

An Integrated Approach for Studying Architectural Evolution

Qiang Tu and Michael W. Godfrey

Software Architecture Group (SWAG)

Department of Computer Science, University of Waterloo

email: {qtu,migod}@swag.uwaterloo.ca

Abstract

Studying how a software system has evolved over time is difficult, time consuming, and costly; existing techniques are often limited in their applicability, are hard to extend, and provide little support for coping with architectural change. This paper introduces an approach to studying software evolution that integrates the use of metrics, software visualization, and origin analysis, which is a set of techniques for reasoning about structural and architectural change. Our approach incorporates data from various statistical and metrics tools, and provides a query engine as well as a web-based visualization and navigation interface. It aims to provide an extensible, integrated environment for aiding software maintainers in understanding the evolution of long-lived systems that have undergone significant architectural change. In this paper, we use the evolution of GCC as an example to demonstrate the uses of various functionalities of BEAGLE, a prototype implementation of the proposed environment.

1 Introduction

Software systems must evolve during their lifetime in response to changing expectations and environments. While change is inevitable in software systems, it is also risky and expensive, as careless changes can easily jeopardize 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 these goals, software engineers have to learn from the history. By studying how successfully maintained software systems had 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 change patterns of software systems, explores the un-

derlying mechanisms that affect software changes, and provides guidelines for better software evolution processes.

To discover the evolution patterns 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 paper, we describe an approach towards efficient navigation and visualization of evolution histories of software architectures. Furthermore, we also introduce several methods to track and analyze the software structural changes from past releases. Our goal is to explore new approaches to help software maintainers better understand software evolution. The evolution history of a real-world software system, the GCC compiler suite, is used to illustrate the effectiveness of our approaches.

2 Challenges to Software Evolution Research

Many studies on software evolution emphasize the statistical changes of the software system by analyzing its evolution metrics [13, 7, 5, 14]. Apart from several visualization tools [8, 11, 6], little work has been done to help understanding the nature of the evolution of software architecture.

Performing a long-ranging and detailed evolutionary case study of a software system presents several difficult technical problems. Some previous studies have examined only a small number of the many versions of the candidate system, while in other cases the time period studied has been relatively short, which makes it difficult to generalize the results. The enormous amount of work required by large-scale empirical study makes it almost impossible without specialized tools and an integrated environment dedicated for software evolution research, such as the *SoftwareChange* environment [3]. Strong tool and environment support has been proven a key factor in conducting a successful empirical study on software evolution.

We believe there are three major challenges that must

be overcome in software evolution research. These obstacles limit our ability to understand the history of software systems using effective empirical study, and furthermore prevent us from generalizing our observations into software evolution theory.

The first challenge is how to organize the enormous amount of historical data so that the appropriate information can be accessed quickly and easily.

The second challenge is how to incorporate different research techniques of software evolution into one integrated platform. Evolution metrics are precise, flexible, and can be used for statistical analysis. Visualization diagrams provide the overview of the evolution history and have visual appeal to the users. When used together, they could effectively boost the productivity of software evolution study.

The third challenge is how to analyze the structural changes of software systems. Traditional name-based comparison techniques are not effective when the software system adopts a different naming scheme for program modules or has changed its source directory structure. New research methods must be explored to solve this challenging problem.

In this paper, we demonstrate an integrated environment called *Beagle* that tries to address all these challenges. We construct *Beagle* as a web application platform that hosts several services related to software evolution studies, and provides them with a common infrastructure that includes an evolution data repository, a query interface, and a browser-based user interface. Individual services are provided by specific components that are plugged into *Beagle*. Users of *Beagle* can develop their own service components and customize *Beagle* to serve their research interests.

3 Discussion of Methodologies

In this section, we introduce an approach that attempts to answer 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.

We first discuss how the data are selected and stored in the platform. Then we discuss how different analysis methods can be integrated in the platform, and they can be applied to solve problems in software evolution research.

3.1 History Data Repository

3.1.1 Data Source

As a software system evolves, the various activities related to its evolution produce many new and changed artifacts. The archives of each of these artifacts reveal one or more aspects of the software evolution. Program source code,

Makefiles, compiled binary libraries, and executables are the main products of software development activities. Artifacts related to the requirements specification and architecture design include functional and non-functional requirements specifications, user manuals, and design documents. Also there are artifacts related to testing and maintenance activities, such as defect reports, change logs, new feature requests, and automatic testing scripts.

In this paper, we have chosen to focus on the source code archives of Open Source Software (OSS) systems. The reason is that most OSS projects maintain complete archives of program source code and a version control database for history releases on their FTP sites, and access to the OSS source code is free and unrestricted. Furthermore, there are many software reverse engineering techniques that can extract and rebuild some of the software architecture information from the source code. Consequently, we selected source code and the extracted software architecture facts from reverse engineering tools as the primary data sources for studying the evolution of software architecture.

3.1.2 Software Architecture Model

In this paper, we use the term *software architecture* to refer to the structure of a software system, emphasizing the organization of the 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.

3.1.3 Evolution Metrics

Code-based evolution metrics provide valuable information to study the evolution attributes of individual program entities. The metrics we selected include basic metrics such as 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, the number of input/output statements, as well as composite metrics including S-complexity, D-complexity, Albrecht metric, and Kafura metric. The details of these evolution metrics and their role in analyzing software architecture evolution will be discussed in section 3.3.1.

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 and the version identifier.

3.2 Navigation of Evolution Information

3.2.1 Incorporating Evolution Metrics with Software Visualization

Previous work in incorporating software metrics with visualization techniques in program comprehension have been discussed by Demeyer *et al.* [2] and Systa *et al.* [15]. 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. 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.

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

3.2.2 Comparing Differences between Releases

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, such as Gall's colored 3D evolution graph [6]. The other approach focus on illustrating the architectural differences between two releases, as seen in GASE [8] and KAC [11] systems.

Our method is to display the two types of evolution visualization graph at the same time. First, we provide a tree-like view 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" of the tree represent subsystems and program modules. The "leaves" of the tree represent functions defined in the program modules. The 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 their saturations to model the evolution status of each entity. 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), different levels of red are used to represent their relative

"ages". Entity in vivid red came into the system most recently, while 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, as shown in figure 1.

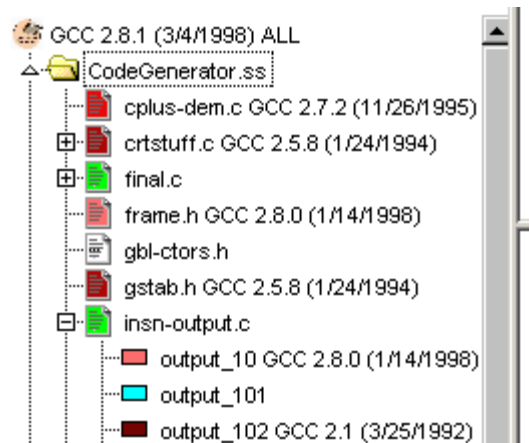


Figure 1. Structure diagram for GCC v2.8.1 compared to several other versions

Another diagram is shown next to the tree diagram: it is designed to emphasize the architecture 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 its schema by adding evolution related entities and relations, such as new files, changed subsystems, or deleted data references. The right-bottom portion of Figure 5 shows an instance of dependency diagram.

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 5 shows a prototype implementation of the ideals discussed above. The detail of this screen and its components will be discussed in section 4.3.

3.3 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 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 [4] (low-level modification to the program structure) activities involve reorganizing the program source code by relocating functions or files to other places, with little change actually made to the program entity. Meanwhile, their names 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.

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.

3.3.1 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 such as 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 identify a small group of “suspects” quickly and easily from a population of tens of thousands of candidates. We can then use other advanced techniques that requires more computing power to filter out

the real culprit 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 [12].

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 exact 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 “disappeared” list that contains functions that existed in the immediate previous release, but do not exist in the current release. We 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.
3. Select the top five best matches. Among these good matches, compare their function name with the “new” function being matched. Choose the one whose function name is the most similar to the “new” function. If none of them has good name match, pick the best match that has the shortest Euclidean distance.

The last step of comparing function name works as a filter to discard mismatched functions, since it is possible that two irrelevant functions happen to have very similar Bertillonage measurements. Here is an example from the case study of GCC that illustrate 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:


```

1. combine in fold-const.c:
d=1005745.47

2. recog_4 in insn-recog.c:
d=2496769.23

3. insn-recog.c in recog_5.c:
d=7294066.05

4. fprop in hard-params.c:
d=8444858.78

5. build_binary_op_noddefault in c-typeck.c:
d=8928753.44

```

The obvious choice should be match number 5, which has the exact filename as the “new” function (however, this does not represent all the other cases because many pairs of good matches have different function names). **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.

3.3.2 Dependency Analysis

We use the following analogy to explain the basic idea behind *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. **There 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: **Find all the caller functions of this “new” function.** 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.** Any functions that are selected more than once are candidates for the origin of the “new” function.
3. Analyze the callee functions: Find all the functions that this “new” functions calls in the reference release. 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.** 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.**

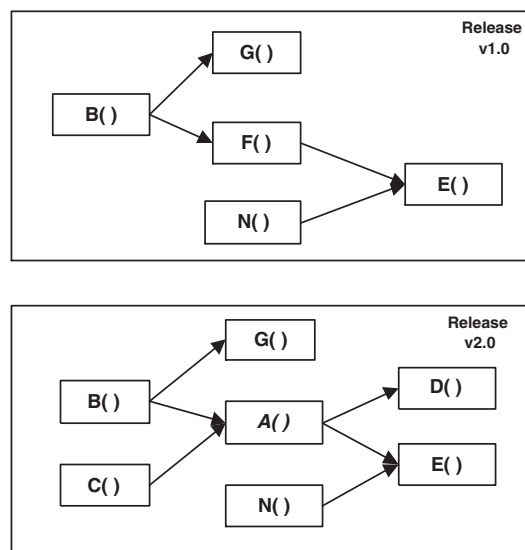


Figure 2. Call-Relation Change Analysis

Figure 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.

- *Caller Analysis*: Function A is called by Function B and C in v2.0. However, only B exist 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.

4 BEAGLE: An Integrated Environment

To validate the research techniques we have just discussed, we have built an research platform called *BEAGLE*¹ that integrates several 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 resembles a three-tier web application. 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, the 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.

4.1 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 a relational database. Functional components in the logic tier access

¹BEAGLE is named 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 hope that our tool will also prove to be a useful vehicle for exploring evolution.

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.

4.1.1 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. 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 [16].

The BEAGLE data repository has four fact tables. They model the system structure and relations between program entities at four abstract levels. Each level of architecture fact is stored in its own table for all the history releases. *Entity-Level Table* stores the low-level architecture facts pulled out from the source code using source code extractor *cfx*. *File-Level Table* stores the entity level facts induced from entity-level facts using relational calculus formulas and *grok* [9]. *High-Level Table* stores the high level facts, where relations between files are a set of higher-level relations (*userproc*, *usevar*, and *implementby*) that are merges from the relations modeled by file-level facts. *Architecture-Level Table* contains not only relations between program files, but also higher level architecture facts between file and subsystem, subsystem and subsystem, and containment relations between files, low-level subsystems, and high-level subsystem.

Surrounding the fact tables, there are six dimension tables. They provide additional information for entities and dependencies modeled in the fact tables, such as the version number, release date, build configuration, function complexity and file complexity. Figure 3 shows the relations between fact tables and dimension tables, as well as the schema of each table.

4.1.2 Repository Access Interface and Comparison Query

To access the data stored in BEAGLE repository, we provide a query interface in the data tier so that all the functional components in the logic tier can query and “slice and dice” the collected history data for navigation and analysis purposes.

For example, we want to find all the 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 the SQL statement that implements this query:

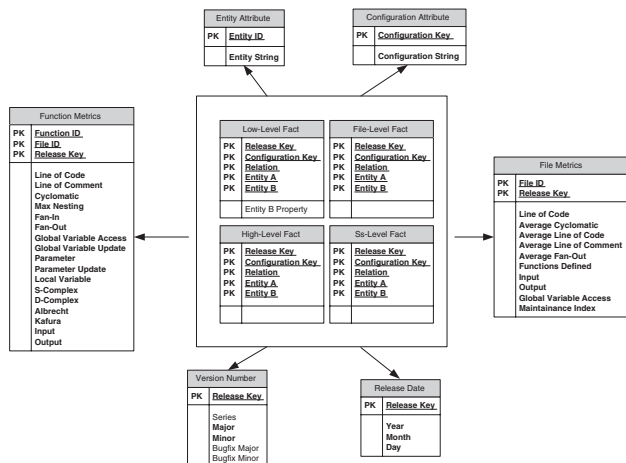


Figure 3. Schema of BEAGLE Data Repository

```
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 exist in version v2, but not in version v1. Then it refers to the Entity Attribute table to convert the entity key to entity name.

Table 1 shows a section of the output of the above query when comparing GCC 2.7.2.3 and GCC 2.8.0.

Function	File	LOC	KAFURA
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

Table 1. Result of the example query

4.2 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*.

4.2.1 Version Comparison and Evolution Visualization

In BEAGLE, we adopt a novel approach to visualize the difference between various releases. Figure 5 shows the screen shot of BEAGLE visualizing the architecture differences between GCC version 2.0 and GCC version 2.7.2.

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* indicated that parent-level entities, such as subsystems and files, contain either “new” entities or have entities deleted from them. If the none of the contained entities 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 “red”s with various levels of saturation are used to differentiate their “tenure” within the system. At the left bottom of figure 5 we can see three new files under “Scanner” subsystem. `c-pragma.c` first appeared in GCC at version 2.3.3. It is the oldest among all three 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 5 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*.

4.2.2 Origin Analysis

The *origin analysis* component applies both Bertillonage analysis and dependency analysis to exam every “new” function in the selected release with its immediate previous release to find out its “origin”, and examine all the “deleted” functions with its immediate next release to find out its “destination”.

Origin analysis is useful to understand the following two types of changes. In the first case, source files are moved to new locations in the later releases, usually to new directories, as a maintenance effort to reorganize the source directory structure. In the second case, related source files are given common prefix or suffix in their file names for easier understanding of their responsibility in the system. These types of changes make architectural comparison tools such as GASE and KAC ineffective, because they treat a file with a different path or name as a completely new file. However, with origin analysis, Beagle is able to related these apparently “new” files with their true “origins” from the previous release, thus discovering many “undocumented” software evolution activities.

4.3 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:
EGCS 1.0.1 (1/4/1998)
EGCS 1.0.2 (2/12/1998)
EGCS 1.0.3 (2/12/1998)
EGCS 1.1 (3/4/1998)

Choose A Build Configuration:

Show the Software Architecture of ☒ Newest Release ☐ Oldest Release

Figure 4. Query Interface

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 4. 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. 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 5.

More discussions on the design and implementation details of *BEAGLE* could be found in one of the authors' master thesis [17].

4.4 Case Study: From GCC To EGCS

EGCS is an experimental development project after the release of GCC 2.7.x [10, 1] to provide better support for the new 1998 ANSI C++ standard. There are many differences between the source code structure of 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.

With *Origin Analysis* we can analyze whether a particular function in a EGCS release has a correspondent from the GCC release, or it is newly written for the EGCS project. 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 1.0. 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. For example, the result of “Origin Analysis” on file `gcc/c-decl.c` in EGCS 1.0 shows that among the 70 functions defined in this file, 41 functions can be traced back to their origin functions defined in the previous GCC release using Bertillonage analysis method. 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

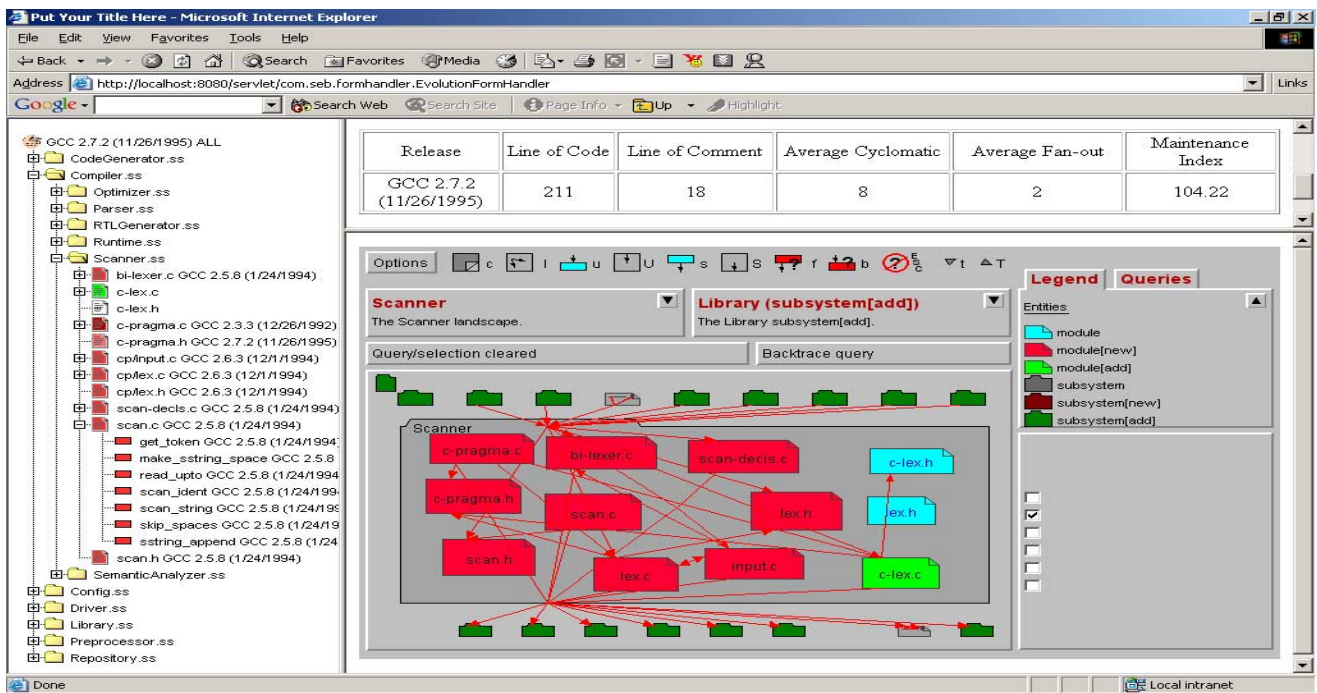


Figure 5. Architecture Comparison

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. Here we present the result for two representing subsystems of GCC: Parser and Code Generator. One is from 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 define 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, especially for a subsystem that is based on mature techniques such programming language parser. Table 2 lists the complete result.

File	Func	New	Old	Type
gcc/c-aux-info.c	9	0	9	Mostly Old
gcc/fold-const.c	44	15	29	Mostly Old
gcc/objc/objc-act.c	167	17	150	Mostly Old
gcc/c-lang.c	16	14	2	Mostly New
gcc/cp/decl2.c	57	50	7	Mostly New
gcc/cp/errfn.c	9	9	0	Mostly New
gcc/cp/except.c	25	20	5	Mostly New
gcc/cp/method.c	30	26	4	Mostly New
gcc/cp/pt.c	59	57	2	Mostly New
gcc/except.c	55	52	3	Mostly New
gcc/c-decl.c	70	29	41	Half-New
gcc/cp/class.c	61	31	30	Half-Half
gcc/cp/decl.c	134	84	50	Half-Half
gcc/cp/error.c	31	16	15	Half-Half
gcc/cp/search.c	81	40	41	Half-Half

Table 2. Origin analysis - parser

Table 3 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

tem 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	Func	New	Old	Type
gcc/cplus-dem.c	36	36	0	Mostly New
gcc/crtstuff.c	5	5	0	Mostly New
gcc/insn-output.c	107	95	12	Mostly New
gcc/final.c	33	20	13	Half-Half
gcc/regclass.c	20	12	8	Half-Half

Table 3. Origin analysis - code generator

5 Conclusions

The main contribution of this paper is to demonstrate how we can conduct empirical study on software evolution using an integrated approach, with an emphasis on the evolution of software architecture and internal structure of program components. We have constructed an integrated platform that automates the history data collection, interpretation, and representation processes. It incorporates different research methods such as evolution metrics, software visualization, and structural evolution analysis tools to allow the user to examine the evolution patterns of software systems from various aspects. We also developed a concept of *origin analysis* as a set of techniques for reasoning about structural and architectural changes when the newer system release adopts a complete new source code structure or naming scheme.

The current implementation of Beagle is still at the prototyping stage. We are planning to add caching mechanism to the next release of Beagle to minimize loading time when the software structure tree is constructed. We are also planning to create new types of visualization graph in Beagle after an origin analysis is performed on a release to explicitly show the “origin relation” between new program entities and their origins from the previous release.

The example of GCC evolution shown in the case study is only one of several cases discussed in author’s thesis[17]. We are planning to conduct more analyzes on different subsystems of GCC at its different release stages, with the feedback and evaluation from real GCC developers.

The results of this paper and the *BEAGLE* environment we have implemented enable us to conduct large scale empirical studies on software projects with long and successful histories to expand our knowledge on software evolution.

References

- [1] E. by Chris DiBona, S. Ockman, and M. Stone. *Open Sources: Voices from the Open Source Revolution*. O’Reilly and Associates, Cambridge, MA, U.S.A., 1999.
- [2] S. Demeyer, S. Ducasse, and M. Lanza. A hybrid reverse engineering approach combining metrics and program visualization. In *The 6th Working Conf. on Reverse Engineering (WCRE’99)*, 1999.
- [3] 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.
- [4] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] R. Holt and J. Pak. Gase: visualizing software evolution-in-the-large. In *Proc. of the 3rd Working Conf. on Reverse Engineering (WCRE’96)*, 1996.
- [9] R. Holt. Pbs: Portable bookshelf tools. available at <http://swag.uwaterloo.ca/pbs/>, 1997.
- [10] <http://www.gnu.org/software/gcc/>. GCC homepage. Website.
- [11] 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.
- [12] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Working Conf. on Reverse Engineering (WCRE’97)*, Amsterdam, Netherlands, 1997.
- [13] M. M. Lehman. Metrics and laws of software evolution - the nineties view. In *Proc. Metrics 97 Symp*, 1997.
- [14] 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.
- [15] T. Systa, P. Yu, and H. Mller. Analyzing java software by combining metrics and program visualization. In *Conf. on Software Maintenance and Reengineering (CSMR’99)*, 2000.
- [16] E. Thomsen. *OLAP solutions: Building Multidimensional Information Systems*. John Wiley and Sons, New York, U.S.A., 1997.
- [17] Q. Tu. On navigation and analysis of software architecture evolution. Master’s thesis, University of Waterloo, 2002.