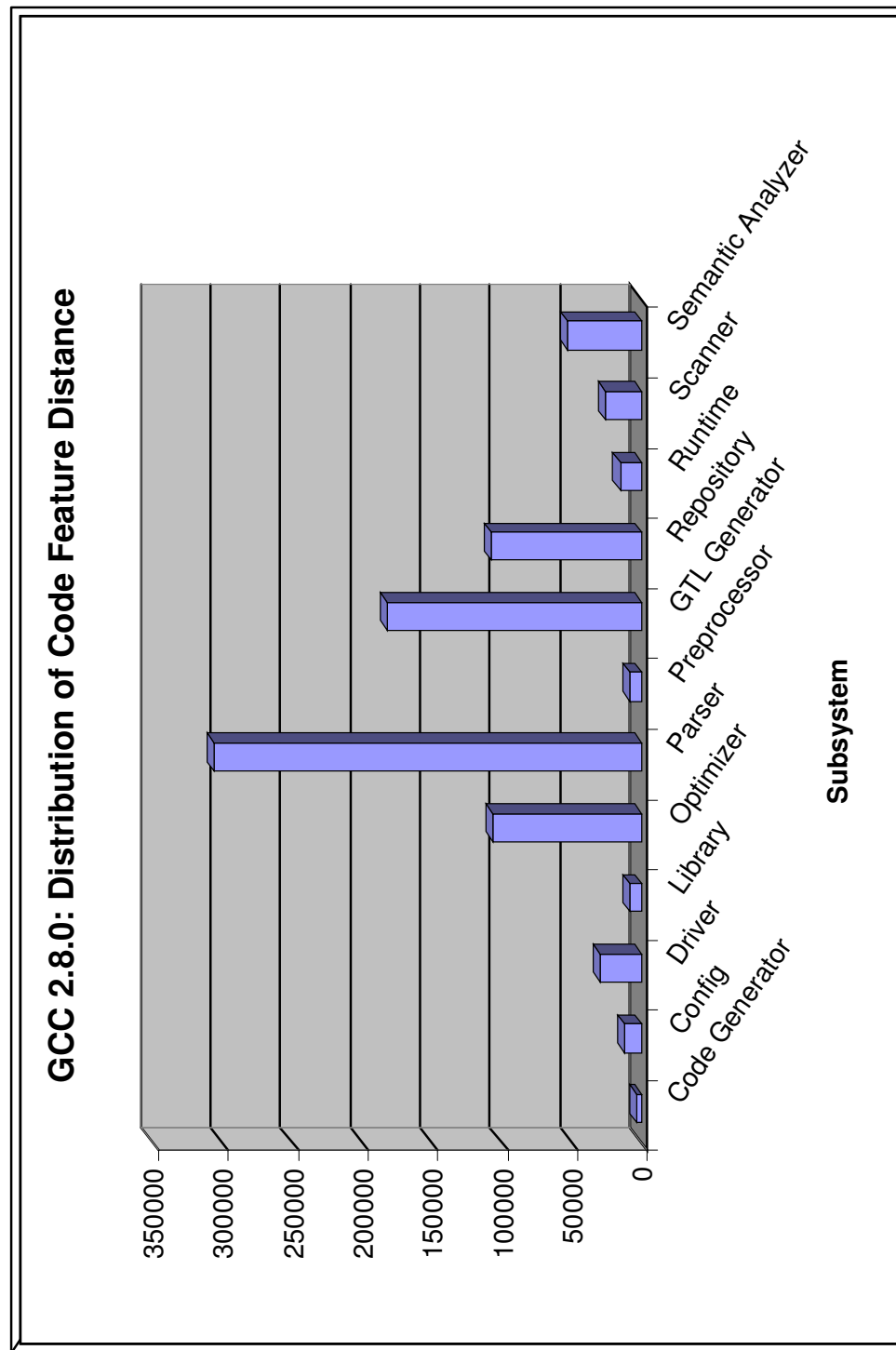


Figure 4.28: Distribution of Code Distance Across Subsystems - GCC 2.8.0

were added into GCC 2.8.0. Most development effort of 2.8.0 was in code cleaning and bug-fixing. In figure 4.28, we can see Parser, RTL Generator, Repository, and Optimizer received most of the changes in their source code.

The second dimension of our measurement is along different releases. For a particular subsystem, at what stage of the product life does it exhibit most active evolution activities? Specific to GCC system, we expect the scanner and parser to evolve fast in the early releases, and then stabilize. For back-end subsystem, such as the optimizer and code generator, we expect to see their continuous evolution, especially at later stage of the project release history. Also, we expect less code feature distance between minor releases, while much more distance between major releases.

Figure 4.29 shows how the development of Code Generator subsystem has been changed along the history of GCC releases. GCC 2.0, EGCS 1.0, EGCS 1.1, and GCC 2.95 are four major milestone releases where there were significant changes to the Code Generator subsystem. GCC 1.42, GCC 2.7.2, GCC 2.8, EGCS 1.0.2, and EGCS 1.1.2 are some of the maintenance releases that also change the internal structure of the Code Generator subsystem significantly.

From these charts, we can observe that this method is more suitable to measure the changes between software releases when it is in stable stage. The condition to use this metric relies on the assumption that very few program entities (function, file, and subsystem) are added or deleted from the system during the period that the code feature distance is measured. For example, the data is more meaningful for GCC releases from 1.37.1 to 1.42, and from GCC 2.0 to GCC 2.8.1. When GCC experienced major changes to its fundamental system structure, like the case in EGCS 1.0 and GCC 2.95, this method generate meaningless results, because every function of the release is actually being compared with a null function that does not exists.

One possible improvement to this simplified model is to create a linear model that incorporates all factors such as code feature distance, AST (Abstract Syntax Tree) and LOC in calculating the single-value code distance between two releases of the software system. When a function is detected in both releases, code feature distance contributes more to the calculation. If the function is newly added, or deleted from one release, LOC and AST should contribute more to the calculation. To create such model, extensive

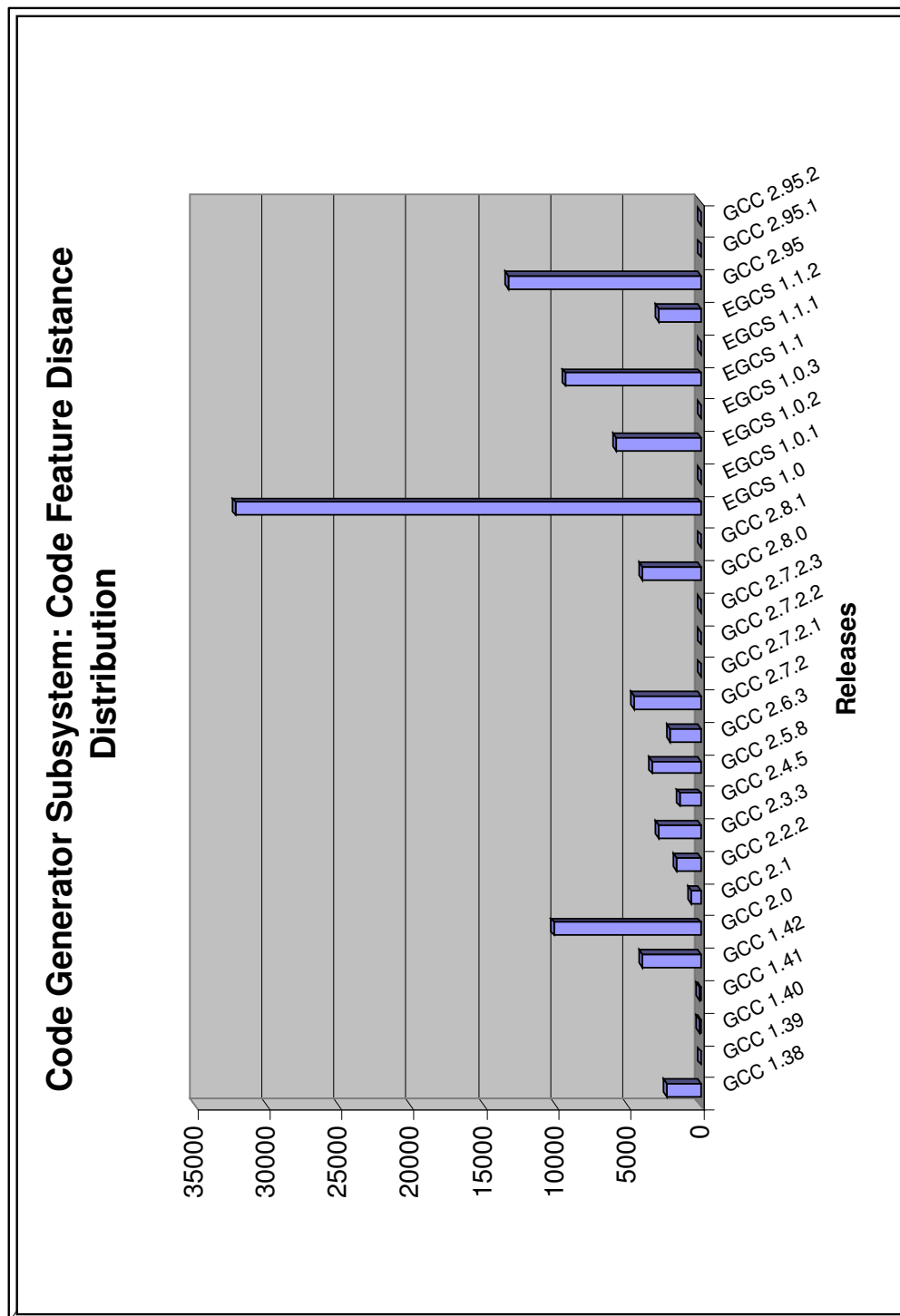


Figure 4.29: Distribution of Code Distance Across Releases - Code Generator

empirical study is needed to find the ideal coefficient of all the contributing factors.

4.5 Summary of Observations

In our case study, we have successfully investigated the evolutionary history of the GCC system using the BEAGLE platform and methodologies introduced in chapter 3. We are able to answer many questions on the history of GCC system, especially those concerned the high level structural changes made to the GCC during several historical milestone releases of GCC and its development branch, EGCS.

We have demonstrated the effectiveness of our methodology of integrating the evolution metrics with visualization techniques in the process of answering these questions. First we select the history releases of GCC that we are interested in compare. Then BEAGLE will present the comparison results in a web browser with three different views. The tree view summarized the evolution status of different program components at all levels of abstraction. The landscape view shows the evolution detail of specific GCC subsystem including the changes of relations between internal subsystems and modules. The metric view provides further information about the complexity and structure change history of specific file or function by listing the history values of representing evolution metrics.

The new naming scheme and source directory structure adopted by EGCS — the experimental release branch of GCC — justify the need for *Origin Analysis* that we have developed and integrated in BEAGLE. Origin analysis reveals the connections between entitles in the new software architecture of EGCS and those in the traditional GCC architecture. It can also be used to discover undocumented refactoring activities as part of the source code maintenance effort in many GCC releases.

Our case study also revealed some interesting evolution patterns of GCC system. Production releases, such as GCC 2.0 to GCC 2.7.2.3, follow a slow but steady evolution path, where new modules are added to the system gradually. Experimental releases, such as EGCS 1.x, took a more aggressive evolution path by rewriting almost half of the system modules and also adopting a new source directory structure. Different GCC subsystems also exhibit distinct evolution patterns. Subsystems in the compiler front-end such as scanner and parser evolved quickly in the early stage, when there were significant changes

in the program language standard supported by GCC, and also when additional program language compiler was integrated in GCC. The back-end subsystems such as code generator and optimizer tend to stabilize during, but experienced quick evolution in experimental releases, and also when new CPU architectures are supported by GCC.

Chapter 5

Summary and Future Work

5.1 Summary

The main contribution of this thesis is to propose an integrated approach to studying software evolution, with emphasis on the evolution of software architecture and internal structural changes of program components. The goal is to automate the history data collection, interpretation, and representation processes so that researchers can conduct more effective empirical studies on software evolution histories. A research platform is designed and implemented to incorporate evolution metrics, software visualization, and structural evolution analysis tools into a unified environment: the *BEAGLE* system.

In the beginning of this thesis, we compare the similarities between software evolution and biological evolution. Then we discussed the motivation for studying software evolution, and the need for an integrated platform and new methods to analyze the structural changes of software system. In the next chapter, we reviewed the existing works on related topics, including the “laws” for software evolution, evolution models based on software metrics, and visualization techniques that represent the change history of software system in graphics. Then we proposed a framework that includes an integrated research platform and specialized analysis methods for software architectural evolution to overcome shortcomings found in exists research approaches.

Our approach starts with the selection of appropriate data source for software evolution study. We chose program source code and the extracted architecture facts from the

source as the primary evolution data source, supplemented by evolution metrics and version control database. We store these archived data in a relational database, which serves as the data repository for BEAGLE. A query interface, implemented in SQL statements, is provided for applications to access the information stored in the data repository, and perform architecture comparison and other analysis. Users can compare the differences between releases and investigate evolution patterns in a web-based user interface.

We also discussed the concept of *Origin Analysis*. Origin analysis is a technique to relate “new” program entities such as functions and source files found in the more recent release with those entities that existed in the previous release, but are no longer present in the later release. These entities have very similar code features and dependencies with other program entities. These types of changes are usually caused by system rearchitecturing and code refactoring activities. We demonstrated two methods that match “new” program entities with their “origins” in the previous release, if they do exist. The first method compares the code feature of candidate functions, and chooses the one with the shortest distance between the two code feature vectors. The second method analyzes the change patterns of functions that have dependencies with the “new” program entities to find out their “origins”.

We believe our approach is helpful for the software developer who is interested in understanding the evolution history of a software system. The BEAGLE platform demonstrates that an integrated environment is invaluable tool for answering many questions related to the evolution of software system, as shown in our case study using the evolution of GNU Compiler Collection as example. For instance, we discovered that GCC exhibited different evolution pattern in its experimental release stream (EGCS 1.x and 2.x) comparing to its production release stream (GCC 1.x and 2.x). BEAGLE is also able to tell us what portion of the EGCS 1.0 release source code can be traced back to the previous GCC release, and what portion is complexly newly developed for this release.

The main contributions of this thesis is summarized as follows:

- We provided an integrated environment for researchers to conduct empirical studies on the evolution of software systems. It offers many research capabilities such as evolution metrics, visualization of source code change history, peer or group release comparison, structural change analysis, as well as a power query and browsing

interface.

- We also developed an analysis technique that tracks the structural changes of the software code architecture when its directory structure and/or naming scheme for files and functions are changed in the newer release. This technique assists us in understanding the architectural differences and relations between traditional GCC releases and the experimental EGCS releases, where the naïve name-based comparison techniques previously failed to handle.

5.2 Future Work

In our approach, we have developed a web application that allows users to access the query and analysis facility provided by BEAGLE from any client machine connected to the Internet. On the other hand, many open source software projects are using web-based CVS tools to manage source code check-in and check-out. One of the possible extensions to BEAGLE is to integrate it with web-based source control system, so that when new code is checked in, the architecture facts will be automatically extracted from the new source code and stored in the BEAGLE's data repository. The submitter could then view the changes to the system architecture shortly after she submitted the new code.

We presented two methods for origin analysis in this thesis. We have also examined their effective using many examples from GCC history releases. However, to be able to provide more accurate analysis results, it is essential to find out more analysis methods. By combining more than one origin analysis results will definitely improve the analysis accuracy.

In order to further examine our BEAGLE platform, and to collect more first hand information on the architectural evolution of large-scale long-life software systems, it would be useful to use BEAGLE to analyze more software systems. Possible candidates are those open source software systems whose extracted architecture have been studied using PBS or other reverse engineering tools, such as the Linux kernel, VIM editor, Netscape Mozilla, and Linux Nautilus file manager system. The BEAGLE system will enable us to expand our knowledge on software evolution through effective empirical studies on large and successful software projects.

Bibliography

- [1] http://www.csst-technologies.com/genericieee_software_maturity_metric.html. Website.
- [2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, U.S.A., 1986.
- [3] T. Ball and S. G. Eick. Software visualization in the large. *IEEE Computer*, 29(4), April 1996.
- [4] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Reading, MA, U.S.A., 1998.
- [5] A. Binkley and S. Schach. Validation of the coupling dependency metric as a predictor of runtime failures and maintenance measures. In *Proc. of the 20th International Conference on Software Engineering*, 1998.
- [6] I. Bowman and R. Holt. Software architecture recovery using conway’s law. In *Proc. of CASCON’98*, 1998.
- [7] I. Bowman, R. Holt, and N. V. Brewster. Linux as a case study: Its extracted software architecture. In *Proc. of Intl. Conf. on Software Engineering (ICSE’99)*, 1999.
- [8] Ivan Thomas Bowman. Architecture recovery for object-oriented systems. Master’s thesis, University of Waterloo, 1999.

- [9] E. Burd, S. Bradley, and J. Davey. Studying the process of software change: An analysis of software evolution. In *Proc. of the 7th Working Conf. on Reverse Engineering (WCRE'00)*, 2000.
- [10] E. Burd and M. Munro. An initial approach towards measuring and characterizing software evolution. In *Proc. of the Sixth Working Conference on Reverse Engineering (WCRE'99)*, 1999.
- [11] Edited by Chris DiBona, Sam Ockman, and Mark Stone. *Open Sources: Voices from the Open Source Revolution*. read at <http://www.oreilly.com/catalog/opensources/book/tiemans.html>. O'Reilly and Associates, Cambridge, MA, U.S.A., 1999.
- [12] S. Demeyer, S. Ducasse, and M. Lanza. A hybrid reverse engineering approach combining metrics and program visualization. In *The 6th Working Conference on Reverse Engineering (WCRE'99)*, 1999.
- [13] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mocku. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1), January 2001.
- [14] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2), February 2001.
- [15] Understand for C++ from Scientific Toolworks. <http://www.scitools.com/ucpp.html>. Website.
- [16] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proc. of the Intl. Conf. on Software Maintenance (ICSM'98)*, 1998.
- [17] H. Gall, M. Jazayeri, R. Kloesch, and G. Trausmuth. Software evolution observations based on product release history. In *Proc. of the 1997 Intl. Conf. on Software Maintenance (ICSM '97)*, 1997.

- [18] H. Gall, M. Jazayeri, and C. Riva. Visualizing software release histories: The use of color and third dimension. In *Proc. of the IEEE Intl. Conf. on Software Maintenance (ICSM99)*, 1999.
- [19] M. Godfrey and Q. Tu. Growth, evolution, and structural change in open source software. In *International Workshop on Principles of Software Evolution (IWPSE'01)*, 2001.
- [20] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proc. of the Intl. Conf. on Software Maintenance (ICSM'00)*, 2000.
- [21] T. L. Graves, A. F. Karr, J.S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7), July 2000.
- [22] R. Holt and J.Y. Pak. Gase: visualizing software evolution-in-the-large. In *Proc. of the 3rd Working Conf. on Reverse Engineering (WCRE'96)*, 1996.
- [23] R. C. Holt. An introduction to ta: The tuple-attribute language. Website <http://swag.uwaterloo.ca/pbs/papers/ta.html>.
- [24] F. P. Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, Reading, MA, U.S.A., 1995.
- [25] J. P. D. Keast, M. G. Adams, , and M. W. Godfrey. Visualizing architectural evolution. In *Proc. of ICSE'99 Workshop on Software Change and Evolution (SCE'99)*, 1999.
- [26] C. F. Kemerer and S. Slaughter. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, 25(4), July/August 1999.
- [27] T. M. Khoshgoftaar and R. M. Szabo. Improving code churn prediction during the system test and maintenance phases. In *Proc. of the Intl Conf. on Software Maintenance (ICSM'94)*, 1994.
- [28] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Working Conference on Reverse Engineering (WCRE'97)*, Amsterdam, Netherlands, 1997.

- [29] O. Kwon, G. Shin, C. Boldyreff, and M. Munro. Maintenance with reuse: An integrated approach based on software configuration management. In *Proc. of 6th Asia-Pacific Software Engineering Conference (APSEC'99)*, 1999.
- [30] B. Lagu, D. Proulx, E. Merlo, J. Mayrand, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proc. of the Intl Conf. on Software Maintenance (ICSM'97)*, 1997.
- [31] M. M. Lehman. Programs, life cycles and laws of software evolution. In *Proc. IEEE Special Issue on Software Engineering*, pages 1060–1076, 1980.
- [32] M. M. Lehman. Metrics and laws of software evolution - the nineties view. In *Proc. Metrics 97 Symp*, 1997.
- [33] M. M. Lehman. Quantitative studies in software evolution - from os/360 to feast. In *Workshop on Process MOdelling and Empirical Studies of Software Evolution*, 1997.
- [34] M. M. Lehman, D. Perry, and J. Ramil. On evidence supporting the feast hypothesis and the laws of software evolution. In *Proceedings of Metrics'98*, 1998.
- [35] P. Lindsey, Y. Liu, and O. Traynor. A generic model for fine grained configuration management including version control and traceability. In *Proc. of Australian Software Engineering Conference (ASWEC'97)*, 1997.
- [36] M. Mattsson and J. Bosch. Observations on the evolution of an industrial OO framework. In *Proc. of the IEEE Intl. Conf. on Software Maintenance (ICSM'99)*, 1999.
- [37] A. Mockus, S. G. Eick, T. Graves, and A. F. Karr. On measurement and analysis of software changes. Technical report, Bell Labs, Lucent Technologies, 1999.
- [38] S. Muthanna, K. Kontogiannis, K. Ponnambalam, and B. Stacey. A maintainability model for industrial software systems using design level metrics. In *Working Conference on Reverse Engineering (WCRE'00)*, 2000.
- [39] J. Nadler and T. Josling. Writing a compiler front end. online manual.

- [40] M.C. Ohlsson and C. Wohlin. Identification of green, yellow and red legacy components. In *Proc. of Intl. Conf. on Software Maintenance (ICSM'98)*, 1998.
- [41] D. E. Perry. Dimensions of software evolution. In *Proc. of the 1994 Intl. Conf. on Software Maintenance (ICSM'94)*, 1994.
- [42] D. E. Perry, A. A. Porter, and L. G. Votta. Empirical studies of software engineering: a roadmap. In *ICSE - Future of Software Engineering Track*, pages 345–355, 2000.
- [43] J. F. Ramil and M. M. Lehman. Metrics of software evolution as effort predictors - a case study. In *Proc. of the Intl. Conf. on Software Maintenance (ICSM'00)*, 2000.
- [44] E. Raymond. *The Cathedral and the Bazaar*. O'Reilly and Associates, Sebastopol, CA, U.S.A., 1999.
- [45] I. Robertson. Evolution in perspective. In *Proc. of the Intl. Workshop on the Principles of Software Evolution*, 1998.
- [46] M. Shaw and D. Garlan. *Software Architecture: Perspective on an Emerging Discipline*. Prentice-Hall, 1996.
- [47] R. Stallman. Using and porting the gnu compiler collection. online manual, 2001.
- [48] T. Systa, P. Yu, and H. Mller. Analyzing java software by combining metrics and program visualization. In *Conference on Software Maintenance and Reengineering (CSMR'99)*, 2000.
- [49] L. Tahvildari, R. Gregory, and K. Kontogianni. An approach for measuring software evolution using source code features. In *Proc. of Sixth Asia Pacific Software Engineering Conference (APSEC'99)*, 1999.
- [50] E. Thomsen. *OLAP solutions: Building Multidimensional Information Systems*. John Wiley and Sons, New York, U.S.A., 1997.
- [51] J. B. Tran. Software architecture repair as a form of preventive maintenance. Master's thesis, University of Waterloo, 1999.

- [52] Q. Tu and M. Godfrey. Exploring structural change and architectural evolution. <http://plg.uwaterloo.ca/~migod/papers/beagle-csermay2001.ppt>.
- [53] Qiang Tu and Michael W. Godfrey. The build-time software architecture view. In *Proc. of Intl. Conf. on Software Maintenance (ICSM 2001), Florence, Italy*, 2001.
- [54] A. von Mayrhauser, J. Wang, M.C. Ohlsson, and C. Wohlin. Deriving a fault architecture from defect history. In *Proceedings of the 10th International Symposium on Software Reliability Engineering*, 1999.
- [55] M. Wein, S.A. MacKay, D.A. Stewart, C.A. Gauthier, and W.M. Gentleman. Evolution is essential for software tool development. In *Proc. of the Second Working Conference on Reverse Engineering (WCRE'95)*, 1995.
- [56] C. Wohlin and M. C. Ohlsson. Reading between the lines: An archival study of software from nine releases. In *Proc. of ICSE'99 Workshop on Software Change and Evolution (SCE'99)*, 1999.