Toward an improved understanding of software change

by

Lijie Zou

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, 2003

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.
I understand that my thesis may be made electronically available to the public.

Lijie Zou

Acknowledgements

I would like to thank many people who have helped me in the preparation and writing of this thesis. My supervisor, professor Michael W. Godfrey, has offered great help in the development and improvement of thesis contents. In the process of revision, professor Richard C. Holt and Charlie Clarke have given their thoughtful comments.

I am also thankful to all the members in SWAG group. They have developed wonderful tools that my experiment has based on. Also, numerous interesting discussions with them have made the research process really pleasant and enjoyable.

Finally, I would like to thank my husband Yi Lai, my parents Xishen Zou and Yunzhen Wang, and my sister Limin Zou, for their continuous support for my study in Waterloo.

Abstract

Structural changes, including moving, renaming, merging and splitting are important design change decisions made by programmers. However, during the process of software evolution, this information often gets lost. Recovering instances of structural changes in the past, as well as understanding them, are essential for us to achieve a better understanding of how and why software changes.

In this thesis, we propose an approach that helps to recover and understand the lost information of structural changes.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Studying Software Evolution

Software must evolve over time to adapt to its ever-changing environment. The research discipline of software evolution explores technical and managerial activities that can help to ensure software continuously meet its organizational and business objectives.

Empirical studies of how software systems evolve have revealed that:

- Different software systems evolve differently. However, some common patterns and "rules" have been observed to hold (*e.g.*, Lehman's laws [31]).
- Examining a software system from different perspectives, such as growth of code base and user-visible features, helps to provide different views of software evolution.
- Visualization is often useful in both understanding the software evolution process and discovering interesting phenomena or patterns.

Empirical studies differ in various ways, such as the type of software system under study and the analysis technique being used. Among them, one important difference is how they characterize the idea of *software changes*, the fundamental activity in software evolution. Some researchers study lines of code (LOC) or number of modules to measure growth of system or subsystem. Some derive metrics from change logs or other sources to discover patterns of other properties of evolving software.

To understand the *nature* of software evolution, we need a better understanding of software change itself. That is, in additional to using a single metric summarizing all the changes and try to understand them, we should be able to answer the "what/how/why" questions: *What changes have been made, and how? Why was software changed in this way?* We should be able to identify various software changes, understand them and then learn from them.

There is a particular family of changes — *structural changes* — that has received relatively little attention in software evolution research, although they are activities commonly applied to improve software design and support future evolution. Structural changes, including function/file *moves*, *renames* and *merges/splits*, are applied by maintainers during refactoring or restructuring, to reduce the complexity of software and improve the cohesion, so that the software becomes more flexible and easy to maintain in the future. Although the raw effects of such changes may be plainly evident in the new artifacts, the original "intent" of design changes is often lost, such as we might not know that the removal of code from a file was actually caused by a specific functionality being extracted out to form a new function. Recovery of structural changes can help developers to understand design changes made in the past.

The ground work for the research described in this thesis on studying structural change was performed by Tu and Godfrey [20, 36]. They investigated a set of techniques, called *origin analysis*, to detect structural changes. In their approach, they derived origin information of software entities by comparing their metric fingerprints and call relationships. To support origin analysis, as well as other tasks in software evolution, they built a tool called *Beagle*. Their case study of GCC showed that this approach could help us in identification of function/file moves and renames.

However, this approach as implemented in its initial work had limitations: it has limited interaction with users, thus reasoning about structural changes was hard to perform; it was not flexible enough to deal with various situations in which structural changes may occur; only simple changes, namely *move* and *rename*, were addressed; and no support was provided for building a persistent evolution history of software changes.

Also identification of structural change instances is still not enough. We need to understand why changes occurred; we need to investigate whether there exist patterns and

anti-patterns for software changes, so that we can apply them or avoid them in future maintenance activities. We also want to capture intent behind changes.

1.2 Statement of the Problem

The statement of problem of this thesis is:

We can use improved "origin analysis", visualization and categorization to achieve a better understanding of software changes.

Given two versions of a software system, "origin analysis" is to find matching portions of source code, where matching is defined as one was derived from the other. Compared to its previous work [20, 36], we have made several improvements, including defining a generalized matching model and adopting a semi-automatic and iterative process. All these enhancements enable detection of function/file merges and splits, which was not addressed before. Also, structural changes identification using improved origin analysis has increased flexibility and accuracy, as human knowledge can be incorporated in the analysis process.

We adopt a technique called "scatter plot" to visualize structural changes between different releases. Our study shows that this visual aid not only helps to improve our understanding of structural changes identified, but also helps to discover new change instances and patterns.

Our case study of PostgreSQL shows that using our approach we were able to identify a large number of structural changes. Based on these change instances, our further analysis using categorization technique helps to recover even more knowledge about software evolution.

1.3 Organization of Thesis

The remainder of this thesis is organized as follows: In Chapter 2 we describe related work, including software evolution and previous work of origin analysis. As relationships exist between clone detection and origin analysis, we make a brief comparison between the two in this chapter as well. In Chapter 3, we present the improved version of origin

analysis and its tool support Beagle. Chapter 4 describes our case study of PostgreSQL, including details of structural changes, summaries, and results obtained after applying change analysis. Finally in Chapter 5, we summarize this thesis and discuss future work.

Chapter 2

Related Work

In this chapter, we describe previous research on software evolution, including empirical studies on industrial software systems and open source software, as well as various techniques used to visualize evolution history. We also discuss previous work of identifying structural changes using *origin analysis* and the tool Beagle, which serves as the starting point for our work. As origin analysis is related to clone detection and finding reusable components in software reuse, we briefly discuss these research areas as well.

2.1 Software Evolution

It is well accepted that successful software has to be changed over time. Although it would be ideal to be able to build a system that meets all the requirements for the future and requires little maintenance effort during its lifetime, in reality, it is rarely, if ever possible. We always build assumptions, both implicit and explicit, into a software system. As time goes by, these assumptions may be violated, as the running environment of software — including requirements, technical environment, development team and other factors — change continuously [34]. Thus, we have to change the software itself to adapt to these changes. The phenomenon of *software aging* is inevitable [33]; that is, the deterioration of software quality and flexibility cannot be avoided; software becomes harder and harder to be changed, no matter how excellent its initial architecture and design might be.

The research discipline of *software evolution* investigates various techniques, both tech-

nical and managerial, to ensure that software continues to meet organizational and business objectives in a cost effective way [1]. Although there is no generally accepted formal definition of software evolution, researchers have already agreed on the importance of studying software system in its long term and have been investigating this both empirically and theoretically.

Extensive empirical studies have proven to be effective in revealing the nature of software evolution. They differ in the way that evolution is characterized: some study growth of code base size, some investigate patterns of system functionalities. These studies may also differ in other ways, such as techniques applied. Various visualization techniques have been used in empirical studies. They have demonstrated great potential for detecting patterns in software evolution.

In the remainder of this section, we first describe the major contributions of empirical studies in software evolution in Section 2.1.1. Then, we review research on the evolution of open source software in Section 2.1.2. In Section 2.1.3, we discuss visualization techniques that are applied to software evolution. Finally, we summarize software evolution studies in Section 2.1.4.

2.1.1 Empirical Study of Software Evolution

In empirical studies of software evolution, researchers make observations on software systems in the real world to examine how software evolves throughout its lifetime.

Lehman did his pioneering empirical study of software evolution on 20 releases of IBM's OS360 [31]. In this study, he formulated five laws of software evolution based on his observations and experiences. These laws, being amended and validated against more software systems, were later increased to eight [29] — they are *continuing change*, *increasing complexity*, *self regulation*, *conservation of organizational stability*, *conservation of familiarity*, *declining growth*, *declining quality* and *feedback system*. The meaning of each law and the year it was developed is described in Table 2.1. Although some of the laws, such as *continuing change*, may seem obvious, it is surprising to observe that a large amount of software systems from different domains appear to obey the same set of laws. Noticing that the practical meaning of the laws to software engineering may not be well understood, Lehman further illustrated over 50 rules for applications in software system process plan-

ning, management and implementation, as well as suggestions for supporting tools to be built [30].

Year	Brief Name	Law
1974	Continuing Change	E-type systems must be continually adapted, else they become progressively less satisfactory.
1974	Increasing Complexity	As an E-type system evolves, its complexity increases unless work is done to maintain or reduce it.
1974	Self Regulation	E-type system evolution process is self regulating with distribution of product and process measures close to normal.
1980	Conservation of Organizational Stability	The average effective global activity rate in an evolving E-type system is invariant over product lifetime.
1980	Conservation of Familiarity	As an E-type evolves, all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behavior to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.
1980	Continuing Growth	The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime.
1996	Declining Quality	The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.
1996	Feedback System	E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

Table 2.1: Laws of software evolution

Many studies of software evolution were surveyed by Kemerer and Slaughter in 1999 [25]. They examined empirical studies performed by seven groups of researchers and observed that techniques used by them were limited, such as the sample size was limited and only single variable OLS (ordinary least squares) regression model was applied in pattern/trend analysis. They noted that these techniques were insufficient to meet the

longitude nature of empirical study of software evolution, and proposed a set of new techniques and methods. Some of these methods were borrowed from other research areas, such as sequence analysis and phase mapping. They also presented the design of their new approach that was based on large amounts of raw data, reliable procedures, and some new techniques. They asserted that their new approach could be able to provide more insights into the evolution process.

There were some other notable studies not mentioned in Kemerer and Slaughter's survey. We now summarize them as well as some empirical studies done since 1999.

Gall *et al.* examined a release history of 20 versions of a telecommunication switching system [16]. They measured system size using number of modules instead of LOC and investigated both the growth of the entire system and different subsystems. They discovered that the evolutionary behavior of the whole system differs from that of different subsystems — although the whole system seems stable, individual subsystems may exhibit high rate of change over their lifespans.

Burd and Munro proposed two metrics based on calling structure to assess change of maintainability when software evolves [14]. They assumed that maintainability was related to comprehensibility, and comprehensibility was closely related to the calling structure. The two metrics they used to measure maintainability were based on a count of the addition and deletion of functions, and the change of dominance tree relations in the call graph. From the case study of 30 versions of GCC, they found that both metrics seem to indicate the same change of maintainability. After they verified these results by examining change logs and interviewing maintainers, they concluded that the two metrics seem to be able to offer some important insights into the comprehensibility and maintainability of software.

Antón and Potts investigated how system functionality grows and evolves by analyzing the evolution of services in a domestic telephony system over 50 years [10]. They classified services into categories and analyzed change of *benefits* and *burdens* of each category. They found that functional evolution was punctuated over gradual enhancement. When a large number of services are introduced, a small decline of the number of services and benefits will usually follow. These findings seem to conform to results from FEAST project: rapid functional evolution may lead to fission [2].

Barry and colleagues introduced software volatility as a measure of software evolution

[12]. Software volatility consists of *amplitude*, *periodicity*, and *dispersion*, which measure the size of change, frequency of change, and variance of change respectively. In this approach, data of software volatility were further categorized into eight classes, and a volatility vector containing these volatility classes at different times was used to describe the evolution of a system. Based on the case study of 23 software systems, four groups of volatility patterns were discovered. For example, the two systems shown in Figure 2.1 and Figure 2.2 are in group 4, which can be characterized by constant large modification with wide variance in the beginning and decreasing change size and frequency thereafter. They also found that these two systems have same volatility patterns, as shown in the two figures, although they have different life cycle volatilities.

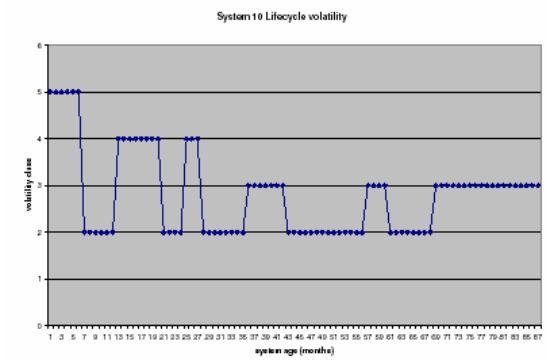


Figure 2.1: System 10 life cycle volatility

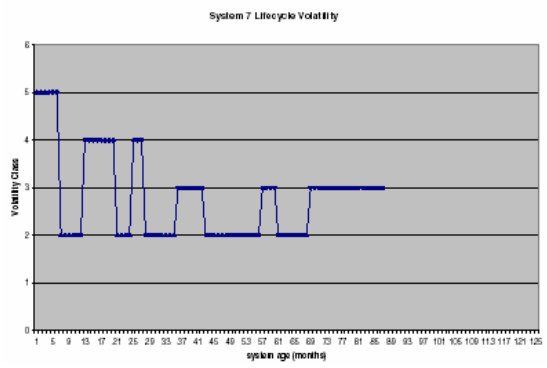


Figure 2.2: System 7 life cycle volatility

While many researchers are making observations on how software has evolved in the past, some researchers have tried to build predictive models of software evolution. Ramil and Lehman proposed six models to predict cost based on past evolution histories [35]. These models were built upon their previous observation in FEAST project [2]: the evolution of E-type software (software whose functionality evolves within an environment) has regularities, patterns, and invariants. In all six predictive models, the estimated effort required to evolve a system from time t to $t+1$ is a function of activity from time t to $t+1$. What makes a prediction model different is its measurement of activity from time t to $t+1$. They tested all the models on a data set covering 17 years evolution of a mainframe operating system kernel, and found that all the models predicted lower efforts than two

selected base models. The best model was based on coarse granularity. The phenomenon that measures with coarse granularity demonstrate better quality than measure with fine granularity was observed before, such as in [2, 28], where the number of modules seemed to be a better indicator than LOC.

We list these works in Table 2.2 in chronological order.

Author	Major contribution	Data	Evolution Path	What to measure
Gall 1997 [16]	Observe difference growth behaviors for the whole system and subsystems	Telecommunication switching system, 20 releases over 21 months	Time series analysis of growth at different architectural granularities	Added, removed or changed modules
Burd 1999 [14]	Propose two metrics measuring code maintainability	30 versions of GCC	History of proposed metrics	Two metrics derived from deletion and addition of functions, and changes of dominance relations
Ramil 2000 [35]	Predict cost as a function of a suite of metrics	ICL VME kernel	Regression	Number of modules/subsystems created, changed or handled
Antón 2001 [10]	Analyze the evolution of user-visible features	Telephony service over 50 years	Time series analysis	Change of services, benefit& burden
Barry 2003 [12]	Discover evolutionary patterns existing in real software systems	23 software systems	Phasic analysis	Software volatility metrics

Table 2.2: Empirical study of software evolution

2.1.2 Open Source Software Evolution

There is growing attention being paid to open source software (OSS) development by both industry and the research community. Features of OSS, like *free redistribution* and *access to source code and derived works* [6], make rapid evolutionary selection possible in OSS development. A large number of successful OSS are now being widely used, such as the GNU/Linux kernel, the GCC compiler suite, the PostgreSQL RDBMS and the Vim text editor.

Godfrey and Tu examined the growth of Linux kernel over a six year period using several metrics [19]. They found that the development releases of Linux kernel grew at a super-linear rate, which contradicts Lehman and Turski's inverse square growth rate hypothesis [28, 37]. Similar as Gall did in [16], they examined the growth patterns of different subsystems. The results showed that the rapid growth of `driver` subsystems contributed most to the growth of whole system. They also suggested that the *nature* of the subsystems and *evolutionary patterns* need to be investigated to gain a better understanding of how and why the whole system has evolved.

In [32], Nakakoji and his colleagues studied not only the evolution of OSS systems, but also the evolution of associated OSS communities. In their case studies of four OSS projects, they found that different collaboration models within OSS community exist, and that the difference in collaboration model results in different evolution patterns of OSS systems and communities. Based on their findings, they proposed a classification of OSS projects into three types: *Exploration-Oriented* (such as GNU software and Linux kernel), *Utility-Oriented* (such as Linux excluding Linux kernel), and *Service-Oriented* (such as PostgreSQL). The classification was based on following items: project objective, control style, system evolution pattern and community structure.

Ye and Kishida extended the work in [32] towards a study of the motivation of OSS developers in [38]. They argued that *learning is one of the major motivational forces* that attract software developers and users to participate in OSS development and to become member of OSS communities. This argument was based on learning theory: *Legitimate Peripheral Participation* (LPP), which essentially says that learning takes place when members of a community interact with each other in their daily practice. Noting that learning is one of the major driving forces in OSS development, they pointed out several practical

implications for this finding. For example, peripheral participation should be encouraged and allowed by different means to motivate developers and users; also integration of education and research with OSS might be a possible way to educate and train new software professionals.

2.1.3 Visualization of Evolution History

Visualization of how software changed in the past can aid in improving our understanding of software evolution. It helps maintainers to develop a mental picture of the software history. It may also lead to discoveries of patterns and hidden rules that are embedded in the large amount of data produced during software evolution.

Holt and Pak presented a tool GASE [21] to visualize software structural changes. For a pair of versions, red, grey, and blue colors were used to display new changes, common parts between two versions, and old parts respectively. By comparing the two graphs visualizing the system structure in the old version and new version, as shown in Figure 2.3 and Figure 2.4 (colors were not shown in the two figures), significant restructuring could be observed.

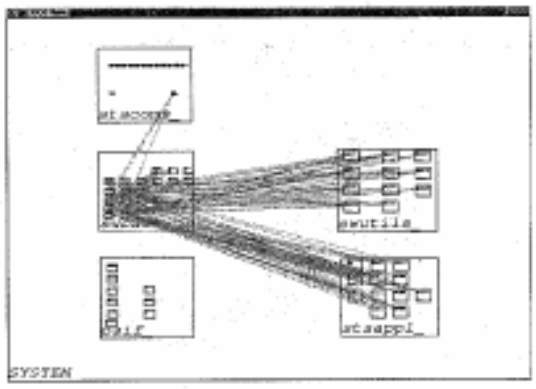


Figure 2.3: Old dependencies

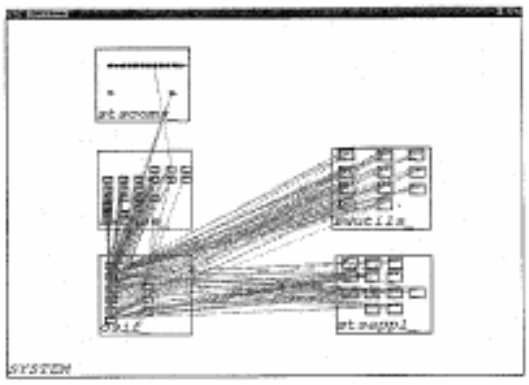


Figure 2.4: New dependencies

Gall *et al.* applied color and 3D in visualizing software release histories [17]. They displayed system structure as a 2D or 3D graphs, and used time as the third dimension of software release history. Different colors were used to represent different values of a

property, such as release sequence number (RSN). When the data of system structure was large, a view called "percentage bar" was used to offer a compact representation. For a software release history, multiple views could be produced. For example, Figure 2.5 shows system structures at different time, where a hierarchical tree was used to display the system structure in each release, and color of each program element in the trees represents the version it was last changed. Figure 2.6 shows a compact view of the previous figure, where a horizontal bar is a summary of how old entities are in a release: different ratios of different colors in a bar represent how many entities remain unchanged ever since which old version.

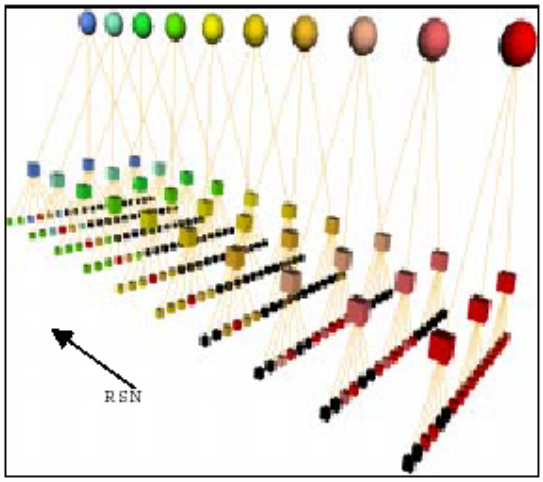


Figure 2.5: Visualizing release history

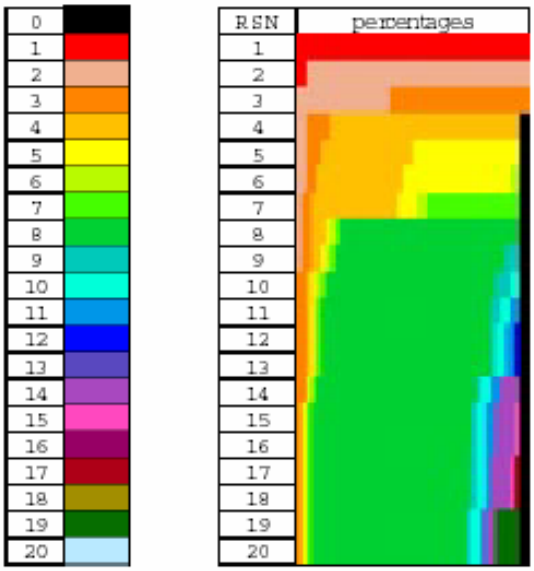


Figure 2.6: History with percentages

Lanza combined software metrics in visualization of software evolution [27]. In the *evolution matrix* that displays the evolution of classes in a software system, two-dimensional boxes are used to represent classes. The width and height of box represent values of any two metrics, such as *number of methods (NOM)* and *number of instance variables (NIV)*. Classes in different releases were displayed at the same time. For example, Figure 2.7 shows the evolution history of an example software system. With the help of evolution matrix, classes with evolution patterns, such as *pulsar* (class that grows and shrinks repeatedly),

and *supernova* (class that suddenly explodes in size), could be identified.

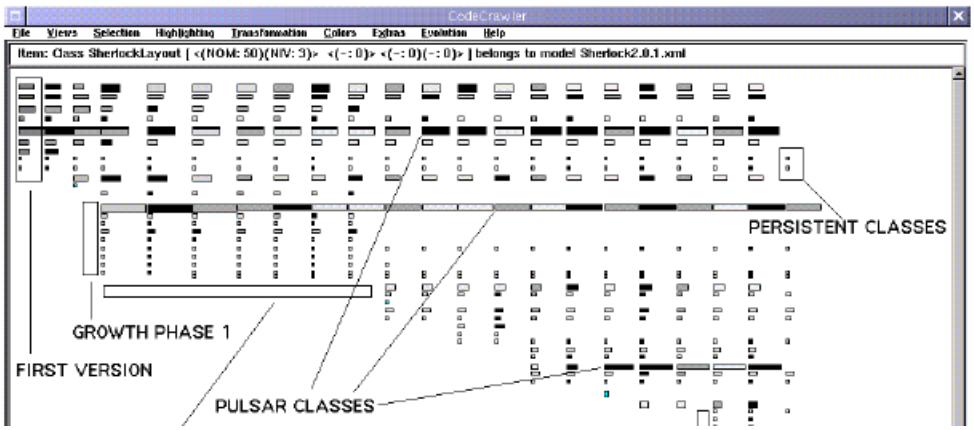


Figure 2.7: The evolution matrix of Sherlock

Cain and McCringdle investigated the combination of spatial and temporal visualization in [15]. In this approach, software evolution process was displayed as a movie, where each picture in the movie was generated from code in configuration management repository at one particular time. As shown in Figure 2.8, a label represents a class, the size of the label indicates the number of references to the class, and the color indicates the number of classes that the class uses. By examining the change of size and/or color in the movie, different phenomena of software evolution could be observed. For example as shown in Figure 2.8, although it was initially hardly visible, class `cBlock` was growing in its importance, as the number of references to it (represented as the size of the "cBlock" label) was increasing.

2.1.4 Summary

There is a significant body of work in empirical studies of software evolution. Researchers have investigated software changes from various aspects for different types of software systems using different analysis techniques. With or without aid from visualization techniques, some patterns or rules have been discovered. Results from empirical studies can be summarized as:

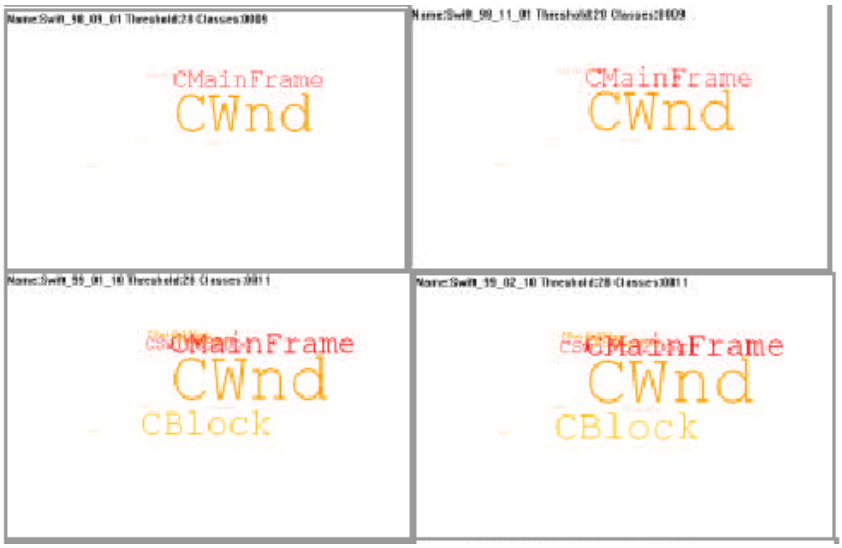


Figure 2.8: Four consecutive images at two-month intervals

- Different software systems evolve in different ways. However, some common patterns and "rules" may exist.
- It is useful to examine a software system from different perspectives to get a better understanding of it.
- Visualization is often useful in both understanding software evolution process and discovering interesting phenomena or patterns.

2.2 Previous Work on Origin Analysis and the Beagle Tool

Although extensive empirical studies of software evolution have been conducted by many researchers, few of them have focused on a better understanding of software changes themselves, the fundamental activities in software evolution. From the discussion in Section 2.1, we can see that most researchers have tried to use one or a set of metrics to summarize

software changes, such as lines of code (LOC) or software volatility. Although measuring a software system using a set of numbers does help to reveal some patterns of how software evolves, it often fails to provide answers to the "what/how/why" questions of software changes: *What changes have been made in the past? How? Why was software changed in this way?* As answers to these questions comprise a detailed story of how software evolves, they are essential for a better understanding of the *nature* of software evolution.

There are various kinds of software changes. Among them, a particular type of change — *structural change* — is important. Structural changes, including *moves*, *renames* and *merges/splits*, are often applied by maintainers during refactoring or restructuring, to improve the software design and support future evolution.

Structural changes are important because they reflect decisions of design changes made in the past. While the raw effects of such changes may be plainly evident in the new artifacts, the original intent of the design changes is often lost. Failure to identify and understand them will greatly affect our understanding of software changes. These decisions, once they are recovered, however, will become knowledge we can learn from, *e.g.*, they may include patterns that we can reuse in future maintenance.

Godfrey and Tu [20, 36] proposed an approach called *origin analysis* to track structural changes during software evolution process. Basically, they used two techniques:

- *Entity analysis* — a fingerprint is created for each function based on its various attributes, including S-complexity, D-complexity, Albrechts metric, Kafuras metric and cyclomatic complexity.
- *Relation analysis* — entities are considered for similarity based on the commonalities in various pre- and post-relational images, such as "are the two functions called by the same clients ?"

For each function G that appeared to be "added" in a new version, and for each function F that appeared to be "deleted" in an old version,

1. the entity analysis fingerprints were compared,
2. the *calls* and *called_by* relational images were compared, and

3. a simple string matching algorithm on the function prototypes was performed.

In their case study, they performed origin analysis on release change from GCC 2.7.2.3 to EGCS version 1.0. They found a large number of apparently new functions in EGCS 1.0 were actually old functions. For example, in `parser` subsystem, out of 848 apparently new functions, only 460 were truly "new" and the remainders were very similar to functions in older version.

2.3 Origin Analysis vs. Clone Detection

Origin analysis is related to clone detection [11, 23, 26, 13]: origin analysis borrows some techniques from clone detection, such as metric-based fingerprint of a function; and clones between two different versions may indicate instances of structural changes, such as file move and rename. However, there still exist significant differences between them.

2.3.1 Clone Detection

Clone detection is used to find duplicate or near-duplicate code introduced in maintenance activities. Code cloning occurs for a variety of reasons, such as under time constraints. Clones can increase the cost of maintenance, because errors embedded in the code are also copied at the same time when code is copied.

Different techniques exist for clone detection; four major groups are text-based, metrics-based, AST-based and token-based.

Johnson discussed sources of textual similarity and possible applications of clone detection in [23]. He investigated the clones in two versions of GCC using text-based matching and found a small part of the source files were clones in each version. He also demonstrated that clone detection could be used to find structural changes at the file level between different releases: if two files from two different versions are actually clones, then the file in new version may be just a moved or renamed version of the one in old version.

Baker presented a scalable approach to detect textually identical code in large software systems [11]. This approach also supported a systematic substitution applied to codes to increase flexibility, called *p-match*. By adopting an efficient algorithm based on a data

structure *p-suffix tree*, the overall running time could be roughly linear, as demonstrated in their case studies of two software systems. The case study also showed that in both systems around 20% of the code was cloned (of at least 30 lines).

In [26], Kontogiannis proposed a metrics-based technique for detecting duplicate functions. He used five metrics, including *Kafura*, *S-Complexity*, *D-Complexity*, *McCabe* and *Albrecht*, to together form a function signature. If the Euclidean distance of two function signatures is small, then the two functions are considered as clones. Results from case studies of four software systems suggested that this approach was fast and could be easily used. However, results were sensitive to the expected precision: high precision resulted low recall rate, which means only a small part of clones were identified; when recall was high, results had only low precision, which means many clones identified were actually not clones.

Baxter and colleagues applied abstract syntax tree-based clone detection in [13]. They applied a fast algorithm to detect subtree clones using a hashing technique, and then identified sequences of clones and near-misses clones. In the case study of a process-control system having 400K lines of code, they found that in the newer subsystems, percentages of cloned code were higher; clones in utility programs were rarely "cleaned up".

Toshihiro Kamiya and colleagues proposed a token-based clone detection system called CCFinder [24]. They transformed input source text and compared token sequences to extract code clones. They applied the CCFinder to various kinds of source codes in various languages. Their case study showed that this approach was efficient and scalable to large software systems. As another contribution, they used CCFinder to explore the differences or similarity of two or more systems.

2.3.2 Using Clone Detection to Identify Structural Changes

Johnson applied clone detection to identify structural changes between two versions in [23]. He applied exact substring matching to files from two versions, and then put files into one group if they were clones. These file groups were further clustered according to the number of files in the group as well as release information. There are two clusters that contain structural changes:

- *ABX* — containing one file from the *A* release and one file from the *B* release with different names.
- *Cplx* — containing three or more files.

In cluster *ABX*, as each file group contains two different files from two different releases with same content, it may indicate a file rename has occurred. In the case study of GCC compiler of version 2.3.3 and 2.5.8, a substantial number of file renames were found. These renames were caused by reorganization of the `configuration` subdirectory.

Considering file groups in cluster *Cplx*, if all the files were in one release, then they were just clones in one release; otherwise, they may represent instances of structural changes, such as file move or rename, depending on other information. In cases of structural changes, file names were used to guess the cause of changes.

In Johnson's approach of detecting structural changes, only files were analyzed; fine-grained software entities such as functions were not discussed; also, as text-based matching does not consider semantics or syntax, it is sensitive to various changes, which are usually unavoidable or even expected in the context of software evolution.

2.3.3 Origin Analysis vs. Clone Detection

From our above discussions, we can see that origin analysis is related to clone detection in some perspectives. However, origin analysis is different from clone detection in following aspects:

- Clone detection is usually applied to one version of software system, while origin analysis is applied to two versions.
- The purpose of clone detection is to find "similar code fragments", while origin analysis is to find "same software entities". The difference between "similar" and "same", and between "code fragments" and "software entities" result in situations where results from origin analysis are not equivalent to results from clone detection.
- Techniques in clone detection have to be cheap to be applicable. While in origin analysis, it may be still practical to adopt expensive techniques, as in many cases only a small set of entities are involved.

2.4 Finding Reusable Software Components

Origin analysis is also related to some techniques used to find reusable software components.

In software reuse, components stored in repository have to be found for specific reuse purposes. There are several techniques to find components that meet with certain requirements, such as browsing a hierarchical organization of components. One technique that is related to origin analysis is called *signature matching* [39], a library search technique based on function type and module interface.

In signature matching, two functions are matched only if their types are the same, or the same after some relaxations. Relaxation in a search was transformation performed on types in the search, either on types in the query or the types in the library. Typical predefined relaxations include rename and reorder *etc.*. As type information of a function reveals what kinds of data is manipulated and is relatively stable during release changes, signature matching can be a candidate technique adopted in origin analysis.

As we will see in Chapter 3 about improved origin analysis, we have a *declaration-matcher* that compares two functions by comparing their formal parameter names. As a function declaration is composed of both formal parameter name information and type information, and we only consider the first one in our current implementation, we think incorporating signature matching into origin analysis, as additional consideration to the type information in the function declaration, will be an improvement to our approach in the future.

2.5 Summary

In this chapter, we have reviewed related work to our research, including software evolution, previous work on detecting structural changes using "origin analysis", and finding reusable software components. In the next chapter, we will present our improvements to origin analysis, as well as improved version of its tool support Beagle.

Chapter 3

Improving Origin Analysis

From our discussion in Chapter 2, we can see that various aspects of software changes have been investigated empirically to reveal patterns or rules in the evolution process. These patterns and rules help us to gain a better understanding of how software evolves in its lifetime.

Origin analysis and its tool support Beagle, as proposed in the initial work [20, 36], was useful in identifying structural changes. In origin analysis, basically two major techniques — entity analysis and relation analysis — were applied on those software entities that appeared to be inserted into new version or deleted from old version. Simple structural change, such as function moving and renaming, could be detected. However, there are improvements need to be made: the analysis routines were performed all at once in a batch with no interaction with end users; there was only limited support for reasoning about structural changes; only two types of changes, move and remove, were discussed, and complex changes, such as merge and split, were not addressed; all the results would get lost if Beagle shuts down, thus no support was provided for building a persistent model of evolution history.

In this chapter, we describe our improvements to origin analysis as well as the Beagle support tool. The remainder of this chapter is organized as follows: Section 3.1 gives the definition of origin analysis. Section 3.2 describes the major improvements to origin analysis. Section 3.3 presents three basic steps in origin analysis, with their details discussed from Section 3.4 to 3.6. Section 3.7 discusses features in Beagle. Finally, Section 3.8

describes how to detect structural changes of different types using our approach.

3.1 Definition of Origin Analysis

Origin analysis was defined as follows:

> Suppose G is a software entity (such as a function, class, or file) that occurs in a particular version of a software system, call it V_{new}. Suppose further that G did not "exist" in the previous version, call it V_{orig}, in the sense that there was no like entity of the same name and/or location.
>
> *Origin analysis* is the process of deciding if G is a program entity that was newly introduced in V_{new}, or if it should more accurately be viewed as a renamed, moved, or otherwise changed version of an entity from V_{orig}, say F.

While this informal definition helps to show the intuition behind our research, *origin analysis* as we have implemented and nvestigated it in this thesis is slightly more complex:

- Origin analysis can be performed in either direction: old-to-new, or new-to-old. That is, the above formulation essentially asks the question: *"Are these apparently new entities really new?"*; one might be just as interested in asking: *"Are these apparently deleted entities really gone from the new version?"*. The original implementation of origin analysis in the Beagle tool considered only the first question, but we now support looking in both directions.

- Since merging and splitting of software entities may occur, there may be several G_is that were split from a single F, and there may be several F_is that were merged into a single G. It may also be the case that several F_is are merged into a G that is present in both versions of the system (or analogously, an F that exists in both versions may split off some of its "old" functionality into one or more "new" G_is into the new version).

3.2 Improved Origin Analysis

We note that while simple renaming and moving of entities are easy to define formally and fairly easy to detect, the more general concept that *G is a changed version of F* is not. A structural change can be as simple as moving a software entity to a new location, or as complex as merging/splitting. Moreover, a single entity could be involved in several structural changes between two releases of a system, or closely related entities may change as a group all at once. Cases of structural changes can be so complicated that they are hard to detect even by manual examination. Thus how to identify structural changes of various forms remains as a difficult problem.

We believe origin analysis must be a semi-automatic approach to be useful; that is, a user must apply experience and common sense to decide whether an entity is a changed version of another. At the same time, computation must be cheap to achieve effective human-computer interaction. Also, origin analysis must have a flexible structure, so that the user can apply different technologies for various situations and reason about changes. Extensibility is greatly appreciated, as new techniques may need to be added in the future.

We consider all the above requirements in our improved version of origin analysis. Essentially, we have built a generalized model of matching and adopted an iterative and interactive process. The generalized matching model characterizes software entities and matching. It enables different matching techniques to be "plugged-in" or "unplugged" when necessary. Performing origin analysis becomes semi-automatic and iterative. The user has flexibility in choosing matching techniques. (S)he can also apply knowledge in deciding real matches.

We have redesigned Beagle to support the improved version of origin analysis; the new Beagle tool is a complete re-implementation.

3.2.1 Generalized Matching Model

"Matching" in a central concept in improved origin analysis. Generally speaking, it refers to a process to find match candidates using a certain strategy, where match candidates are pairs of software entities that one of them is a possible "origin" of the other. We have generalized this concept into an abstract interface called `Matcher` and used "matcher" to

refer to any object that implements this interface. Details of the interface will be given in Section 3.5.1. As we make no assumptions for what is inside a matcher, matchers can have different ways to produce match candidates, *e.g.*, based on different definitions of how similar two entities are. They can also coexist and be plugged in or unplugged easily. A matcher may even use other matchers to provide its own matching service.

The generalized matching model reflects how we model structural change identification. It involves other related concepts, including similarity measures and entity attributes. We now examine this model in more details:

- *Software entities are characterized using attributes.*

  We use *attributes* to represent those properties we choose to characterize software entities. Different kinds of entities may have different attribute set. For example, functions have the attributes of name, declaration, implementation metrics and call relations, while files have attributes name and metrics. Common types of attributes may exist for different types of entities; *e.g.*, name is a common attribute for functions and files.

- *Different similarity measures can be defined based on entity attributes.*

  A similarity measure is used to indicate how closely related two entities are from a certain perspective. For example, we can use the longest common subsequence of two function names to define the similarity of functions from the perspective of their names. Different similarities can be defined, such as on different sets of attributes. Even for the same set of attributes, different similarity criteria may be used.

- *Different matchers may use different similarity measures to implement their matching strategies.*

  When matchers are based on different similarity measures, they are able to measure how similar two entities are from different perspectives. Thus even in case that a software entity changes in many aspects, we may still be able to track its origin based on other aspects that remain roughly similar. For example, suppose we have a matcher that is based on similarity of function declarations. If a function changes its name as well as its implementation completely in a new version, but with no changes

to its declaration, then we may be still able to detect this instance of *move&rename* using the matcher that compares function declarations.

We can see the big advantage of adopting the generalized matching model in origin analysis in flexibility. Now we are able to apply different strategies for different situations of structural changes. We can also easily create a new strategy or remove an old one without breaking the uniform structure of origin analysis.

3.2.2 Interactive and Iterative Process of Origin Analysis

The generalized matching model incorporates various techniques for origin analysis into one unified framework. However, without an appropriate process of applying origin analysis, it is still not enough for identification of structural changes in an effective and efficient way. We adopt an interactive and iterative process for origin analysis to achieve flexibility and accuracy in change identification.

The basic process of the improved origin analysis is a semi-automatic one: looking for match candidates and then deciding which ones are real. User interaction is an integrated part of it, as the user must choose matching strategies, make decisions over match candidates, and reason about changes. Applying origin analysis is also iterative, with different strategies applied on possibly different entities in different iterations. The iterative and interactive process helps to improve the efficiency of origin analysis, which we will explain in following paragraphs. Details of how to perform origin analysis will be described from Section 3.3 to Section 3.6.

Like a library searching system, we consider two measures of how good a search system is: *recall* and *precision*. Recall is a measure to the ability of a system to present all relevant items, and precision is a measure to the ability of a system to present only relevant items. In the context of origin analysis, recall represents the ability to capture all the real matches, and precision represents the ability to present only real matches for user decision. An ideal approach would be one that has high recall and high precision, which means that the searching result is a small list of candidates containing real matches.

However, achieving high recall with high precision at the same time is not easy. For a target entity, we may find multiple entities whose properties are similar to it. If we apply

a strict matching strategy (one that sets tough conditions for entities to be considered as candidates), such as setting a high threshold for similarity, then we may be able to get a small list of candidates, but we may also risk not retrieving the real match at all at the same time; if we apply a relaxed strategy, such as with a low threshold of similarity, to ensure that the real match is in the candidates, then we may get a large number of candidates, which imposes great effort for decision.

Using improved origin analysis, we are able to choose a matching strategy to be applied in one iteration; we can further choose the entities to participate. By reasoning about changes and by applying heuristics, we are able to reduce the problem caused by the dilemma between recall and precision:

- Suppose a relaxed matching strategy is being applied. Considering that in many cases the candidates can only come from particular entity sets — such as only apparently deleted entities from an old release and apparently inserted entities from a new release in case of function renames — we can set the candidates to be only these entities without losing the real ones. If we perform the matching strategy on the reduced entities, the size of the candidate set will decrease, and the decision will be easier to make.

- For a strict strategy, as a real match may not be obtained, matching strategy can be slackened in later iterations. At the same time, considering that a strict strategy may result few candidates for each entity even applied on a large number of entities, which means it is still easy to decide real matches, we can set the entities in the matching to be large to increase the number of possible real matches identified in this iteration. As matches identified in previous iterations may be excluded from matching in later iterations when relaxed strategies are used, the more matches identified in this iteration, less effort is required in later ones.

The combination of the improved model and process provides better support for dealing with a variety of structural changes. For example, complex structural changes can be detected by comparing entities from different perspectives using different matchers; chained changes that involve entities dependent on each other, can be identified in multi-

ple iterations. We will show examples of this phenomenon in the case study in the next chapter.

3.2.3 Improved Tool Support

We have redesigned the Beagle tool to support the new model and new process of origin analysis. We have adopted a plug-in architecture for different components supporting the generalized matching interface. We have redesigned the database to support a persistent model of software evolution, thus changes identified before are still available when Beagle stops and starts. We have built various sub-tools for different phases of origin analysis. We also incorporate visualization techniques in both structural change identification and understanding, for example, we use a scatter plot to visualize structural changes for entities in two versions. There are many other features in Beagle that provides additional support for origin analysis, such as transactions, and navigation between matched entities.

We will describe more details of Beagle in Section 3.7.

3.3 Performing Origin Analysis

Origin analysis is performed iteratively, with each iteration consisting of following three basic steps:

1. *Defining matching space: SET_{orig} and SET_{new}*

   The user defines an entity set from each of the two different versions to be involved in matching; match candidates will be chosen from and only from these entities.

2. *Matching*

   The user applies one or more matchers. Matchers are sub-tools that implement matching strategies and produce match candidates.

3. *Decision*

   The user decides if there are any real matches from match candidates. In this step, the user needs to decide whether evidence is strong enough to make a commitment.

(S)he may examine the output of Step 2 as well as other sources of information, such as source code.

Figure 3.1 shows the data flow between the three steps.

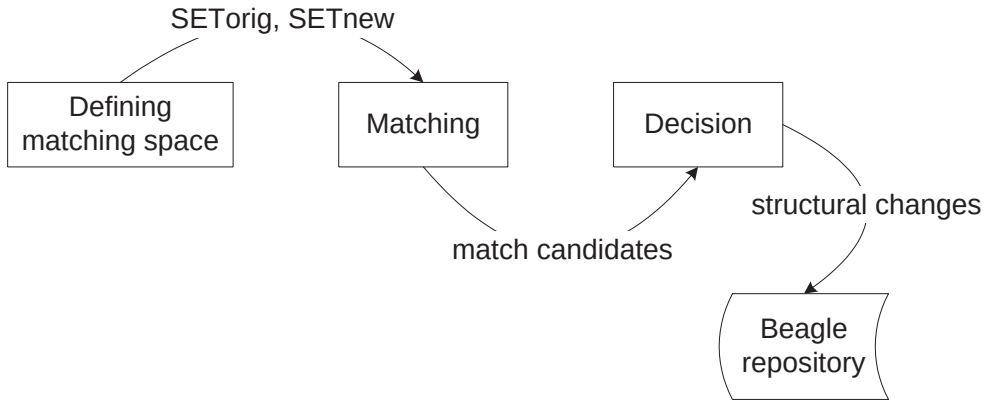


Figure 3.1: *Three basic steps of origin analysis*

The user can repeat the matching process for the same sets of entities until (s)he is satisfied that the "correct" origins have been found or, alternatively, that no such entity exist. In either case, once the decision has been made, it is recorded in the Beagle repository.

We will explain each step in details in following sections from 3.4 to 3.6.

3.4 Step 1: Defining Matching Space

We use SET_{orig} and SET_{new} to denote the two sets of entities, from V_{orig} and V_{new} respectively, to be considered in matching. As we have discussed before, we must be careful in setting up the two sets. We suggest some heuristics for this step.

- Consider whether the matching strategy to be applied in Step 2 is strict or relaxed. If we are going to choose a strict matching strategy, such as exact name matching, then we consider big matching space, since a strict matching strategy generally results a small candidate list with high precision even on a big set of candidates. If it is a relaxed one, consider other heuristics.

- Consider the characteristics of the structural changes to be identified, and restrict the matching space to only those entities that could be involved. For example, if we want to identify function renames, we can restrict SET_{orig} to be only entities apparently deleted from V_{orig} and SET_{new} to be only entities apparently inserted into V_{new}. If we further know (or suspect) that these renames must have happened in subsystem `access`, we can get an even smaller matching space by further restricting SET_{orig} and SET_{new} to be in subsystem `access`. This heuristic helps to reduce both computation time and decision effort without sacrificing precision. It can be applied to both relaxed and strict matching strategies.

- If possible, deliberately separate a big matching space into several small ones and match them one by one. For example, try matching on only one subsystem each time, considering that structural changes are generally performed in the same subsystem.

More heuristics may exist about how to define a good matching space. They can be discovered through experimentation and may be peculiar to the system under study.

3.5 Step 2: Matching

Matching is a process that produces candidate matches for a defined matching space. To make it flexible and extensible, we have generalized the notion of matching and built a set of matchers that implement different matching techniques.

In the remainder of this section, we first present how we generalize the notion of matching in subsection 3.5.1, including the `Matcher` interface and matcher plug-ins. Then we describe five matchers we have implemented.

3.5.1 Generalized Matching

We generalize the notion of matching as follows:

> Suppose we have two sets of entities from V_{orig} and V_{new}, called SET_{orig} and SET_{new} respectively, matching is simply a process of producing a list of match candidates in the form (target, candidate), where target is from V_{orig}, candidate

is from V_{new}, and candidate is an entity that is considered to be a possible match to target.

Matching can also be performed in the other direction $V_{new} \rightarrow V_{orig}$, that is to look for candidates from SET_{orig} for entities in SET_{new}.

For simplicity of explanation, we will discuss the direction $V_{orig} \rightarrow V_{new}$ only in the remainder of this section, which means targets are in SET_{orig} and candidates are in SET_{new}.

Note in our generalized matching, we put no restriction on the types of the entities — they can be functions or files. We make no assumption about techniques used in matching — they can be comparison based on two names, or comparison based on call relations, or even both. Although we have not studied other types of software entities, such as global variables, types and classes, incorporating them into this general model, is straightforward task.

We define `Matcher` to be the common interface for all matching service providers, called *matcher*s. The `Matcher` interface defines a set of essential operations that each matcher must support to provide its service. A matcher may have its own way to compute similarities of entities and produce match candidates, but it must be able to communicate with its clients via a common interface.

The `Matcher` interface, shown as Java code, defines two operations:

- `Object chooseOption()`
- `Vector doMatch(Vector` SET_{orig}`, Vector` SET_{new}`, Object option)`

Method `chooseOption` allows matcher to be configured, such as whether the matching should be exact or inexact, or the value of similarity threshold. Its implementation can vary from matchers; the one used by most matchers we have implemented is to pop up a dialog with configuration choices to be specified by the user. All the choices, once chosen, consolidate into the configuration object returned by this method.

Method `doMatch` asks a matcher to perform its matching process. The three parameters contain all the information needed for a matching: SET_{orig} and SET_{new} are entity sets to be searched for match candidates, and `option` specifies configuration of the matching. A matcher applies its own algorithm to compare entity attributes, compute similarities, and

select match candidates. The return value is a list of elements in the form of (F, G, r), where F is a target entity from SET_{orig}, G is a match candidate for F from SET_{new}, and r is an object that contains detailed information of matching F and G (generally including similarity measure and its computation details). Associating match details with each candidate makes it easier to use matching results; for example the user can rank candidates by similarities, or check how the similarity is computed when deciding real matches.

3.5.2 Implemented Matchers

We have implemented five matchers, including four single attribute matchers that compare only one attribute, and `expression-matcher` that allows comparison of multiple attributes at the same time. The four single attribute matchers are `name-matcher`, `declaration-matcher`, `metrics-matcher`, `relation-matcher`, considering attributes name, declaration, metrics and call relations respectively. The `name-matcher` can be applied on all types of entities, including functions, file and subsystems, while the other three can only be applied to functions. The fifth matcher, `expression-matcher`, is built upon the four single attribute matchers. It integrates services from other matchers and makes it possible to query candidate considering multiple attributes at the same. Figure 3.2 shows the relationship between the five matchers.

We note that there exists a viewer for each set of candidates produced by each matcher. Viewers separate displaying match candidates from accessing the data directly, thus provide additional flexibility for navigating candidates and reasoning about them. For example, different viewers may be attached with match candidates produced by different matchers; for the same candidate set, multiple displays are possible to provide multiple views. In our current implementation, we have two viewers, `simple-viewer` and `expr-viewer`. All the single attribute matchers use `simple-viewer`, and only `expression-matcher` uses `expr-viewer`.

Using `simple-viewer`, we can query all the distinct targets ranked by best candidate (the candidate having highest similarity with the target), and list all the candidates for a given target ranked by similarity values. Using the other viewer, `expr-viewer`, we can perform the similar ranked listing, but with additional constraints on the visible candi-

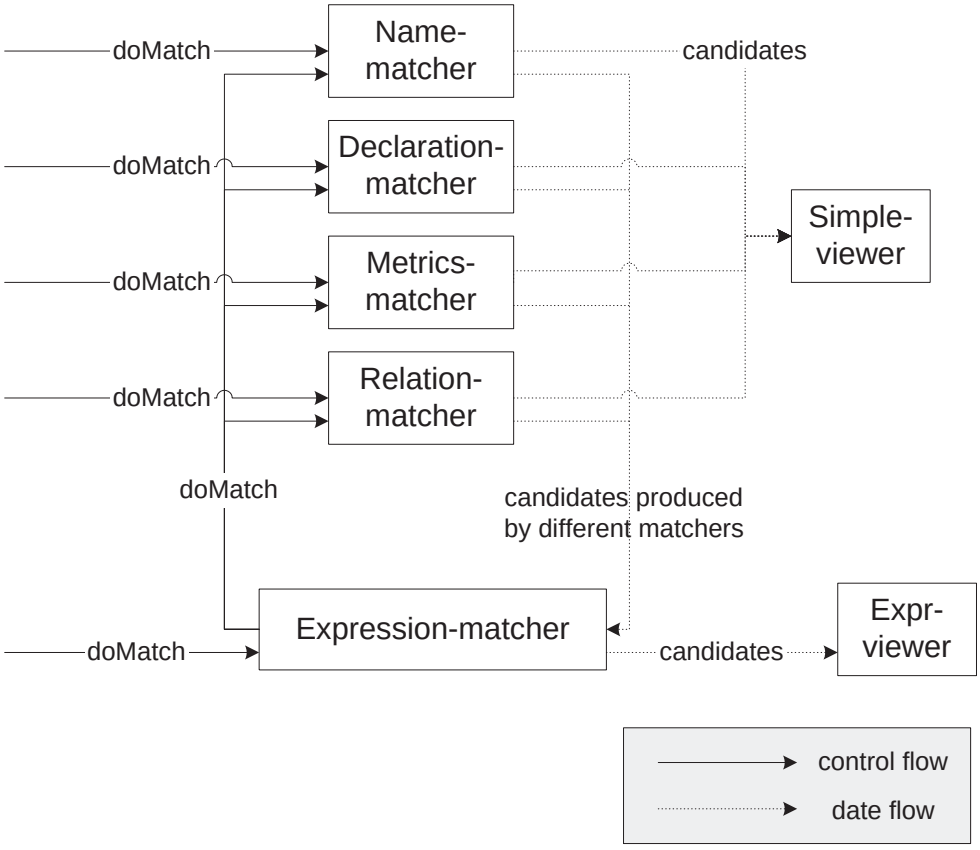


Figure 3.2: *Five matchers implemented*

dates. The constraints are in form of boolean expressions. For example, an expression like "name>0.5 & decl>0.5" will result that only match candidates, whose similarities computed by `name-matcher` and `declaration-matcher` are both greater than 0.5, are included. The `expr-viewer` requires the candidates to be organized in a 2D matrix data structure, which is only supported by `expression-matcher`. As `expr-viewer` is closely related to the 2D matrix data structure, which is also the central data structure of `expression-matcher`, we will discuss them all together in Section 3.5.2.

In additional to displaying candidates, viewers support manipulation of candidate set. For example, we can delete a target, which will result in all its candidates being deleted.

We can also delete arbitrary number of candidate of a target. These operations are useful, as we may need to eliminate uninteresting results and focus only on particular candidates.

Although the four single attribute matchers are different in the attributes they compare, they do share some commonality in their supporting options and matching algorithms. For example, they generally support both inexact and exact matching; in the case of exact matching, hashing technique is often used to reduce computation time. Also, they all support threshold of similarity to reduce output size. Finally, their matching results are all associated with `simple-viewer`.

In the remainder of this section, we will describe each matcher in more details. For each matcher, we will focus on following aspects: similarity computation, options supported, and running time analysis. When we discuss `expression-matcher` in Section 3.5.2, we will describe the `expr-viewer` as well.

Name-matcher

`Name-matcher` compares names of software entities, including functions, files and subsystems. Name is an important attribute for all the software entities, as it is generally a description of an entity's role or function. When the functionality of an entity changes, its name might change as well. An entity may also changes its name because a new naming convention is adopted. We will have more discussions in Section 4.5 about various cases of function renaming.

- *Similarity computation:*

  `Name-matcher` computes similarity of two names based on Longest Common Subsequence (LCS). The similarity of two names s_1, s_2 is defined as

  $$\frac{length(LCS(s_1, s_2)) \times 2}{length(s_1) + length(s_2)}$$

  where $LCS(s_1, s_2)$ is the longest common subsequence between s_1 and s_2, and $length(s)$ is the length of a string. For example, if two names are "PgDeleteAttribute" and "DeleteAttribute", then the LCS of the two is "DeleteAttribute", and the similarity is $15 \times 2 \div (17 + 15) = 0.9375$.

Other approaches are possible, of course, to measure similarity of function names. We can easily use a different measure, or even multiple measures in our approach.

- *Option:*

  `Name-matcher` provides four options:

  1. case sensitive or insensitive,
  2. exact or inexact,
  3. whether to apply a change schema (a rule of name change), and
  4. threshold of similarity.

  The first option is applied during LCS computation: if it is set to be case sensitive, comparison of characters when LCS is computed is case sensitive, otherwise it is case insensitive.

  For the second option, if it is set to be exact, hashing technique will be applied internally to reduce computation time to $O(n+m)$. When it is set to be inexact, all the $n \times m$ name pairs will be compared without optimization.

  The third option is used in case that a name change schema is known or suspected. A name change schema is a rule in the form "$A \rightarrow B$". Applying it on a string will result all the substring As in the string be replaced with Bs. The change schema, if there exists one, is applied on targets in SET_{orig} before they are compared with entity names in SET_{new}.

  The last option, threshold of similarity, controls the candidates to be returned: only candidates with similarity value bigger than or equals to the threshold are included in the return list.

  The four options can be used either individually or in combination.

- *Complexity:*

  The running time for computing LCS for two strings is $O(pq)$, where p, q are lengths of the two strings. If we reasonably assume that p and q is less than a const for names in a software system, then computing LCS requires const time.

If no hashing technique is applied, then the running time for the whole matching process is $O(mn)$, where m is the number of entities in SET_{orig} and n is the number of entities in SET_{new}. If hashing is used in case of exact matching, then the running time can be reduced to $O(m + n)$.

Declaration-matcher

`Declaration-matcher` compares two functions based on their declarations. Similar to name, a function declaration contains information about what a function does. Moreover, it exposes the function interface that contains details about how other functions can interact with this function. As the interface is exposed to others, its change may result changes to all the other functions that use the function. As a result to avoid such a change propagation, maintainers are often reluctant to change function declarations if it can still remain unchanged. The resulting stability of function declarations, makes it a valuable attribute in the detection of structural changes, especially for function renames.

- *Similarity computation:*

  In our current implementation of `declaration-matcher`, we compare function declarations by comparing names of formal parameters textually. We first concatenate all the parameter names in a function declaration in alphabetic order into a string, and then use the LCS to compute similarity. In the formation of the string, we choose alphabetic order instead of the sequence order to ensure the comparison is insensitive to parameter shuffling; we also choose formal parameter names rather than parameter types to be able to avoid noise that might be caused by renaming of data types. The disadvantage of not choosing the types is that we may have difficulties in matching declarations whose parameter names are changed while types remain unchanged.

  A possible improvement to our current method is to consider both the type and name of formal parameters. Options can be provided to choose whether to match type only or name only, or both type and name.

  The similarity of two function declarations is defined based on LCS of the two strings, s_1 and s_2, which are composed from formal parameter names:

$$\frac{length(LCS(s_1, s_2)) \times 2}{length(s_1) + length(s_2)}$$

where $LCS(s_1, s_2)$ is the longest common subsequence between s_1 and s_2, and $length(s)$ is the length of a string s.

For example, suppose a function has a declaration looks like `(int total, char* str)`, then the string formed by concatenating formal parameter names in alphabetic order is "str:total". If another function declaration is `(char** str, int total)`, then the similarity of the two is 1.

- *Option:*

  There are two options supported by `declaration-matcher`:

  1. exact or inexact, and
  2. threshold of similarity.

  If the user chooses exact matching, then only match candidates with similarity 1.0 are returned. Otherwise, only match candidates whose similarity value is greater than or equal to the threshold specified in the second option are returned. In exact matching, hashing technique is used internally to reduce computation time.

- *Complexity:*

  Similar to `name-matcher`, we reasonably assume that computing LCS requires const time. Then if no hashing technique is applied as in the inexact matching, the running time is $O(mn)$. If hashing technique is used in the case of exact, the running time is reduced to $O(m + n)$.

Metrics-matcher

`Metrics-matcher` compares two functions based on their metrics information. The basic rationale of this matcher is: if a function does not change much in a new version, then moving to another place or changing to a new name, its value of metrics should be still similar to that in old version.

- *Similarity computation:*

  We use five metrics A, B, C, D, and E to form a signature of a function. We omit names of these metrics here, as we found several different selections of the metrics set produce similar results.

  Similarity of two function signatures, $\{A_1, B_1, C_1, D_1, E_1\}$ and $\{A_2, B_2, C_2, D_2, E_2\}$, is define as:

  $$S_A + S_B + S_C + S_D + S_E$$

  where S_A, S_B, S_C, S_D and S_E are normalized similarity of each metric. For example, A represents LOC (lines of code), and the value of S_A is computed as follows:

  $$S_A = \begin{array}{ll} 0.2 & \text{if } |A_2 - A_1| < 5 \\ 0.1 & \text{if } 5 \leq |A_2 - A_1| < 10 \\ 0 & \text{if } 10 \leq |A_2 - A_1| \end{array}$$

  Normalization of other metrics may be different from A, as they measure a function from other perspectives. However, the basic idea — normalizing a value of difference to $0 \sim 0.2$ by dividing it into several ranges and treating them separately — is the same.

- *Option:*

  We have one option for `metrics-matcher`: whether it is an exact matching or not. If the matching is set to be exact, then only functions with exact same metric values are considered to be candidates. We use a hashing technique to speed up an exact matching process; hash code is an integer composed of values of all the five metrics in a function signature.

- *Complexity:*

  The running time for `metrics-matcher` with m entities in SET_{orig} and n elements in SET_{new} is $O(mn)$, as similarity for each pair of entities needs to be computed. In the case of exact matching when hashing technique is used, the running time is reduced to $O(m + n)$.

Relation-matcher

There are many types of relation among software entities that can be used in entity comparison, such as inheritance, global variable use and includes. Currently, we focus on only one type of relation — function call. We will consider other relation analysis in our future work.

`Relation-matcher` compares two functions by their call relations. When a function is renamed or moved, its call relations often remain similar. When a function merges or splits, its call relations may merge or split as well. Moreover, patterns may exist in the change of call relations. All these characteristics of call relation makes it an important attribute in origin analysis. We will discuss detection of merges/splits using detailed call relation analysis in Section 3.8.2. Here we only describe some basic information of `relation-matcher`.

We first give some notations we use in this section:

1. $caller(F)$: the set of functions that call F
2. $callee(F)$: the set of functions that F calls
3. $call(F)$: $caller(F) \cup callee(F)$
4. $\sharp s$: number of elements in set s
5. $\sharp match(f, g)$: the number of functions in set f that has matched functions in set g. A function G is a matched function of F iff that G is the origin of F

- *Similarity computation:*

  Similarity of call relations of two functions F, G is defined as follows:

$$\frac{(\sharp match(caller(F), caller(G)) + \sharp match(callee(F), callee(G))) \times 2}{\sharp call(F) + \sharp call(G)}$$

- *Option:*

  We have one option for call relation based similarity computing: whether the matching considers both callers and callees, or it considers just callers only or callees only.

If the option is set to both callers and callees, then the similarity is the one we just defined above. If the option is set to callers only, then only callers are matched and the similarity becomes:

$$\frac{\sharp match(caller(F), caller(G)) \times 2}{\sharp caller(F) + \sharp caller(G)}$$

Similarly, if the option is set to callees only, the similarity is defined as:

$$\frac{\sharp match(callee(F), callee(G)) \times 2}{\sharp callee(F) + \sharp callee(G)}$$

Comparing functions based on combined similarity of both caller and callee sets is a good technique for finding functions that have been moved or renamed, but it works less well for finding merges/splits of patterns (discussed in Section 3.8.2), as they require fine-grained analysis of call relations. Allowing the user to match on similarity of only caller or only callee sets provides the additional flexibility to get a more accurate ranking when merging or splitting is suspected to have occurred.

- *Complexity:*

  The running time for `relation-matcher` with m entities in SET_{orig} and n elements in SET_{new} is $O(mn)$. It is possible to improve this by hashing technique. We intend to do this in the future.

Expression-matcher

`Expression-matcher` is a matcher that helps to integrate results from other matchers. It allows the user to choose and name a set of matchers, then automatically invoke them. After it obtains results produced by these matchers, it stores them as attributes into a similarity matrix. By building such an integrated and attributed data model for match candidates, it provides flexible query support for candidates, together with `expr-viewer`.

We first look at the central data structure used by both `expression-matcher` and `expr-viewer`: the *similarity matrix*. Figure 3.3 shows its basic structure. Essentially, it is a 2D matrix with attributes associated with each node (a cell in the matrix) indicating

similarities. Its column entities are SET_{orig} and row entities are SET_{new}. Each $node(F, G)$ — the node whose column entity is F and row entity is G — has a set of attributes, which contains information about how different matchers match F with G. If a matcher named m consider F and G as a pair of match candidates, and it computes their similarity as r, then $node(F, G)$ will have an attribute with name m and value equal to r, we denote it as $node(F, G).m = r$. If the matcher m does not consider that F and G are a pair of match candidates, then there is no attribute named m in $node(F, G)$. If two matchers $m1$, $m2$ both think F and G are a pair of candidates, and they compute the similarity as $r1$, $r2$ respectively, then there will be two attributes $node(F, G).m1 = r1$ and $node(F, G).m2 = r2$, as shown in Figure 3.3 .

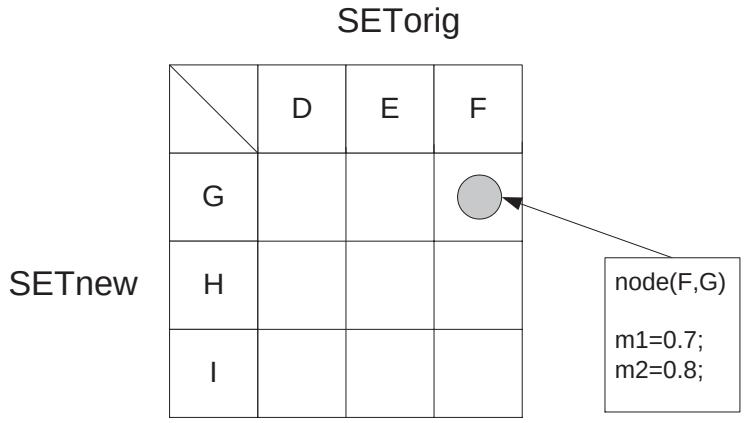


Figure 3.3: *Similarity matrix*

Now we look at how `expression-matcher` provides its enhanced query service for results from other matchers.

First, like other matchers, options for matching needs to be configured in `choose-Matcher()`. The user is allowed to choose several matchers and assign names for them as their unique identifiers. These names are the only variables that a query expression in `expr-viewer` can use. The user can even choose multiple instances for the same matcher if (s)he assigns different names for them. Thus, it is possible to choose two `relation-matchers` with names "rel-caller-only" and "rel-callee-only", one comparing callers only and the other comparing callees only. By doing this, we have additional

flexibility to compare entities. Each chosen matcher instance needs to be configured too; these configurations, together with chosen matcher instances and their names, become the configuration of `expression-matcher`.

During matching, according to the configuration information, the `expression- matcher` applies the chosen matchers one by one and processes candidates produced by each matcher in following way: for each matcher named m, for each match candidate (F, G, r) returned, it creates an attribute m with value r for $node(F, G)$, that is letting $node(F, G).m = r$. In this way, all the match candidates become attributes in the similarity matrix, ready to be queried.

`expr-viewer` is the query engine that works with `expression-matcher`. It takes the similarity matrix produced by `expression-matcher`, and supports querying of candidates using expression-like requests. The query expression must be boolean, and its variables must come from the names for matchers in the configuration of `expression-matcher`. For example, a boolean expression "`name>0.5 and decl>0.4`" is a correct one, if "`name`" and "`decl`" are names for two matchers, say `name-matcher` and `declaration-matcher`, chosen when `expression-matcher` is configured. Editing query expressions is supported by an expression editor as shown in Figure 3.4. In this editor, the user can choose expression variables from a list that contains all the available names (marked as *selected simple matchers* in the figure). (S)he can also choose operators and edit literals to compose an expression. Thus, specifying a boolean expression is an easy task.

Whenever a query expression is completed and submitted, `expr-viewer` will decide candidates to be returned based on the results of evaluating it on nodes in the similarity matrix. More specifically, if a $node(F, G)$ has attribute values that satisfy the boolean expression, then the pair of nodes(F, G) will be included in the candidate list. For instance, if the expression is "`name>0.5 and decl>0.4`", and $node(F, G)$ in the matrix has attributes $name = 0.6$ and $decl = 0.3$, then the expression is evaluated to *false* as the value of attribute "decl" is smaller than 0.4. As a result, the pair of nodes (F, G) would not be included in the query result.

The query facility provided by both `expression-matcher` and `expr-viewer` increases the flexibility for examining match candidates. We can easily construct strict and relaxed matching strategies by building different expressions. For example, we can build a

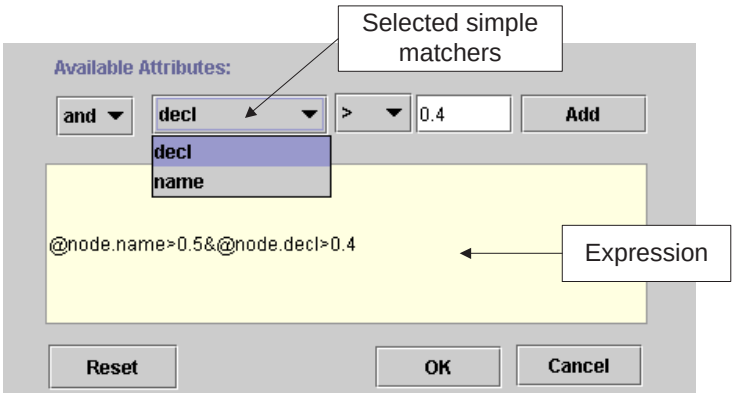


Figure 3.4: *Expression editor of expr-viewer*

strict matching strategy by writing an expression as "`decl> 0.6 and caller>0.9 and callee>0.8`"; we can also build a relaxed one by specifying the expression as "`decl>0.5`". As match candidates have alread been stored in the similarity matrix, there is no need to perform matching again and again when we try different matching strategies, thus greatly improving the matching efficiency.

3.6 Step 3: Making the Decision

The final step in the origin analysis process is to make a decision about the "origin"s of the functions and files being considered. After match candidates are produced in Step 2, the user needs to decide which is the correct "origin" of an entity by examining different sources of information:

- *Details of similarity computing produced by matcher*

  A matcher generally associates a short description of how similarity is computed with each pair of match candidates. For example, a description produced by `relation-matcher` matching two functions F and G might be "==@→elog, >>@→heapClose,...", where "==@→elog" means that both F and G called function `elog`, and ">>@→heap-Close" means that only F called `heapClose`. Compared with single-number similarity value, this description gives more details about how similar two entities are.

- *Source code*

  Source code is a good place to find information that helps to make a decision. Comments often indicate the functionality of a function or a file. Furthermore, in open source software, the quality of comments is often high, as many developers depend on them to understand and maintain the software system. Thus in origin analysis, we can use the comments to decide "origin" of software entities, as well as to understand the intention of design changes.

- *Documentation*

  Useful documentation for deciding matches includes release notes and history information kept in software configuration system, such as CVS log, *etc.*. These documents usually have records about how the original developers thought about changes, thus provide answers to "what has changed, and why".

We may use some heuristics in the decision phases, such as applying multiple matchers for cross-checking. Once decisions about matches are made, they are recorded in system model. In our case study, we made regular use of source code and CVS log in deciding real matches; this is detailed in the next Chapter.

3.7 The Beagle Tool

Beagle is a research tool that supports studying software evolution. It incorporates various techniques and sub-tools from software metrics, software visualization and relational database into an integrated platform. It allows users to query, visualize, and navigate through a system's history, and allows users to build persistent, annotated models of how structural changes have impacted the design of the system.

The initial version of Beagle [36] supported origin analysis in a simplified mode. In our improved version, we have redesigned the Beagle architecture to support the generalized matching model and the semi-automatic and iterative process of origin analysis. We have added many new features to provide strong support for reasoning, identification and understanding of software changes.

In the remainder of this section, we will describe Beagle in more detail. First in Section 3.7.1, we will show how the data in Beagle repository is collected and how it is managed. In Section 3.7.2, we describe the architecture of Beagle. Then from Section 3.7.3 to Section 3.7.5, we demonstrate support in Beagle for various tasks in software evolution study.

3.7.1 Collecting Facts for Beagle

Before we can perform any analysis task in Beagle, we need to collect and load facts into Beagle repository. As in the previous version, the major data input to Beagle are facts extracted from source code using reverse engineering techniques. We re-targeted Beagle to facts produced by SWAGKIT [8], as it produces facts conforming to the Datrix model [22]. The completeness of Datrix model ensures more detailed information about the software system is captured than was captured by the `cfx` extractor that was used by the previous Beagle implementation. Figure 3.5 shows the data flow of facts.

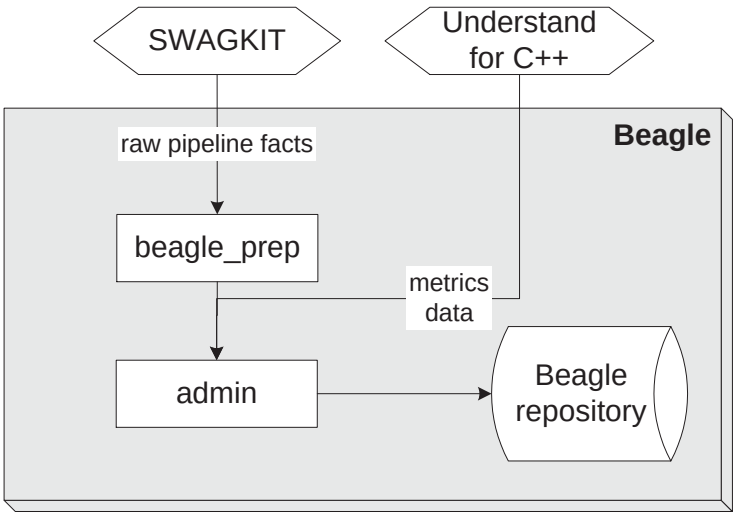


Figure 3.5: *Collect facts*

In the first step of the fact collection process, SWAGKIT produces facts of software architecture for each version of the software system, including:

- software entities at different granularities (*e.g.*, subsystems, files, functions and variables),
- entity attributes (*e.g.*, function name and line of definition),
- contain relations of entities (*e.g.*, subsystem S contains file F_1 and F_2), and
- many other relations (*e.g.*, variable use and function call).

Then `beagle_prep`, a set of small programs that manipulate these raw facts using relational calculator GROK [4], extract facts that are for Beagle usage only, including follows:

- *archInstance* — entities at different architecture levels, including subsystems, C files and functions defined in each C file,
- *ssContain* — contain relations among software entities,
- *cRefersTo* — function call relations, and
- *cDefFuncName* — formal parameter names of a function.

In parallel, metric data for functions and files are computed by `Understanding for C++` [9], forming a metrics report with predefined format for Beagle usage. The columns in the report are:

`Kind, Name, File, CountLineCode, CountLineComment, CountLineCodeDecl, CountLineCodeExe, Cyclomatic, CyclomaticModified, CyclomaticStrict, CountInput, CountOutput, MaxNesting`

The meaning of each column is explained in Understanding for C++ [9]. Here we only give a brief description of some important columns. Column "`Kind`" refers to the type of the entity. If its value is "Function", then it refers to a function. Column "`Name`" is the name of the entity. Column "`File`" is the file that the entity is defined. All the remaining columns are values of various metrics.

Once facts from both sources are ready, they can be loaded into the Beagle repository using an interactive Java program called *admin*. `admin` supports both facts loading and basic management of Beagle repository. Its major functions include:

- create/delete/edit a system in Beagle,
- add/update/remove a version in a system,
- load/remove facts for a version, and
- preview entities at different architectural levels in a selected version.

Figure 3.6 shows a snapshot of the GUI for `admin`. The tree on the left shows all the software systems that have been loaded in Beagle. The tree on the right displays entities in a selected version; the structure of the tree corresponds to the *ssContain* relation among software entities.

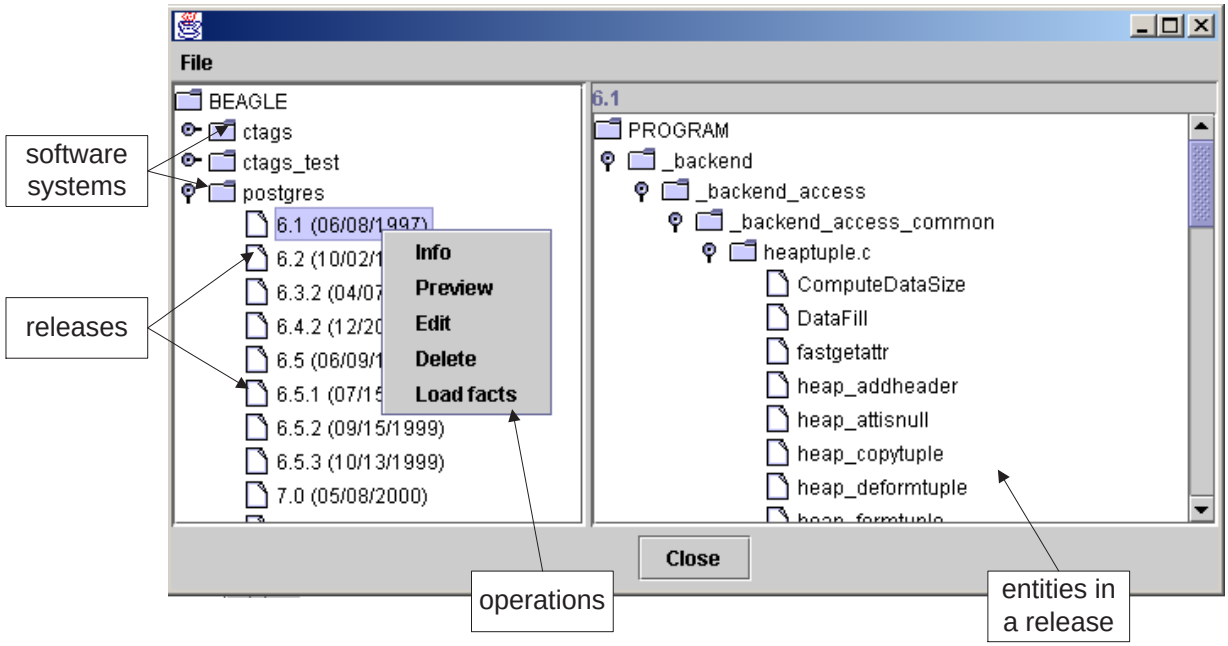


Figure 3.6: *Snapshot of admin*

Facts can be stored in any relational database supporting SQL, for example DB2 in our case. We choose to use a relational database as Beagle repository because it supports standard and efficient querying even for huge amount of data, which is especially valuable

in analyzing massive amount of facts in software evolution. The strong support of querying in RDBMS also matches well with the "exploratory" nature of studying software evolution. The database schema is based on previous version of Beagle, but we have performed many optimizations, so that most queries involving human-computer interactions can be finished within reasonable response time. We also add support for multiple software systems in the repository, so that it is possible to analyze several software systems at the same time. Moreover, we now support matches being saved in the database, thus matches would not get lost if Beagle restarts. With matches for consecutive version changes saved in repository, we are able to build an incremental and persistent model of evolution history.

3.7.2 Architecture

Beagle has a flexible architecture composed of matchers and other components. Figure 3.7 shows the architecture of Beagle. Matchers are plug-ins supporting different matching strategies. All of the matchers, as well as most of the other analysis and visualization components, use an internal data model, where entities are attributed objects. The internal data model is first loaded from the persistent data model in Beagle repository, but the two models are different: the internal data model contains intermediate information for components at running time only, such as uncommitted matches, which does not exist in the persistent data model. Components can use either data model. The `evolution history` is a component to be implemented in the future to visualize the evolution history in the persistent data model.

We have already discussed different matchers in details in Section 3.5.2. In the remainder of this section, we will focus more on other components in Beagle, including their main functionalities, and how they work together to support origin analysis.

Figure 3.8 gives a snapshot of some components in Beagle.

Ignoring matchers, components in Beagle can be grouped into three groups by their functionalities. Here, we only give a brief introduction of components in each group. Details about them will be discussed in the following three sections.

- *Basic information about software system*

  `Entity tree`, as shown in Figure 3.8, displays basic information of entities at differ-

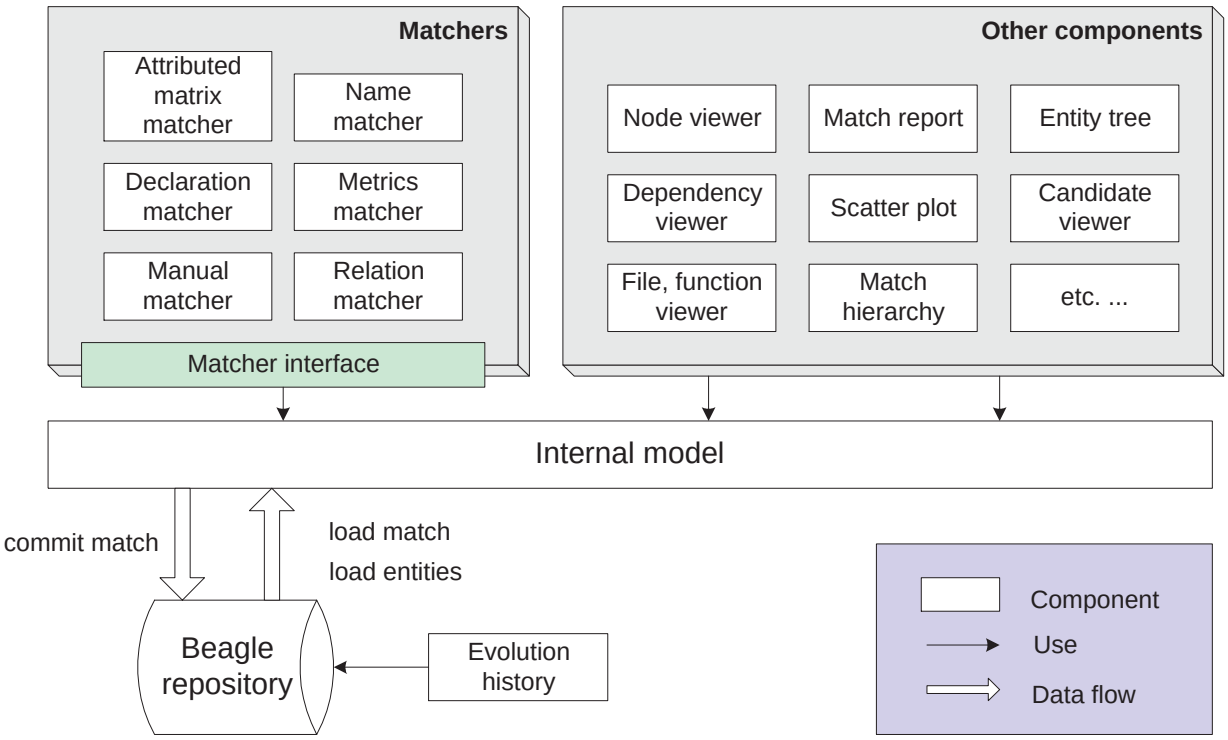


Figure 3.7: *Architecture of Beagle*

ent architectural levels for a given version. Details for a particular type of entities are presented by the particular viewer for them, such as `file viewer` for files and `function viewer` for functions.

- *Special support for performing origin analysis*

  Tool support is provided for each phase in origin analysis: `entity list` allows the user to choose entities to be involved in origin analysis; five built-in matchers implement different techniques for matching; `candidate viewer` displays match candidates and supports deciding on real matches. We also have transaction support that allows the user to commit or rollback matches.

- *Visualization and reporting*

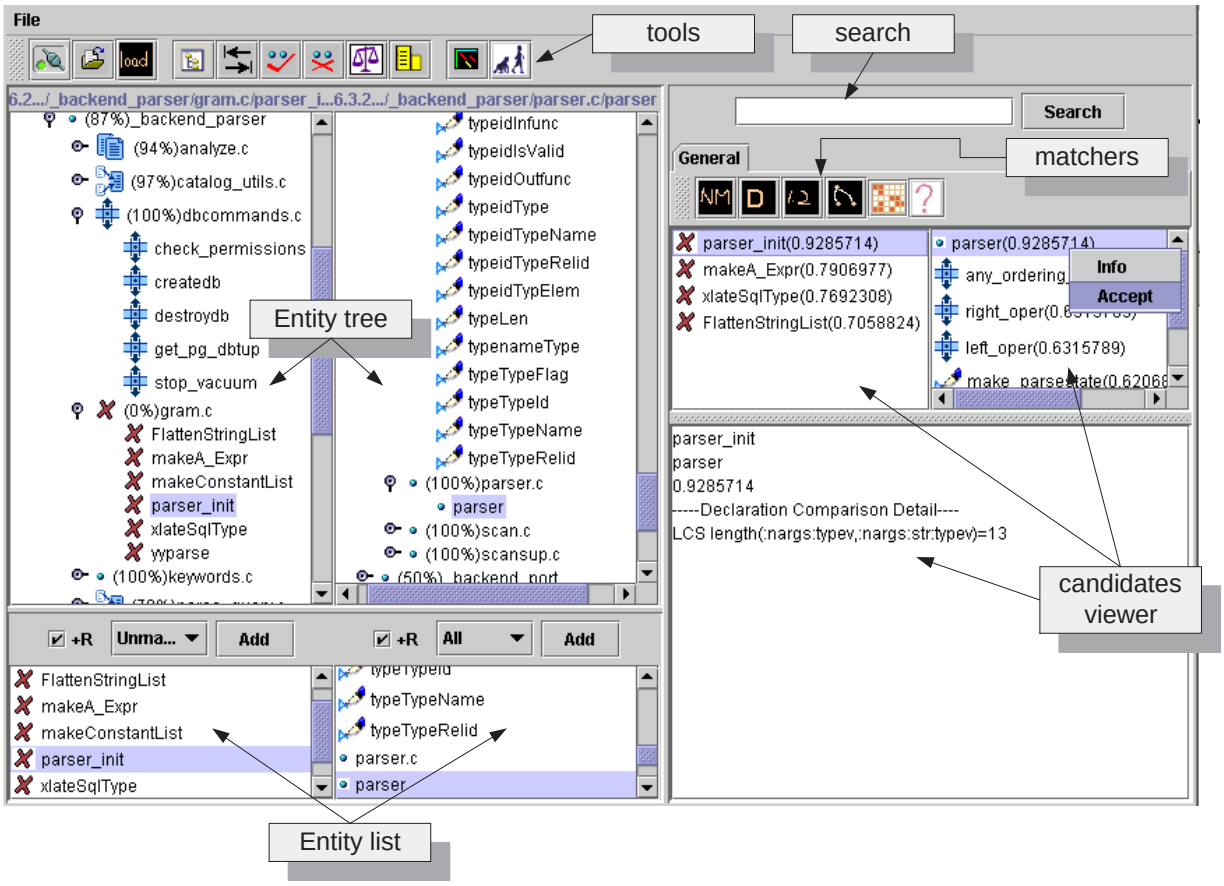


Figure 3.8: *Snapshot of Beagle*

`Scatter plot` is the main tool we use to visualize detailed structural changes between two versions. Moreover, `match status graph` based on dotty [3] displays match status of descendants in a given file or subsystem.

`Match report` summarizes structural changes in a report. It computes the total number of the descendants of a given root entity, and summarizes total number of each type of structural changes.

Although we deliberately divide these components into three groups for explanation purposes, they actually work together in Beagle. For example, when a match candidate in

`candidate viewer` is selected, its image in `entity tree` will be highlighted so that the user can easily know its location in system architecture. Also when a file or function is right clicked in `entity tree`, a `file viewer` or `function viewer` can be triggered if the "info" item on the pop-up menu is selected.

We will describe more details of major components in each group in following subsections.

3.7.3 Basic information about software system

Before the user is able to view the basic information of versions, (s)he needs to connect to Beagle repository and then choose a pair of different versions for the same software system to be studied. Figure 3.9 shows the dialog for connecting to a repository.

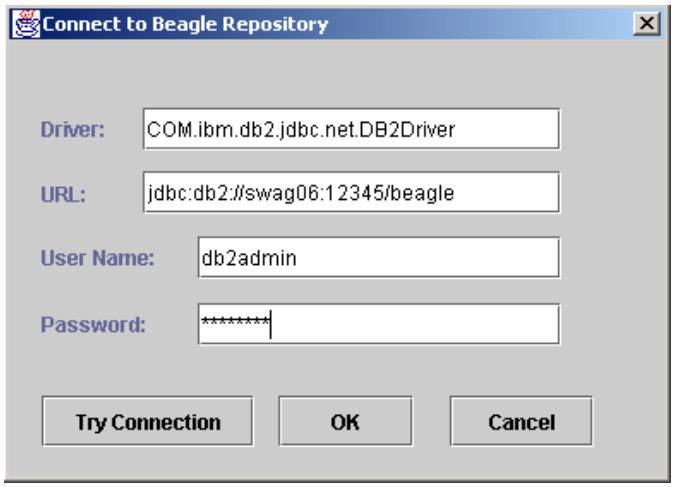


Figure 3.9: *Connect to Beagle repository*

Connecting to a Beagle repository is database independent, as we use the standard SQL database access interface for Java program — JDBC [5] — to perform all the database operations. If repository is successfully connected, a dialog asking the user to choose a pair of versions will be pop up automatically, as shown in Figure 3.10. In this dialog, all the systems available in this repository are displayed in the list box at the top. If the

user selects a system among the list, all the versions for this system will be listed on the left sorted by their release dates. The pair of selected versions are displayed on the right. Direction of origin analysis — either old-to-new or new-to-old — is determined at this time: if the user chooses the first version to be older than the second one, then the direction is old-to-new. Otherwise, the direction is new-to-old.

Once two versions are chosen, entities at different architectural levels in the two versions will be displayed in two `entity tree`s respectively.

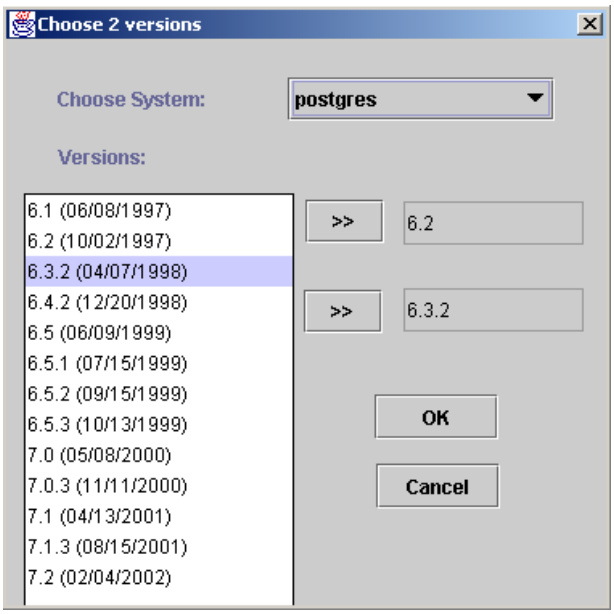


Figure 3.10: *Choose a pair of versions to be loaded*

Note if origin analysis has been performed before for the selected pair of versions, then the user does not have to start from scratch. Instead, (s)he can choose to load matches from Beagle repository into the internal data model and continue to work on them. This feature enables us to reuse previous results and analyze structural changes in an incremental way.

In each `entity tree`, entities are displayed as a tree, with its parent-child relation mapping to "contain" relationship between entities at different architectural levels, as shown in Figure 3.8. Each entity in the tree has an icon attached showing its match status from nine possible values: *deleted*, *inserted*, *unchanged*, *moved*, *renamed*, *merged*,

split, *combined* (having at least two different match statuses listed before) and *unknown* (nothing has been found, the default status). We use an icon, instead of a color, to represent a match status, as it is often difficult to find a color mapping intuitively with a match status. For example, what color should be used to represent status *merged*? Icons can express complex ideas better than colors. Figure 3.11 shows the nine icons we use for the nine match statuses:

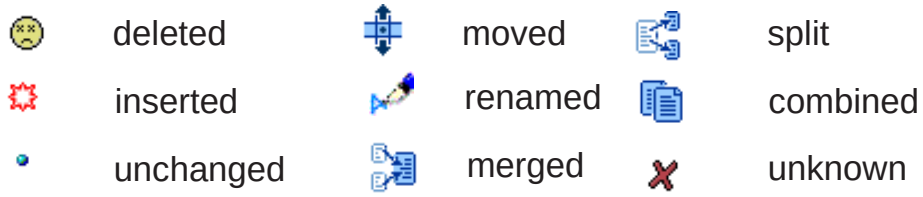


Figure 3.11: *Icons for nine match statuses*

The match status is an entity attribute in the internal data model. Its value is set or changed when matches are identified or deleted. For example, if the user has decided that F is renamed to G in origin analysis, then both F and G will be changed to *renamed* status in the internal data model automatically and their icons in entity tree will be changed as well. Changes to match status are synchronized with changes to the internal data model, thus whether matches are committed to Beagle repository or not, match statuses are always changed when matches are changed.

Simple statistics of how many child entities in percentage have been matched is also displayed for each file and subsystem in the `entity tree`, thus the user can easily determine if and where any unmatched entities might remain.

In additional to the basic operations for exploring a tree structure, such as expansion and collapse, `entity tree` supports navigation between matched entities. If a function F is matched to function G, then if F is double clicked in one `entity tree`, its matching entity G in the other entity tree will be highlighted and be visible; the other direction from G to F also works. If F matches to several Gs, such as split, then all the Gs will be highlighted.

`Entity tree` also supports searching entities by name. If the user specifies the search string to be "index", then all the entities whose name contains substring "index", such as

"create_index", will be highlighted in the entity trees.

`File viewer` and `function viewer` display information related to a file or function, such as metrics, call relations, source path and source code. Both viewers can are triggered by double clicking a file or function in `entity tree`.

3.7.4 Support for Performing Origin Analysis

Beagle has components supporting each basic step of origin analysis. In the remainder of this section, we will use how to perform each basic step as the main thread in our description of these components.

- *Step 1: defining matching space*

  Two `entity lists`, as shown in Figure 3.8, allow two sets of entities to be selected to be involved in matching, the SET_{orig} on the left and SET_{new} on the right. To add entities into each set, the user only needs to select them in the `entity tree` above it, and then add them as a batch by clicking an `Add` button. An option for this action is to add all of their descendants recursively at the same time by selecting `R` checkbox. Another option is to add entities only with particular match status, such as only entities with *unknown* status. The user can also remove entities from an `entity list` by selecting them and pressing the `DEL` key. Provided with the flexibility of editing `entity lists`, the user can easily specify entities to be participated in matching according to different situations.

- *Step 2: matching*

  The major components supporting matching are the five built-in matchers: `name-matcher`, `declaration-matcher`, `metrics-matcher`, `relation-matcher` and `expression-matcher`. In Beagle, they all appear as toolbar buttons, as shown in Figure 3.8. Once SET_{orig} and SET_{new} have been chosen in the two entity lists, clicking one of the matcher buttons will start a new matching process.

  When a matcher is clicked, it may pop up a dialog asking for options specific to the chosen matching process. As matchers usually have specialized configuration options,

dialogs are generally different from each other. Figure 3.12 shows the option dialog for `name-matcher`.

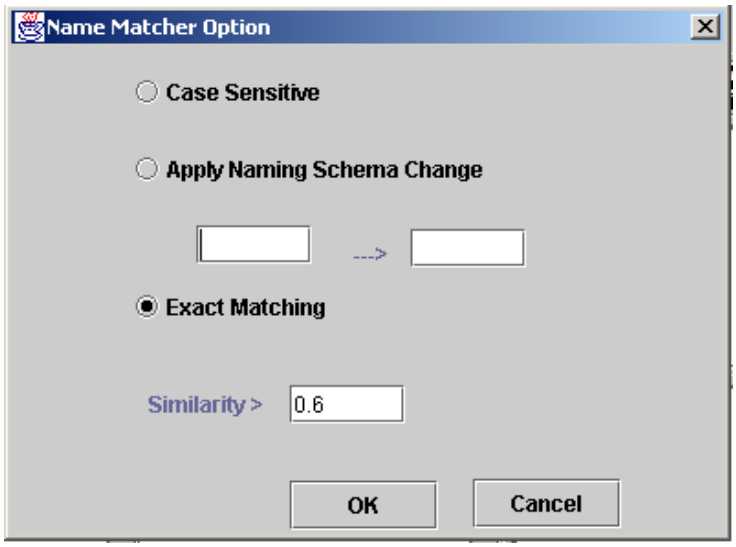


Figure 3.12: *Option dialog of name-matcher*

In this option dialog, for each option for the `name-matcher` as we have already discussed in Section 3.5.2, there is a group of controls that allow the user to specify its value.

Option dialogs for other matchers are similar to that of the `name-matcher`; that is, all provide an interactive way for specifying matching options.

Once options are chosen, the matcher will perform the matching process and produce a list of match candidates. All of the candidates are displayed in `candidate viewer`, as shown in Figure 3.13. `Candidate viewer` displays and allows manipulation of match candidates. How candidates are displayed and manipulated is decided by the viewer attached to each match candidate list, either `simple-viewer` or `expr-viewer` in our current implementation.

If `simple-viewer` is attached, the `candidate viewer` can list all the distinct targets in SET_{orig} on its left corner, ranked by their best candidate. If one of these

targets is double clicked, `candidate viewer` will list all of its match candidates on its right corner, ranked by their similarity values. These rankings indicate which match candidates have high possibility to be real,thus the user can analyze them first. Clicking a candidate will result its detailed match information — often including how the similarity value is computed — displayed at the bottom of `candidate viewer`. Generally, contents of match detail are different for different matchers.

Figure 3.13 shows an example of candidates produced by `relation-matcher`, whose candidates are attached with `simple-viewer`. The user was looking for origins for apparently deleted functions in `heap.c` in PostgreSQL in release 5.0. For function `DeleteTypeTuple`, which was highlighted on the left , two functions in release 6.4.2 on the right were found to be similar with it. The best one was `DeletePgTypeTuple` and why its similarity was 1.0 was displayed at the bottom: their call relations were exactly the same ("==@→foo" means `foo` is a common callee, and "==bar→@" means `bar` is a common caller).

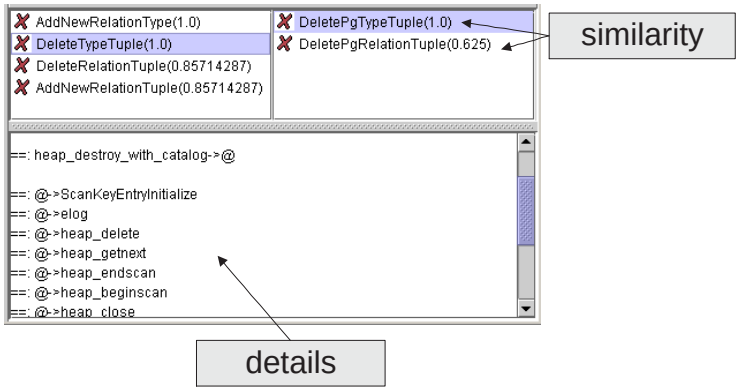


Figure 3.13: *Match detail from relation-matcher*

With the other viewer, `expr-viewer`, the user can further specify which match candidates should be selected by inputting a boolean expression. As we have already mentioned in Section 3.5.2, with `expression-matcher` integrating matching results from other matchers and `expr-viewer` providing query capability, the user is able to try different combinations of matching strategies.

- *Step 3: decision*

  The user may need to check various sources of information (*e.g.*, source code) before (s)he decides on real matches. This decision support is provided by components like `entity trees`, `candidate viewer` and `file viewer`. Once the user has decided that a pair of match is real, (s)he can specify the type of structural changes in a pop up window and accept the match. Change will be visible at once in Beagle, as the icon for each entity displaying match status will be updated.

  Newly accepted matches actually only affect data in the internal data model. They can be either committed or rolled back: if they are committed, they will affect the persistent data model in Beagle repository and will be still visible in the next run of Beagle; if they are rolled back, no change will be made to the Beagle repository, and match statuses will be changed back in the internal data model, resulting entity appearances in `entity tree` being changed back too.

  The transaction support allows the user to "try matching", which is useful in identification of chained structural changes, as it requires multiple iterations, and discovery of matches in later iterations rely on matches identified in previous iterations. Figure 3.14 shows the window that lists all the uncommitted matches. Using the two buttons `Commit` and `Rollback`, the user can commit or rollback selected matches from the uncommitted match list.

- *Other support*

  There is also other support in Beagle for origin analysis. For example, the user can perform a traditional matching on two selected root entities from two versions. In a traditional matching, two descendent entities are considered to be a match if and only if they have same paths to their selected root. The user can also match two entities manually if (s)he finds strong evidence it, even if the five built-in matchers fail to identify it. The user can also delete a match, if (s)he realizes a match is incorrectly identified.

  As mentioned before, the direction of origin analysis — old-to-new, or new-to-old — is specified when two versions are chosen. As it is useful to perform origin analysis from both directions for a given pair of versions, Beagle supports switching between

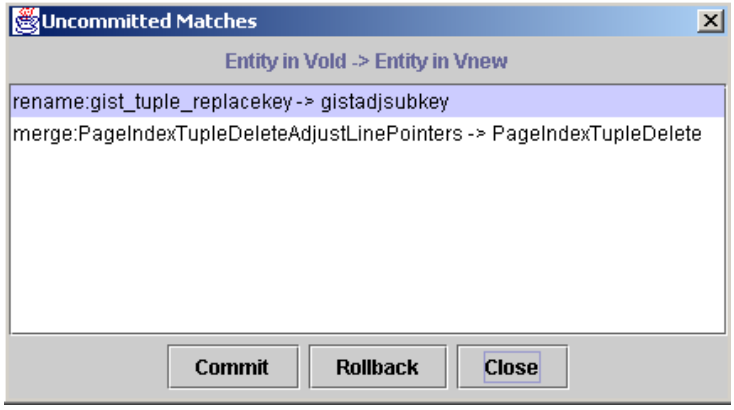


Figure 3.14: *Commit/Rollback matches*

directions during origin analysis without losing any intermediate results. The switching operation is simple; the user only need to click a button, then the two `entity trees` and `candidate lists` dedicated for two chosen versions will be switched, resulting the direction of origin analysis being changed too.

There is a small helper tool that compares call relations of two *sets* of functions. During the comparison, callers or callees that are within the set are omitted; only callers and callees outside the set are considered to be callers and callees of the set. Comparing call relations of a group instead of a single function is useful in merge/split detection, where the union of callers or callees in a suspected merge/split instance needs to be analyzed to identify patterns of merges/splits. We will describe details of merge/split detection using origin analysis in Section 3.8.2.

3.7.5 Visualization and reporting

Beagle supports a variety of visualization tools for browsing structural changes of a software system. One is `scatter plot` viewer, as shown in Figure 3.18. Scatter plots are well known in clone detection research; the basic idea is that entities of interest (say functions or even lines of code) are lined up along the X and Y axes, and dots or colored marks are used to indicate the presence of an "interesting property" (or "hit"), usually that there is a

nontrivial similarity between two entities.

In clone detection, it is typical to put the same entities along the X and Y axes (to make the visualization feasible for large systems, sometimes only subsets of the system's entities are used). Of course, the diagonal should be a solid line of "hits", but often other patterns reveal themselves too, such as where several consecutive lines of code in different parts of the system are similar, indicating that cloning may have occurred. Figure 3.15 shows a scatter plot in clone detection. The number on axes are lines of code in one software system; dots in the plot represent code clones.

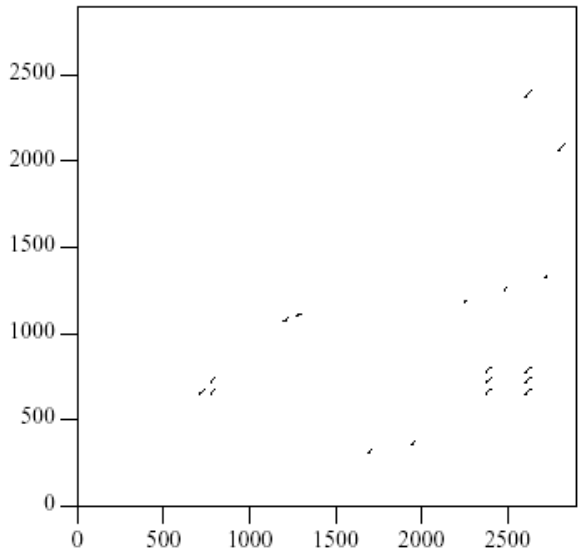


Figure 3.15: *Scatter plot in clone detection*

In origin analysis, we put two different versions of a system along the X and Y axes. Entities on the axes, including files and functions, are ordered by their names with directory information. On either axe, functions in the same file are next to each other. The left part of Figure 3.16 shows an example of this. On both axes, blue rectangles represent functions; white ones are files.

We have different types of "hits", corresponding to different usages. Before we show how to use scatter plot in different ways, we first describe how we address the scalability

problem of scatter plot.

Scatter plots have limited scalability. When the number of entities to be visualized is huge, the plot will also be huge, making it hard to gain a global view and examine details at the same time. To address this problem, we create two views for a scatter plot: *bird's eye view* and *detailed view*. Bird's eye view is actually a zoom-out version of the original scatter plot. The user can use this global view to examine all the changes comprehensively and discover patterns at a high level. If the user wants more details about a particular area in bird's eye view, (s)he can select this area, and use a detailed view to get more information. Figure 3.16 displays the bird's eye view (on the right) and detailed view (on the left).

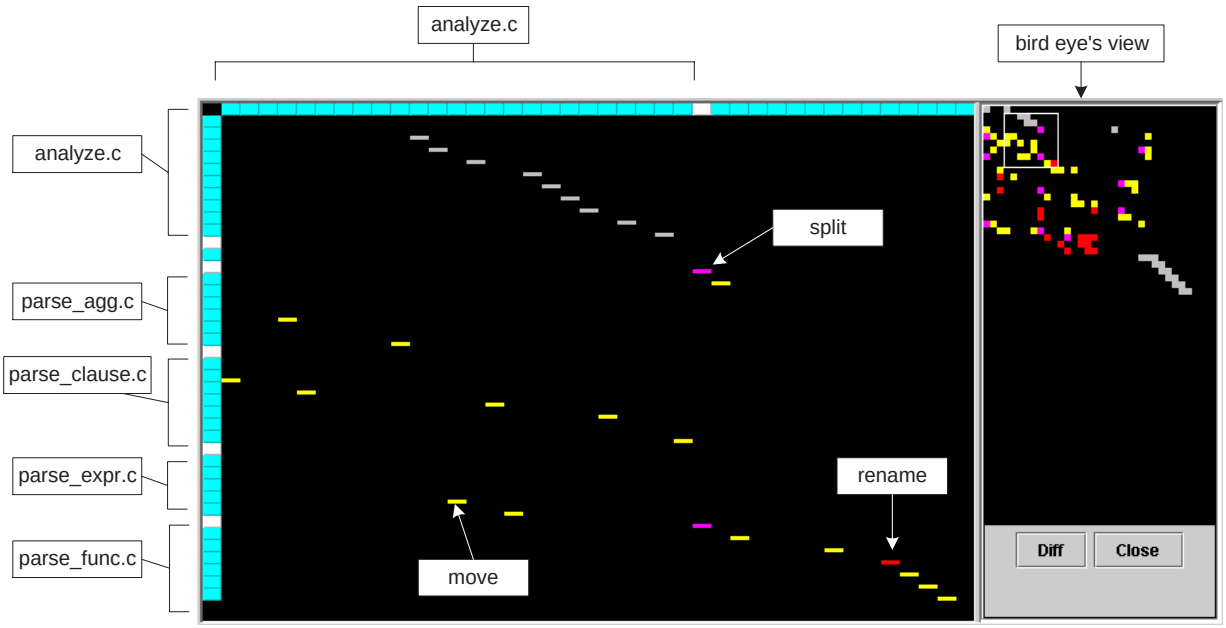


Figure 3.16: *Detailed view and bird's eye view of scatter plot*

We first use a scatter plot to visualize structural changes already identified between two given sets of entities from two versions. We use a "hit" with grey color to indicate a traditional match (a pair of entities with *unchanged* status). For other types of structural changes, we use "hit"s with different bright colors. If a merge happens, the entity after the

merge will have multiple "hit"s with those entities before merge. The "one-to-many-hits" also applies in split case.

We expect to see many grey "hits" along the diagonal, representing stable entities, but we also expect to see colored "hits" or breaks, where entities have been changed, added, or deleted between versions. By observing the graph, we are able to gain a global view of what have happened at a high level. For example, Figure 3.17 shows a stable file in a release change: no function was deleted and no function was inserted; no structural change occurred. Figure 3.18 shows file `heap.c` changing from release 6.4.2 to release 6.5 in our case study of PostgreSQL. We can see that some functions in this file were renamed (red rectangles), but in overall, this is still a stable file. Figure 3.16 shows the restructuring of `parser` subsystem (containing over 150 functions) from release 6.2 to release 6.3.2 in PostgreSQL. A large number of functions moves (yellow rectangles) and renames occurred.

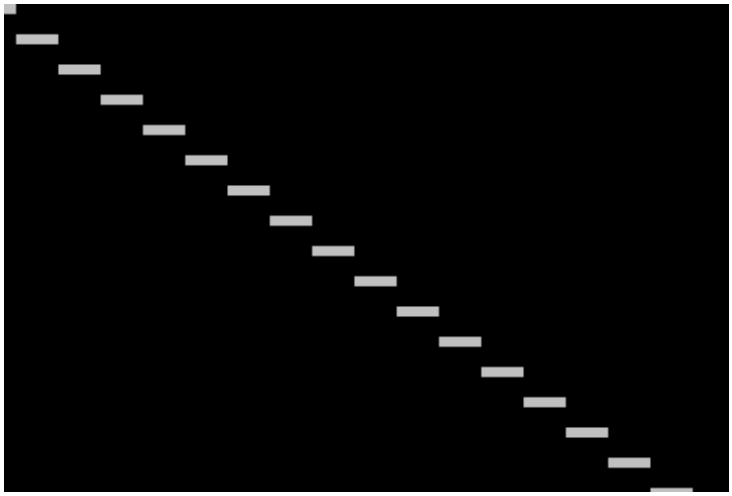


Figure 3.17: *Stable subsystem*

As another usage, we are able to identify moves/merges/splits at high level by examining scatter plots. For example in Figure 3.16, file `analyze.c` in release 6.2 split off a set of new files in release 6.3.2, such as `parse_agg.c` and `parse_clause`.

We also use scatter plot to visualize other interesting properties in structural change identification. For example, for each function that exists in both versions, we visualize

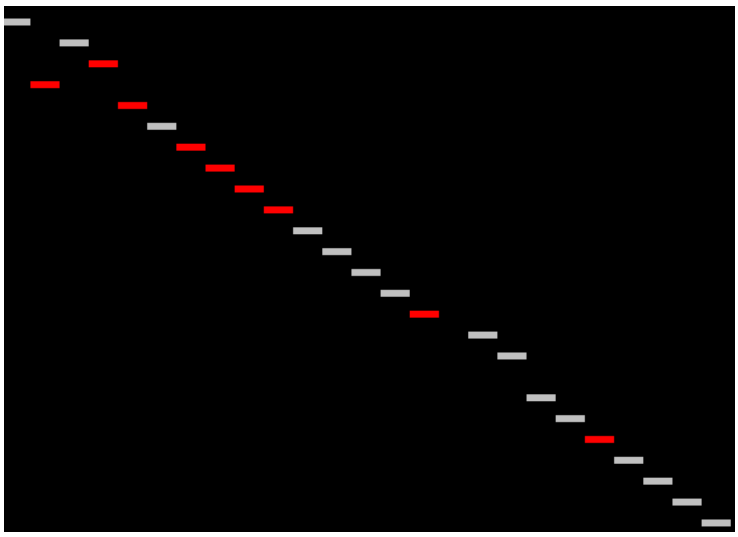


Figure 3.18: *Nearly stable subsystem*

the similarities of its callers and callees respectively. For either callers or callees, we use three different colors to indicate three cases of the number of change: no difference, little difference, and big difference. We are particularly interested in functions that have big decrease in their callees, as it may indicate that splits have occurred. Figure 3.19 shows an example from our case study of PostgreSQL: from release 7.1.3 to 7.2, 17 functions originally called by function `gistbuild` disappeared while a new function called `gistbuildCallback` became a new callee. This finally led to discovery of a function split.

Another visualization tool we have implemented in Beagle is a hierarchical tree showing match status of an entity's descendants, as shown in Figure 3.20. Comparing with `entity tree`, which displays match status for each entity as well, this graph is able to display a large number of entities at multiple architectural levels at the same time. It is easier to see how many entities have been matched and how many not. This visualization technique is based on the tool dotty [3].

We also use a report to summarize matches in a file or subsystem, including total number of entities in the file/subsystem, total number of entities that have performed each type of structural change, and total number of entities that still remain unmatched *etc.*.

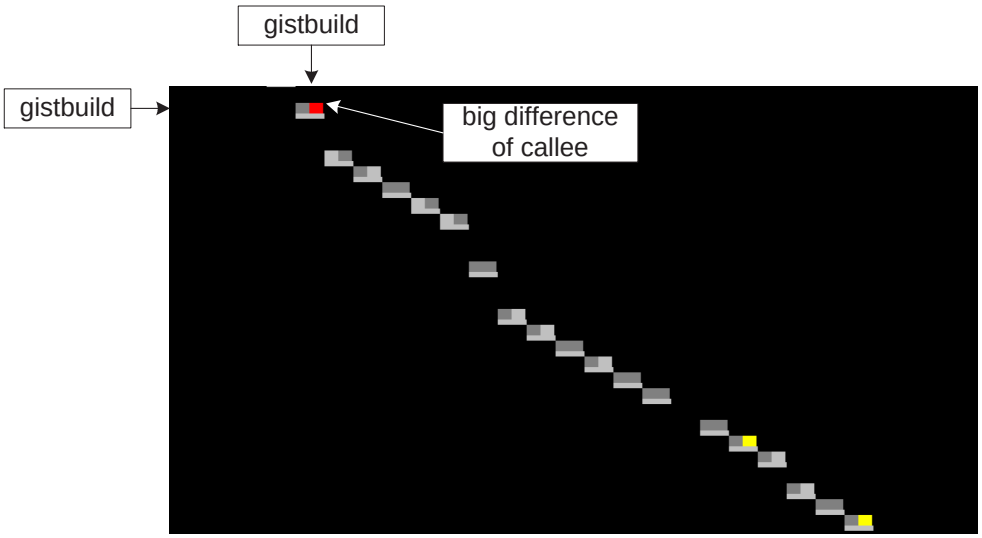


Figure 3.19: *Scatter plot as an aid in detecting function split*

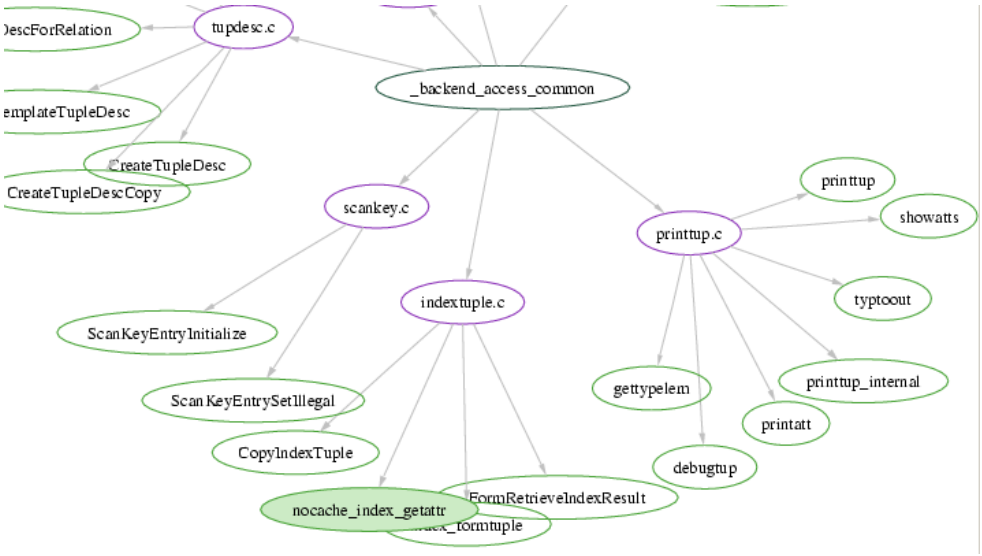


Figure 3.20: *Match status as a tree*

Figure 3.21 shows a report example.

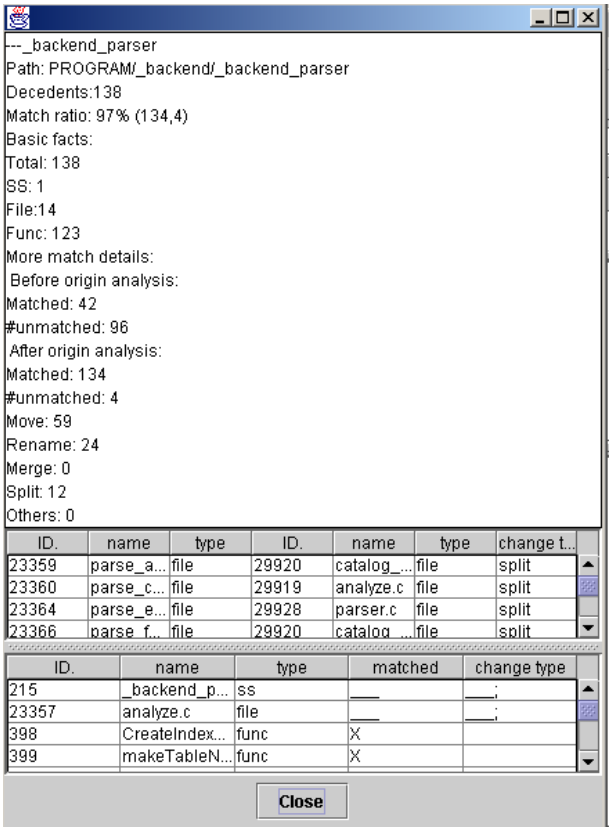


Figure 3.21: *Report summarizing structural changes*

3.8 Detecting Structural Changes Using Beagle

Structural changes can occur at different architectural granularities. When they are at the subsystem level — as files and subsystems are broken up, merged, and moved around — significant changes to the design of the software system are being effected. When structural changes are performed at the function level, this often reflects a fine tuning of the design, as maintainers may strive to improve the cohesion of a function, file, or class or lessen its coupling with other design entities. Since changes at the higher levels of design (i.e., file and

subsystem) can be often inferred from changes at the lower levels, we have concentrated our efforts on extracting and modeling information about structural changes at the function level.

In the remainder of this section, we will describe how structural changes are identified. First Section 3.8.1 shows how to identify move and rename at the function level. Section 3.8.2 describes how to identify merges and splits at the function level. Then Section 3.8.3 explains detection of chained structural changes at the function level. Finally, Section 3.8.4 describes identification of various kinds of structural change at the file level.

3.8.1 Move and Rename at the Function Level

We define a move as the case that an entity moves to another file or subsystem *without* changing its name. If the name is changed at the same time of move, we call it a case of rename instead.

According to our definition of a move, detecting move instances is a rather simple task: only entities apparently deleted from V_{orig} and entities apparently inserted in V_{new} need to be chosen in the matching, and only `name-matcher` need to be applied to find entities with the same names.

For a renaming, often we need to consider only the entities that are apparently deleted from V_{orig} and the entities apparently inserted in V_{new} in matching. However, due to different cases of renames, multiple matchers may be needed in detecting renamings. For example, if there was only a small change to the name, then `name-matcher` might be able to detect the change; if the name changed significantly while the function declaration remained the same, then `declaration-matcher` will work best. As it is generally easier to make decisions for candidates produced by strict matching strategies, such as exact declaration matching or "name>0.8 & decl>0.8" in `expression-matcher`, it is often more efficient to try strict matching strategies first.

Sometimes a group of renames may follow a common changing schema, such as removing the leading underscore. There are various cases that this may occur, as we will see in our case study in Chapter 4. These schemas may be documented, or be observed from renaming instances already identified. If we know or suspect such a case, choosing `name-matcher` with the schema specified as part of the matching options, can be very useful in detecting

more renames in the same group.

3.8.2 Merges/splits at the Function Level

Detection of N-way merges/splits at the function level is based on a detailed analysis of call relations. Let us now consider how merging and splitting can affect the call relationships between the various program entities. To simplify discussions somewhat, we will let $N = 2$. Figure 3.22 shows the before and after of two functions, F_1 and F_2, being merged into a single new function, G. Let us assume that in_1, in_2, and in denote the callers (clients) and out_1, out_2, and out denote the callees of F_1, F_2, and G respectively.

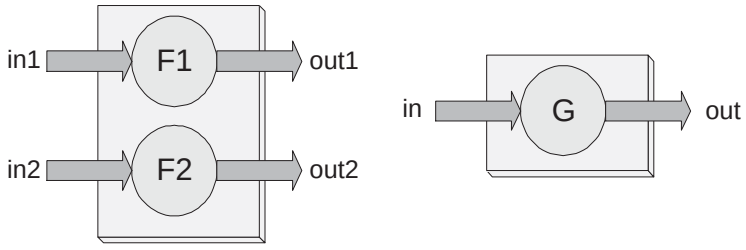


Figure 3.22: *Canonical two-way function merge*

While there are many reasons why merges may occur, we have found three cases that are relatively easy to detect by examining the call relationships:

1. *Clone elimination* — Two (or more) functions that perform similar tasks are merged into one function in the new version.

   A strong indicator of this phenomenon is

   - $in_1 \cap in_2 \approx \emptyset \quad \wedge \quad in_1 \cup in_2 \approx in$
   - $out_1 \approx out_2 \approx out$

   That is, F_1 and F_2 have no clients in common (if they are clones, why would one call both?), and the union of the clients is the client set of the new function. Since the three functions have roughly the same functionality, the set of outgoing calls for each should be highly similar.

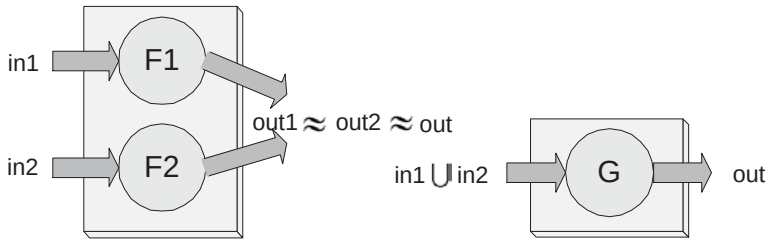


Figure 3.23: *Clone elimination.*

2. *Service consolidation* — Two (or more) functions that perform different services, but are called at the same time by the same clients, are merged into a new, larger function.

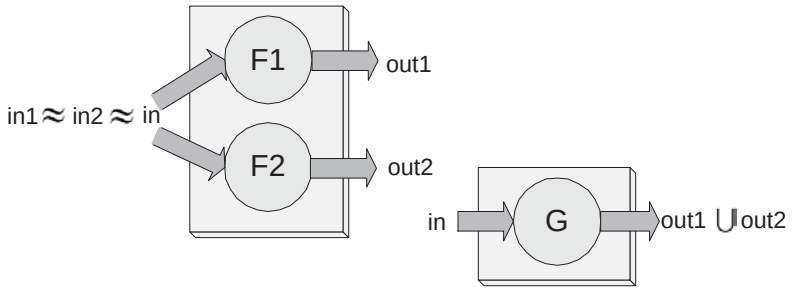


Figure 3.24: *Service consolidation.*

A strong indicator of this phenomenon is:

- $in_1 \approx in_2 \approx in$
- $out_1 \cup out_2 \approx out$

That is, the client sets of F_1 and F_2 are similar to each other as well as to the client set of the new function G, and the union of the callees of F_1 and F_2 are similar to that of G. Since F_1 and F_2 perform different tasks, there is no presumed overlap in the callee sets.

3. *Pipeline contraction* — A function F_1(the service provider) is only ever called by a

single client F_2. In the new version, either the client F_1 consumes functionality of the service provider F_2 directly, or a new function is created that merges both the client and service provider.

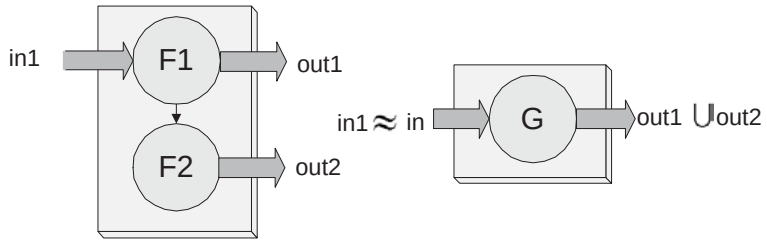


Figure 3.25: *Pipeline contraction.*

A strong indicator of this phenomenon is:

- $F_2 \in out_1 \quad \wedge \quad in_2 = \{F_1\} \quad \wedge \quad in \approx in_1$
- $out_1 \cup out_2 \approx out$

That is, the "service provider" function F_2 is called only by its single client F_1 in the old version, and the client set of F_1 and G are highly similar. Furthermore, the callee set of the new function is similar to the union of the callee sets of F_1 and F_2.

Since, at least structurally, a split is the dual operation of a merge, we note that the analogous patterns of

4. *clone introduction*,
5. *service extraction*, and
6. *pipeline expansion*

We detect merges/splits at function level by iteratively applying origin analysis. After several iterations, we may find multiple candidate functions appear to have the same origin, which indicates that a merge or split may have occurred. In this case, we examine the detailed change of call relations to see why it happened. The helper tool comparing call relations of two sets of functions discussed in Section 3.7 helps us in this phase.

3.8.3 Chained Structural Changes at the Function Level

There are cases that structural changes — including move, rename, merge and split — are chained; that is, entities involved depend on each other, thus discoveries of these changes also depend on each other. Our iterative process of origin analysis proves to be useful to deal with this situation. Here we give a real example taken from our case study of PostgreSQL from release 6.4.2 to 6.5. At first, eight functions in `geqo_eval.c` in `optimizer_geqo` subsystem, seemed to have been deleted. After performing some origin analysis, we found that some functions had actually been merged into file `joinrels.c` in `optimizer_path` subsystem in release 6.5.

The call relations of eight functions in `geqo_eval.c` and `joinrels.c` in the two releases are shown in Figure 3.26 and 3.27 respectively. This example is complicated, so for the sake of simplicity we have adopted some labelling conventions: a circle with a capital letter label — such as A — denotes a "function of interest"; a rectangle with a lowercase label and a number in parentheses — such as $h(8)$ — denotes a set of functions that are callees of the functions of interest, with the number indicating the cardinality of the set. A white box indicates that this set of callees were callees only in one version; a grey box indicates that they were callees in both versions. A grey box that has the same letter label but a smaller (or larger) number denotes a subset (or superset) of the original callee set.

In the first iteration of origin analysis, we decided that E had been renamed to I, based on their similar caller sets (not shown in the diagrams). Then we noticed that both D and H have seven callees in common with (O) (since $h(8)$ and $h(7)$ have seven common functions). After close examination, we concluded that D and H had been merged into O. Next, we noticed that — after taking the above merging into account — the caller and callee sets of C were similar to those of N: they have one common callee in A, and now D and H had been matched to O. Also, G was now appeared to be similar to N: they have a matched caller E and I, and one common callee in $j(1)$ and $j(3)$, plus callee H had been matched O. After examining the source code, we decided that C and G had indeed been merged into N, and by similar chain of evidence that B and F had been merged into M. Thus, we can see that by applying matching iteratively, we succeeded in detecting three chained merges that had occurred at the same time.

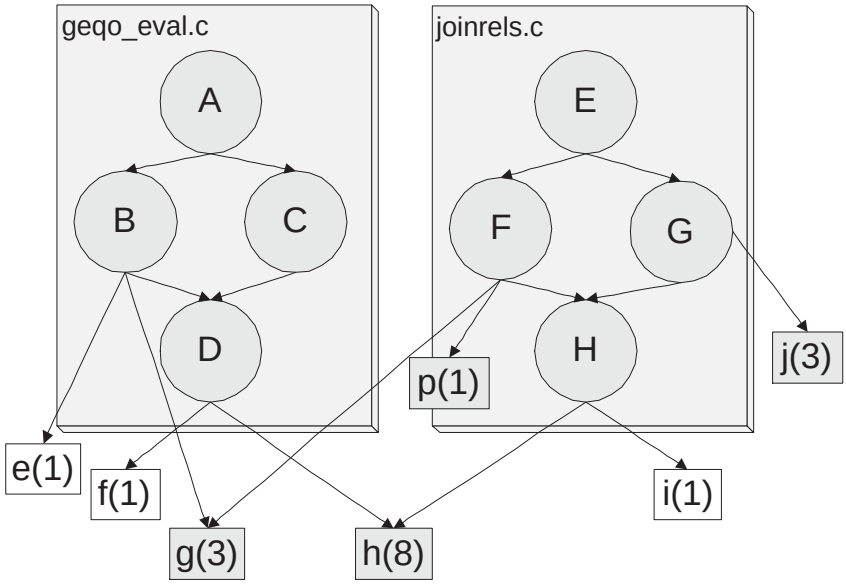


Figure 3.26: *Call relations in release 6.4.2*

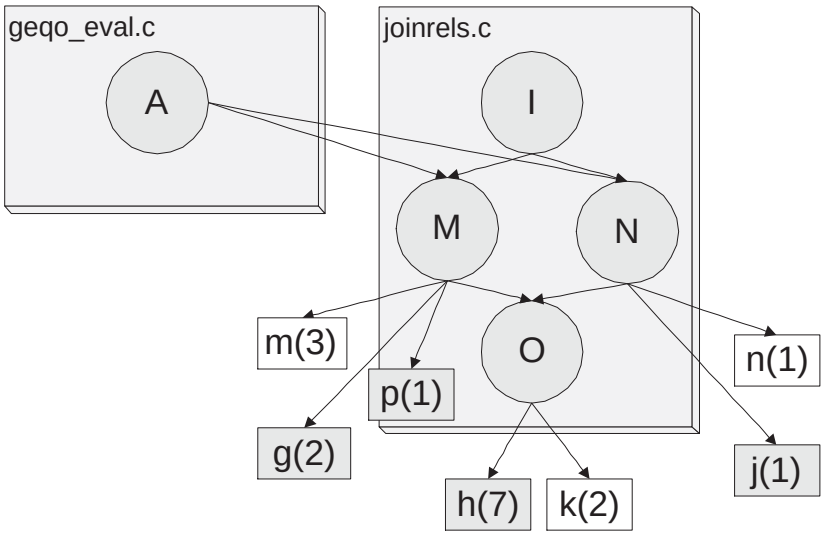


Figure 3.27: *Call relations in release 6.5*

3.8.4 Detecting Move/rename/merge/split at the File Level

In general, changes at the file level are detected manually in Beagle. Suppose a new file G is found to be composed of functions from an old file F; if they have the same name then we say "F moves to G", otherwise, we say "F is renamed to G". If a new file G is found to be composed of functions from two old files F_1 and F_2, then we say "F_1 and F_2 are merged to G". Similar is definition for split at the file level.

Figure 3.28 is the same as Figure 3.16 that we described before. In this figure, entities from the old version and new version are located on X and Y respectively. We can see from this figure that file `analyze.c` in the old version split off a few files, including `parse_agg.c`, `parse_clause.c`, *etc.*, as most functions in these new files were functions that moved from `analyze.c` in the old version.

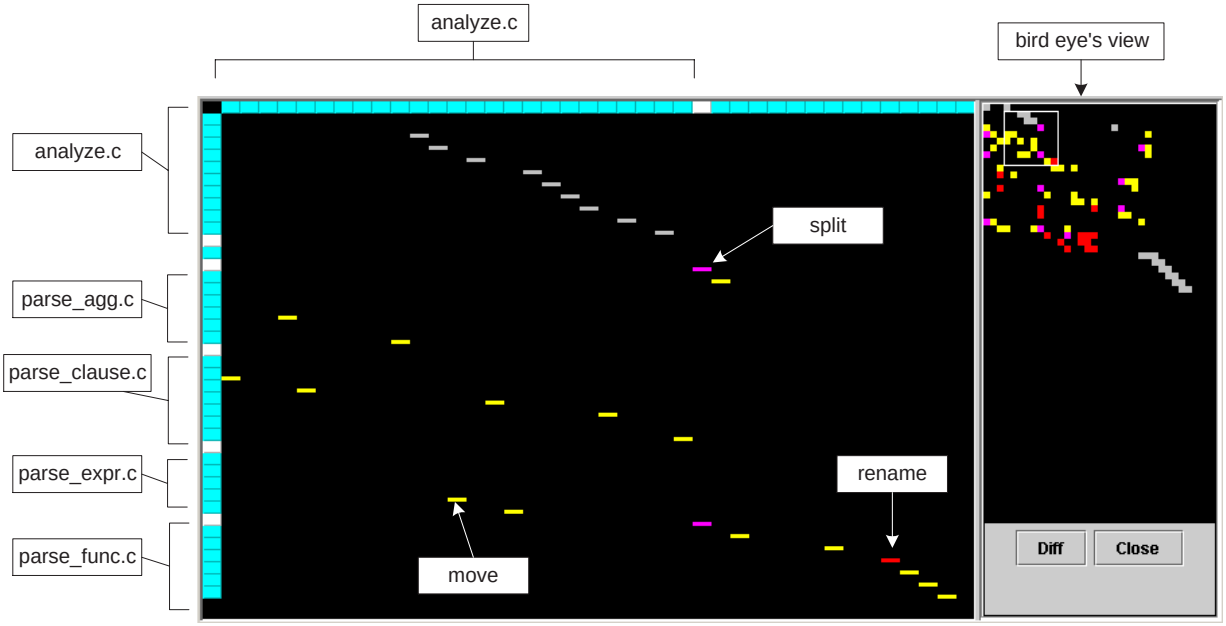


Figure 3.28: *Detect file merge using scatter plot*

3.9 Summary

In this chapter, we present our improvements to origin analysis, mainly including a generalized model of matching and an iterative and semi-automatic process. These improvements enable us incorporate human knowledge into the analysis process; they also enable us to add or remove matching techniques easily. Using the improved origin analysis, we are able to detect four types of structural changes, including rename, move, merge and split, both at the function level and file level. We can also detect structural changes that are "chained" together using multiple iterations.

We describe the new tool Beagle that implements these improvements to origin analysis, focusing on its data flow, architecture, and other new features supporting software evolution, such as navigation between entities involved in a structural change and transaction support for origin analysis.

We borrow a technique called scatter plot to visualize structural changes between two versions. We use it in both understanding the changes, and inferring structural changes at high level from low level.

In the next chapter, we will describe the case study we have performed on an open source software project PostgreSQL using techniques described in this chapter.

Chapter 4

Case Study

In this chapter, we describe a case study of structural changes in an open source software system PostgreSQL. In previous work on origin analysis [36], only two releases were studied. In our case study, we have applied origin analysis to 12 releases (covering more than four years) of the PostgreSQL `backend` subsystem. In overall, we have identified a large number of structural changes. We observed many interesting phenomena during our study, such as "ripple effect" of changes (several changes caused by the same reason spanning multiple releases), patterns of function merging/splitting, and function renaming caused by data structure renaming. We categorized function moves to mine more knowledge. We found that most functions move between existing files, and most functions move within subsystems.

We will first introduce the candidate system PostgreSQL in Section 4.1. Then we briefly describe all the structural changes identified in Section 4.2. In Section 4.3, we present detailed structural changes in each release change. After that, in Section 4.4, Section 4.5, and Section 4.6, we describe our further analysis and summary of different types of changes, including function moves, function renames, and merges/splits.

4.1 PostgreSQL

PostgreSQL [7] is an open source object-relational database management system (ORDBMS), originally based on the POSTGRES system developed at the University of Cali-

fornia at Berkeley. The original POSTGRES project started in 1986; it was abandoned in 1993 only to be reborn the following year as Postgres95. An SQL language interpreter was then added, and its performance and maintainability were greatly improved due to many internal changes. In 1996, the project was renamed as PostgreSQL, and since then many new features have been added. It continues to evolve and is in widespread used, especially within the Linux community. We have chosen to study it, as it is a well-known piece of software of significant size and complexity.

For our case study, we selected 12 releases of PostgreSQL from 6.2 (Oct. 1997) to 7.2 (Feb. 2002). We decided to focus on the `backend` subsystem, which implements all of the server functions. The `backend` subsystem is the largest in PostgreSQL; it comprises about 70% of the whole code base.

In raw numbers, from version 6.2 to 7.2 the `backend` subsystem of PostgreSQL grew

- from 186 KLOC to 279 KLOC,
- from 328 to 388 files, and
- from 3262 to 4531 functions.

Figure 4.1 shows the LOC (lines of code) of the `backend` subsystem in each release in the study. We can see from this figure that PostgreSQL evolves continually in the four years with an annual growth rate about 10%.

4.2 Summarized Structural Changes

We performed origin analysis on each consecutive pair of the 12 releases, including six major release changes (in which the first or second number in the release number is changed, *e.g.*, 6.4.2 to 6.5) and five minor release changes (in which no change to the first or second number in the release number, *e.g.*, 7.0 to 7.0.3). We have summarized the number of each type of structural change in each release change in Figure 4.2 (Y axe is the number of changes). We note that the structural changes we have identified is only a subset of all the structural changes that existed, of which we have no information about its total number. For all the changes that we have identified, as we examined the source code and CVS log before we made decision, we know that they did happen.

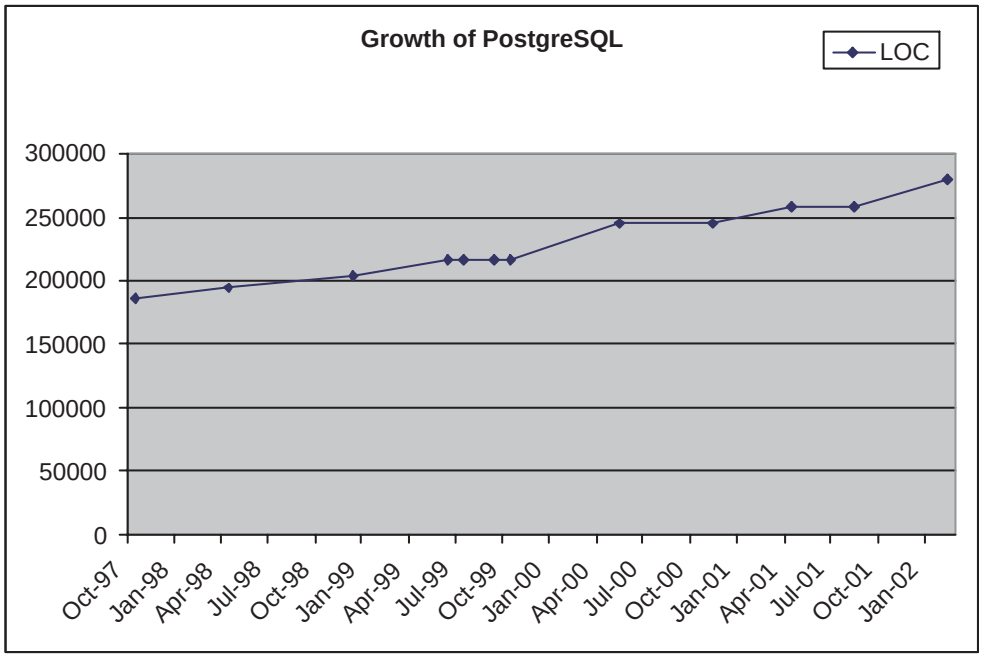
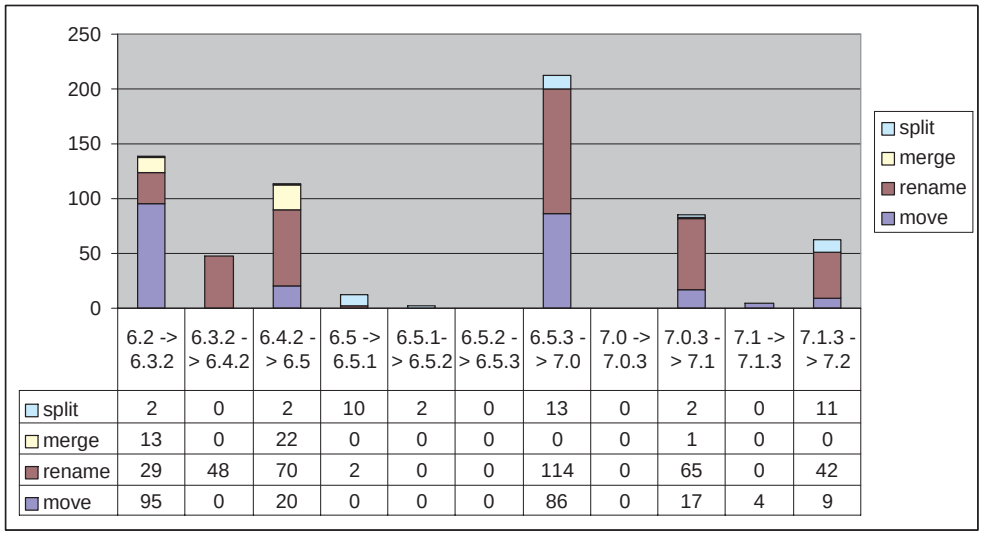


Figure 4.1: *Growth of PostgreSQL*

Perhaps unsurprising, we found that structural changes occurred much more often during major release changes than that in minor release changes: the six largest number of changes all happened in major release changes. Among the four minor release changes, the largest number was 12 (from release 6.5 to 6.5.1), which was still much less than the smallest number of all the major release changes (48 from release 6.3.2 to 6.4.2).

Two types of structural changes — move and rename — had largest number of instances, 231 and 370, which together accounted more than 88% of all the changes. Merges and splits occurred less often, and the total number of instances we have detected in our study was 76.

Once matches have been detected using origin analysis, the number of apparently deleted entities in V_{orig} and apparently inserted entities in V_{new} will be reduced, which gives a measure of the degree that origin analysis has helped us towards a correct understanding of the system structure. In Table 4.1 and Table 4.2, we list the change of the apparently deleted and apparently inserted entities in six major release changes and five



	6.2 -> 6.3.2	6.3.2 -> 6.4.2	6.4.2 -> 6.5	6.5 -> 6.5.1	6.5.1-> 6.5.2	6.5.2 -> 6.5.3	6.5.3 -> 7.0	7.0 -> 7.0.3	7.0.3 -> 7.1	7.1 -> 7.1.3	7.1.3 -> 7.2
split	2	0	2	10	2	0	13	0	2	0	11
merge	13	0	22	0	0	0	0	0	1	0	0
rename	29	48	70	2	0	0	114	0	65	0	42
move	95	0	20	0	0	0	86	0	17	4	9

Figure 4.2: *Structure changes in PostgreSQL*

minor release changes respectively.

	before origin analysis		after origin analysis				change	change
Vorig->Vnew	#deleted	#inserted	#deleted	#inserted	diff #deleted	diff #inserted	rate of #deleted	rate of #inserte
6.2 -> 6.3.2	219	286	91	154	128	132	58%	46%
6.3.2 -> 6.4.2	208	263	160	215	48	48	23%	18%
6.4.2 -> 6.5	213	368	114	284	99	84	46%	23%
6.5.3 -> 7.0	509	954	309	741	200	213	39%	22%
7.0.3 -> 7.1	703	913	618	827	85	86	12%	9%
7.1.3 -> 7.2	199	533	144	471	55	62	28%	12%

Table 4.1: *Change of apparently deleted/inserted entities in major release changes*

Comparing Table 4.1 with Table 4.2 we can see that before origin analysis, the numbers of apparently deleted (column 1) and inserted entities (column 2) in major release changes are much bigger than that in minor release changes. The average number of apparently deleted entities and inserted entities equal to 342 and 553 in major release changes, while in minor release changes they only equal to 4 and 9; particularly from release.5.2 to 6.5.3, no

Vorig->Vnew	before origin analysis		after origin analysis		diff	diff	change	change
	#deleted	#inserted	#deleted	#inserted	#deleted	#inserted	rate of	rate of
6.5 -> 6.5.1	9	13	7	1	2	12	22%	92%
6.5.1-> 6.5.2	3	6	3	4	0	2	0%	33%
6.5.2 -> 6.5.3	0	0	0	0	0	0	NA	NA
7.0 -> 7.0.3	1	7	1	7	0	0	0%	0%
7.1 -> 7.1.3	6	17	2	13	4	4	67%	24%

Table 4.2: *Change of apparently deleted/inserted entities in minor release changes*

apparently deleted or inserted entity at all. This difference contributes to the big difference of the total number of structural changes identified between major release changes and minor release changes, as shown in Figure 4.2.

We now focus on major release changes. Columns "diff #deleted" and "diff #inserted" in Table 4.1 list how many apparently deleted entities and apparently inserted entities were reduced after origin analysis. The last two columns also represent the same information, but in percentage formats. We can learn from this table that, on average in our case study, origin analysis helped us to identify origins of 35% apparently deleted entities and 22% apparently inserted entities during major release changes. In some cases, such as from release 6.2 to 6.3.2, the two ratios reach as high as 50% and 46%.

4.3 Detailed Structural Changes

In this section, we describe structural changes identified in each consecutive release change.

4.3.1 From Release 6.2 to 6.3.2

From release 6.2 (Oct 1997) to release 6.3.2 (Apr 1998), many new features and improvements were added into PostgreSQL, such as supporting full SQL92 subselect capability and socket interface for client/server connection. Before performing origin analysis, we found that there were 3389 entities exist in both releases. As shown in Table 4.1, 219 entities apparently deleted from release 6.2 and 286 entities apparently inserted into release 6.3.2. Using origin analysis, we found 95 instances of move, 29 instances of rename, and 13

merges, which reduced apparently deleted entities by 58% and apparently inserted entities by 46%, also shown in Table 4.1. Of all the instances of structural changes, most of them occurred in `parser` subsystem.

In `parser`, functions in "old" files were redistributed throughout the subsystem; some were placed in existing files, while others were grouped into "new" files. Most functions were left intact themselves. For example, 23 functions were moved out from `analyze.c` in release 6.2 into six new files — `parse_agg.c`, `parse_clause.c`, `parse_func.c`, `parse_oper.c`, `parse_target.c` and `parse_expr.c`, and 19 functions were moved out from `catalog_utils.c` into files `parse_func.c`, `parse_oper.c`, `parse_agg.c` and `parse_type.c`.

New files in release 6.3.2 were formed by functions in different old files. For example, the new file `parse_agg.c` was actually composed of function `agg_error` in `catalog_utils.c`, four functions in `analyze.c` and `ParseAgg` in `parse.c` in release 6.2, and file `parse_func.c` was composed of four functions in `analyze.c` and eight functions in `catalog_utils.c`.

The result of the reorganization is: functions related to "aggregates" from different files in release 6.2 were grouped into file `parse_agg.c` in release 6.3.2; similarly, functions dealing with function calls were regrouped into file `parse_func.c`, and functions related to clauses, operators, nodes and relations were regrouped into files `parse_clause.c`, `parse_oper.c`, `parse_node.c`, and `parse_relation.c` respectively. Compared to the old operation-based file grouping — that all the functions of parsing, transforming from a parse tree into a query tree and utilities were in `parser.c`, `analyze.c` and catalog_utils.c respectively — the new file grouping was based on objects being manipulated, such as all the functions related to aggregates were grouped into `parse_agg.c`.

Based on the structural changes at the function level, we identified merge/splits at the file level, such as `analyze.c` split to `parse_agg.c`, `parse_clause.c`, `parse_func.c`, `parse_oper.c`, `parse_target.c`, and `parse_expr.c`.

For this `parser` restructuring, we later found evidence in the CVS log of PostgreSQL that said "break parser functions into smaller files, group together".

We also identified three file moves between different subsystems: file `aclchk.c` moved from `tcop` subsystem to `catalog` subsystem, `variable.c` from `tcop` to `commands`, and file `dbcommands.c` from `parser` to `commands`. These file moves caused that 33 functions appeared to be deleted from the old version and 33 appeared to be inserted into the

new version before origin analysis. The evidence we found for the three file moves were some descriptions in the CVS log, including "move variable.c to commands and aclchk.c to catalog", and "move dbcommands.c to commands. It should not be in the parser directory".

We identified total 29 renamings. Among them, 21 occurred to functions in `catalog_utils.c` in release 6.2; all of these functions moved at the same time of renaming, to `parse_type.c`, `parse_func.c` or `parse_relation.c` in release 6.3.2. A group of eleven renames in file `parse_type.c` in release 6.3.2 was interesting: their new function names followed the format "inputOutput", where "input" is the data to be processed and "Output" is the returned data; for example, `typeidTypeName` is a function that returns the type name of a given typeid, and `typenameType` is a function that returns a type structure given a type name. As no naming convention could be observed from old names of these functions, it appeared that a naming convention was newly adopted.

4.3.2 From Release 6.3.2 to 6.4.2

From release 6.3.2 (Apr 1998) to 6.4.2 (Dec 1998), again many new features were added.

We identified total 48 instances of renames in this release change. Additionally, four groups with renaming rules were observed:

1. *"mergesort" → "mergejoin"*

   Substring "mergesort" in a function name was replaced with "mergejoin". Renames conforming to this rule include:

   - `make_mergesort` → `make_mergejoin` in `createplan.c`,
   - `mergesortop` → `mergejoinop` in `initsplan.c`,
   - `cost_mergesort` → `cost_mergesort` in `costsize.c`,
   - `create_mergesort_path` → `create_mergejoin_path` in `pathnode.c`, and
   - `op_mergesortable` → `op_mergejoinable` in `lsyscache.c`.

Later, we found the evidence as well as the cause for this group of renames in the CVS log: "MergeSort was sometimes called mergejoin and was confusing. Now it is now only mergejoin",

2. *"Rel" → "RelOptInfo"*

   "Rel" in a function name was replaced with "RelOptInfo". These renames were:

   - `_readRel` → `_readRelOptInfo` in `readfuncs.c`,
   - `_copyRel` → `_copyRelOptInfo` in `copyfuncs.c`, and
   - `_outRel` → `_outRelOptInfo` in `outfuncs.c`.

   We noticed that all the above functions in old release had embedded the name of a data structure "Rel" into their names. In the new release, data structure "Rel" was renamed to "RelOptInfo". Thus we suspected the rename of data structure was the cause for this group of function renames. We later found a description in CVS log saying "Rename Rel to RelOptInfo", which confirmed our hypothesis.

3. *"JInfo" → "JoinInfo"*

   "JInfo" in a function name was replaced with "JoinInfo". Involved functions were:

   - `_equalJInfo` → `_equalJoinInfo` in `equalfuncs.c`,
   - `_readJInfo` → `_readJoinInfo` in `readfuncs.c`,
   - `_copyJInfo` → `_copyJoinInfo` in `copyfuncs.c`, and
   - `_outJInfo` → `_outJoinInfo` in `outfuncs.c`.

   We also discovered that a common substring "JInfo" in all the old function names was a name of a data structure. The data structure "JInfo" was renamed to "JoinInfo" in new release. Considering its high similarity with group "Rel" → "RelOptInfo", we believed that this group of renames was also caused by the rename of data structure.

4. *"CInfo" → "ClauseInfo"*

   "CInfo" in a function name was replaced with "ClauseInfo". Four function renames fell into this group:

- `_equalCInfo` → `_equalClauseInfo` in `equalfuncs.c`,
- `_readCInfo` → `_readClauseInfo` in `readfuncs.c`,
- `_copyCInfo` → `_copyClauseInfo` in `copyfuncs.c`, and
- `_outCInfo` → `_outClauseInfo` in `outfuncs.c`.

  Same as previous two groups, "CInfo" was a data structure name and the data structure was renamed to "ClauseInfo". We believe the cause of these renamings was the same: renaming of data structure.

In addition to above groups with renaming rules, we discovered another group of renames: in `parse_target.c`, all the first character of function names became capitalized; for example, `expandAllTables` was renamed to `ExpandAllTables`. As the old names had already conformed to a naming convention, it seems the capitalization was only to change to another naming convention.

4.3.3 From Release 6.4.2 to 6.5

According the release notes in PostgreSQL, release 6.5 (Jun 1999) "marks a major step in the development team's mastery of the source code". In this release, major features were added more easily due to the increasing size and experience of its worldwide development team.

20 function moves were detected. 17 of them were caused by three file renames: `defind.c` in `commands` subsystem was renamed to `indexcmds.c`; `joinutils.c` and `clauseinfo.c` in `optimizer` were renamed to `pathkeys.c` and `restrictinfo.c` respectively.

A total of 70 function renames were identified, among them, we observed following groups:

- *13 functions in 6 files were renamed following rule "temp" → "noname"*;

  Cause for this group of renames was that a table used to be called `temp` was renamed to `noname` in new version.

- *4 functions in 4 files were renamed following rule "HInfo" → "HashInfo"*;

  Cause for this group was that data structure "HInfo" was renamed to "HashInfo".

- *4 functions in 4 files were renamed following rule "ClauseInfo" → "RestrictInfo"*;

  Cause for this group was that data structure "ClauseInfo" was renamed to "RestrictInfo".

- *9 functions in 5 files were renamed following rule "Aggreg" → "Aggref"*;

  Cause for this group was that data structure "Aggreg" was renamed to "Aggref".

- *5 functions in 4 files were renamed following rule "JoinPath" → "NestPath"*;

  Cause for this group was: data structure "JoinPath" had an alias called "NestPath" in new version, and functions handling nested loop changed to use the alias instead.

- *4 functions in* `heap.c` *removed substring "Pg" from their names*;

  We did not find description for this group of renames in CVS log or other documentation. Our guess is that "Pg" — being part of a function name — is only to indicate that this is a function of "PostgreSQL". It was later found to be unnecessary, thus the "Pg" was eliminated.

Of all the above groups of renames, four were related to data structure renaming. It shows that problems might occur if the name of a data structure is used as part of a function name.

For merges/splits, we discovered a group of function merges in `optimizer` subsystem. Eight functions in file `geqo_eval.c` in release 6.4.2 seemed at first to be deleted. After analysis, we found that five functions had been merged with five other functions in `joinrels.c` because of their duplicated functionalities, such as `gimme_clause_joins` in `geqo_eval.c` and `find_clause_joins` in `joinrels.c` merged and became `make_rels_by_clause_joins` in `joinrels.c` in new release. For the same reason, to eliminate near duplicate functions, two functions in `geqo_paths.c` merged with two other functions.

4.3.4 From Release 6.5 to 6.5.1, 6.5.1 to 6.5.2, and 6.5.2 to 6.5.3

Release 6.5 (Jun 1999) to 6.5.1 (Jul 1999), 6.5.1 to 6.5.2 (Sep 1999) and 6.5.2 to 6.5.3 (Oct 1999) are three consecutive minor release changes. The longest release interval was

only two months. As we have shown in Table 4.2, only a small number of entities were apparently deleted or inserted before origin analysis.

Using origin analysis, we identified twelve splits and two renames all together for all the three release changes. Ten splits occurred from release 6.5 to 6.5.1. More precisely, they are ten instances of *partial clone elimination* (we will describe this pattern in Section 4.6) combined with *pipeline extraction* (described in Section 3.8.2). These splits resulted from the introduction of a standardized way of expression tree walking; this design change eliminated near-duplicate code in ten routines that visit an expression tree recursively.

We found this surprising, as we did not expect to see a major design restructuring implemented by a minor release. We also noticed that as a result of implementing this same walker mechanism, another split occurred from release 6.5.1 to 6.5.2.

4.3.5 From Release 6.5.3 to 7.0

According to release note, release 7.0 showed "continued growth of PostgreSQL". "There are more changes in 7.0 than in any previous release". As a result, a huge number of entities were apparently deleted (509) or inserted (954) before origin analysis. Also using origin analysis, we found the largest number of structural changes in our case study: 86 function moves, 114 renames and 13 splits.

The largest number of changes occurred in `utils/adt` subsystem. At first glance, in release 6.5.3:

- File `dt.c` was deleted. Among its 70 functions, 19 moved to `datetime.c`, 49 moved to `timestamp.c` and 45 renamed&moved to `timestamp.c`. The 45 renames could be grouped into two: 22 function renames following rule "timespan" → "interval" and 23 following rule "datetime" → "timestamp".

- In file `date.c`, all of its functions moved to different files; 18 moved to `nabstime.c`; 21 functions were renamed&moved to `nabstime.c`; two renaming rules were observed "interval" → "tinterval" and "timespan" →"interval".

- In file `datetime.c`, all of its functions moved to different files; 25 moved to `date.c`; other 5 renamed and moved to either `date.c` or `nabstime.c`, following renaming rule "datetime" → "timestamp".

We noticed that the data structure "TimeSpan" was renamed to "Interval" and "DateTime" was renamed to "Timestamp". Based on our similar findings in previous release changes — that is renaming of data structure may result group of function renames — we concluded that the renames following rules "timespan" →"interval" and "datetime" → "timestamp" were also caused by rename of data structures.

In release 7.0 we found:

- In `nabstime.c`, 14 functions were old; among the other 42 functions, 39 were moved from `date.c`.
- In `datetime.c`, only one was an old function; among the other 20 functions, 19 came from `dt.c`.
- In `date.c`, there was only one old function; among the other 47 functions, 28 were from `datetime.c`.
- In `timestamp.c`, only four were old functions; among the other 52 functions, 49 were from `dt.c`.

Changes at the file level can be observed:

- Old file `dt.c` split off new file `datetime.c` and new file `timestamp.c`.
- Old file `datetime.c` merged to new `date.c`.
- Old file `date.c` merged into new file `nabstime.c`.

Combining all the observations in `utils/adt` subsystems together, we concluded that a restructuring of files related to date/time occurred. In this restructuring, no new file was created, and a huge number of functions just moved between existing files, with or without renaming.

There were eight other function renames following rule "destroy" → "drop", such as `heap_destroy` was renamed to `heap_drop`. We did not find descriptions of why these renames happened from documentation. We suspected they were caused by the change of terminology.

13 splits were detected in this release change: six of them resulted from the same expression walker mechanism first occurred in release 6.5.2; other six were caused by an expression *mutator* mechanism similar to the expression walker, which supports a standardized way to modify an expression tree.

4.3.6 From Release 7.0 to 7.0.3

We did not find any structural changes in this minor release change.

4.3.7 From Release 7.0.3 to 7.1

Release 7.1 (Apr 2001) focused on "removing limitations that have existed in the PostgreSQL code for many years". Using origin analysis, we identified total 17 moves and 65 renames.

We did not find any obvious group of moves.

For renames, 24 instances that occurred in file `vacuum.c` had their prefix "vc_" removed. Other small changes were also made to the old names, such as adding a separator "_" for words in the name. Four functions in `arrayfuncs.c` removed their leading underscores.

4.3.8 From Release 7.1 to 7.1.3

From release 7.1 to 7.1.3 (Aug 2001), we found four function moves from file `nbtcompare.c`:

- `bttextcmp` moved to `varlena.c`,
- `btabstimecmp` moved to `nabstime.c`,
- `btfloat8cmp` moved to `float.c`, and
- `btfloat4cmp` moved to `float.c`.

All the moves were across subsystems. We noticed that it seemed reasonable to group these functions either as they were in old release or in new release. For example, consider the function `bttextcmp` which compares two texts: since it is a comparison function, it could be grouped into file `nbtcompare.c` as it was in release 7.1. Alternatively, since it

is a function that compares text, it is also reasonable to be grouped in file `varlena.c` that handles text, as in release 7.1.3. These moves indicate that grouping by operations (in release 6.3.2) was switching to grouping by objects (in release 6.4). This seemed to be similar with the file regrouping in `parser` in release 6.3.2 that we have discussed in Section 4.3.1, where a large number of functions were moved so that grouping based on operations was replaced with grouping based on objects.

4.3.9 From Release 7.1.3 to 7.2

Release 7.2 (Feb 2002) "improves PostgreSQL for use in high-volume applications". We identified total three moves and 42 renames. There were two groups of renames:

- Four functions in `vacuum.c` added prefix "vac_" into their names, such as `show_rusage` became `vac_show_rusage`.
- Four functions in `varbit.c` removed "zp" from their names, such as `zipbit` became `bit`.

We discovered four splits in four files in `access` subsystem. They were caused by a newly introduced callback mechanism that allows tuple processing during index building.

4.4 Summary and Analysis of Moves

Moving a function or a file is a common technique to improve software design by regrouping system functionalities. Developers may do this to increase module cohesion and decrease inter-module coupling so that software becomes easier to maintain. In our case study, we found a total of 231 instances of simple moves and also a large number of complex ones, such as moves that were combined with a rename (we categorized them only as renames when we summarized structural changes in Figure 4.2, but in this section, we include them into our analysis). We were able to infer the intention behind some groups of moves. For example from release 6.2 to 6.3.2 and release 6.5.3 to 7.0, a large number of moves were the results of restructuring. Two groups of moves were found to conform a common regrouping policy: switching from operation-based grouping to object-based grouping.

In the remainder of this section, we will describe how we performed further analysis on move instances identified, and what knowledge we can further obtain.

4.4.1 Multi-dimensional Categorization

Function moves can differ in various ways. Functions in a file can be moved out to form a new file, or to merge into another existing file, with different structures and purposes. Moves can be simple, with no other changes except the change of location, or complex, such as when the name is changed at the same time. Moves can occur across different subsystems or within a subsystem, with the former often having a larger impact on the system structure than the latter.

To examine how moves have changed the system structure, we must consider them from multiple perspectives. Thus we have created a *multi-dimensional categorization* for analyzing moves. Basically, we group and summarize function moves from multiple dimensions, with each of them focusing on one perspective. Then we mine useful information by performing "drill-down" or "roll-up" on these dimensions. We have considered four dimensions: *granularity*, *change complexity*, *locality*, and *ontology*. We note that there exists a conceptual overlap between *granularity* and *ontology*; we will discuss it after we explain all the dimensions.

In the following discussions about each dimension, we suppose we are considering a function move "$F \rightarrow G$", where function F was in file f in V_{orig}, and function G was in file g in V_{new}. Note G may have a different name with F, as function rename may occurred at the same time of move.

Dimension: Granularity

The *granularity* dimension captures information about change structure. It also reflects purpose of a move. We define four patterns by asking following two questions:

1. Did all of the functions in file f move to file g?
2. Did all the functions in file g come from file f?

Answers to above questions result in four patterns, as shown in Figure 4.3:

- *File-Move* — File f moves to file g.
- *Patch* — Part of file f moves out and becomes part of file g.
- *File-Merge* — File f merges to file g.
- *File-Split* — Part of file f moves out and becomes an individual file g.

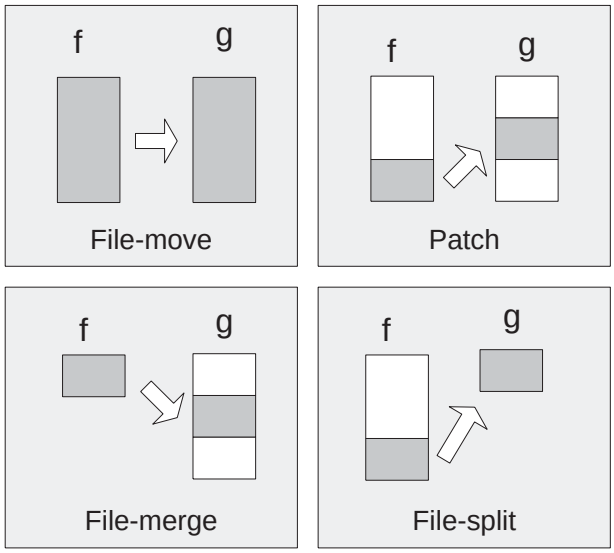


Figure 4.3: *Move patterns*

Dimension: Change Complexity

The *change complexity* dimension focuses on how complicated a function move is. We use a simple method to divide function moves into two complexity groups: if function F and G have the same name, then we say it is a "simple move"; otherwise, we say it is a "complex move". There exist other ways to measure how complex a move is, of course. For example, we will consider to relate complexity with the amount of changed lines of code in the future.

Dimension: Locality

The third dimension measures the change of location in a function move. For a move $F \rightarrow G$, if f and g are in the same subsystem, when we say the location change is small; otherwise, we say it is big. According to our definition, small location change means that function moves within a subsystem, thus it has a relatively small impact on the system structure. Big location change, on the other hand, means that move occur across subsystem boundary, thus it has a bigger impact on the system structure.

Our measurement of move locality is a simplified one. It can be improved in the future, such as in addition to distinguishing whether a move is within the same subsystem or not, we may consider the relative path distance between the subsystems before and after move as well.

Dimension: Ontology

In the dimension of *ontology*, we look at the apparent statuses of the two parent files (the two files that contained the two functions involved in the moving) before origin analysis. We answer the following two questions to characterize a move $F \rightarrow G$ along this dimension:

- Was the parent file f of F apparently deleted from V_{orig}?
- Was the parent file g of G apparently new in V_{new}?

Answers to the two questions result in four groups:

- A — Both file f and g exist in both versions.
- B — File f exists in both versions and file g appears to be added.
- C — File f appears to be deleted and file g exists in both versions.
- D — File f appears to be deleted and file g appears to be added.

If combined with dimension *granularity*, this dimension helps to identify sub-cases of move pattern. For example in pattern *patch*, comparing to the case that both f and g exist in both version, it represents a different case if f is deleted and g is inserted. Including

this dimension also enables us to relate *what appear to be* with *what actually happened*. For instance, we might be able to answer such a question: "among the function moves in apparently inserted files in V_{new}, how many of them are actually resulted from *file moves*, and how many are resulted from *file merges*?"

We note that this dimension may depend on the other dimension *granularity* in some occasions. For example, if the move pattern is *file-move*, then file f must appear to be deleted from V_{orig} and file g must appear to be inserted to V_{new}. We may consider combining these two dimensions in the future.

4.4.2 Categorizing Moves in PostgreSQL

Based on the rules of categorization we described in Section 4.4.1, we performed analysis on the move instances identified in PostgreSQL. Table 4.3 summarizes the result: Dark cells represent meaningless cases, which are caused by the dependency between dimension *granularity* and *ontology*; "diff ss" and "same ss" are values on dimension *locality*, with the first one representing that the move is across different subsystems and the second one representing that the move is within the same subsystem.

		move only				move & rename				
		A	B	C	D	A	B	C	D	SUM
File-move	same ss	0	0	0	17	0	0	0	2	19
	diff ss	0	0	0	33	0	0	0	0	33
Patch	same ss	59	25	28	17	32	5	46	0	212
	diff ss	12	0	0	15	5	0	0	8	40
File-merge	same ss	0	0	0	0	0	0	0	0	0
	diff ss	0	0	0	0	0	0	0	0	0
File-split	same ss	0	11	0	0	0	0	0	0	11
	diff ss	0	0	0	10	0	0	0	15	25
SUM		71	36	28	92	37	5	46	25	340

Table 4.3: *Categorization of moves (grey means meaningless)*

From this table we can see that along dimension *granularity*:

1. No instance was found for pattern *file-merge*.

2. Of all the four patterns, pattern *patch* had the largest number of instances — 252, among them 211 occurred within subsystem. We think this is quite reasonable, because if a function F in f can move to file g, it means F is closely related to both f and g, which is more likely to be true within a subsystem. The largest number of instances (108) in this pattern occurred in category A, meaning both f and g exist in both versions. This may imply that there used to exist quite a few problems with grouping; it may also imply that considerable efforts had been made to solve these problems. We also notice a large portion of function moves — 95 out of 252 — were combined with rename. This may be caused by adaptation that had to be made when a function moves to a new file host, such as changing its name to conform to a different naming convention in the new file.

3. For pattern *file-move*, 50 out of total 52 were simple moves, with no changes to their names. Compared to other patterns, the ratio of simple moves to complex moves in this pattern is much higher, which may imply that when a function moves due to a file move, it is less likely to perform other changes, as the requirement for the function to adapt to a new file host does not exist. We found more instances in this pattern occurred across subsystems than within subsystem, which is similar to pattern *file-split*, but different from pattern *patch*. It may suggest that comparing to fixing existing grouping mechanism (as in the case of pattern *patch*), patterns producing new modules (including both *file-move* and *file-split*) are more likely to occur across subsystems.

We now examine along the *change complexity* dimension. Except in category C, it seems that most moves are just simple moves, only a relatively small number of moves were accompanied by renames.

Along the *locality* dimension, we observed that many more instances (242) occurred within subsystem than across subsystems (98). However, in patterns *file-move* and *file-split*, the numbers of moves within subsystem are both less than that across subsystems.

Since pattern *patch* has 212 instances within subsystem, which is about as twice the sum of all the moves across subsystem in all the other patterns, in overall, moves within subsystem are still dominant.

For dimension *ontology*, we found that A and D are two largest groups. In A, all the instances are of pattern *patch*, which is actually the only possible pattern they can be. D contributes to three patterns. Comparing to A, B and C, D has a bigger percentage of instances occurred between different subsystems. This may suggest that function moves from deleted files to added files are more likely to occur across subsystems than others.

4.4.3 Comparison between Two Restructuring Instances

We also found the categorization helps to identify different types of restructuring. For example, as we discussed in Section 4.3.1 and Section 4.3.5, we have discovered two large scale restructurings: from release 6.2 to 6.3.2 and release 6.5.3 to 7.0. We categorize moves in the two cases in Table 4.4 and Table 4.5.

		move only				move & rename			
		A	B	C	D	A	B	C	D
File-move	same ss								
	diff ss				33				
Patch	same ss		15		14		3		
	diff ss				9				6
File-merge	same ss								
	diff ss								
File-split	same ss		11						
	diff ss								15

Table 4.4: *Categorization of moves from release 6.2 to release 6.3.2*

We can easily observe following differences between the two restructuring instances:

1. Table 4.4 has instances of three different patterns, while Table 4.5 only has instances of pattern *patch*.

		move only				move & rename			
		A	B	C	D	A	B	C	D
File-move	same ss								
	diff ss								
Patch	same ss	49	6	23	3	29		45	
	diff ss	5				4			
File-merge	same ss								
	diff ss								
File-split	same ss								
	diff ss								

Table 4.5: *Categorization of moves from release 6.5.3 to release 7.0*

2. Table 4.4 contains a large number of instances across different subsystems, while most instances in Table 4.5 are within a single subsystem.

3. In Table 4.4, all the instances fell in B or D, which means new files were always involved during the restructuring, while Table 4.5 only has a very small number of instances in B and D.

4. Table 4.4 has no instance of A, which means no functions switch to another file. However, in Table 4.5 A has the largest number of instances.

We can see from above discussions that the multi-dimensional categorization technique can help us examine historical function moves more closely. We are able to mine more knowledge by "drilling-down" or "rolling-up" along multiple dimensions. The dimensions we have chosen in our case study reflect four perspectives that we found useful in studying software evolution. There should exist other perspectives that are also interesting. Our categorization technique has the flexibility to incorporate more dimensions in the future.

4.5 Summary and Analysis of Renames

Renaming is a common technique that maintainers use to improve the code comprehensibility. The reasons for a rename can be very different, such as to fix a wrong word in the name, to be compliant with a naming convention, or to reflect changes in functionality. In our case study, we found total 370 instances of renames, with a large portion of them changed as groups. We were able to identify the cause of renames in some cases. For example, a large number of renames actually resulted from the rename of data structure.

We found there was no unified naming convention in PostgreSQL, which became a significant cause for renames. We observed different ways to separate words in a function name, such as capitalizing the first letter of each word, or using "_" as the word separator. There also exist different ways to distinguish the first word with others, such as lowercasing or capitalizing the first character word, adding a prefix "_", or even adding a prefix like "vc_" to denote the file or subsystem that it was contained. Lack of a naming convention in the whole development team resulted in function renames only to switch between different conventions. For example, from release 7.0.3 to 7.1, eleven functions in `vacuum.c` removed their prefix "vc_". As a member of this rename group, `vc_updstats` was renamed to `update_relstats`. Later in release 7.2, together with three other functions in `vacuum.c` being added prefix "vac_", `update_relstats` was renamed again to `vac_update_relstats`.

We notice a "ripple effect" of renames; that is, similar types of changes span different releases. For example, the leading underscore character was removed from one function in 6.4.2 to 6.5, three functions in 6.5.3 to 7.0, and five functions in 7.0.3 to 7.1.

As name serves as a part of the interface that a function exposes to others, if a function name is changed, additional changes to other functions that use this function might be necessary. Thus for those renamings that cause unnecessary confusion, we need to investigate ways to prevent them. We have summarized different situations of renames that we have discovered during our case study, since identifying causes for renames in the past can help us to avoid some of them in the future.

We have identified following situations for renames:

1. *Replace abbreviation with meaningful word* — Abbreviation in the name is hard to understand or remember. It is replaced with a meaningful word. Example: `genxprod`

→ `gen_cross_product`.

2. *Replace a long name with a short one* — A long name describing many details is replaced with a short summary-like one. Example: `DoChdirAndInitDatabaseNameAndPath` → `VerifySystemDatabase`.

3. *Fix error* — Fix a wrong name caused by misspelling, incomplete or overly description. Example: `estimate_disbursion` → `estimate_dispersion`.

4. *Conform to a naming convention* — Create a naming convention for entities, such as separating words using "_" or lowercase leading character. Example: `qualcleanup` → `qual_cleanup`.

   A naming convention can be more than lexical formatting. For example from release 6.2 to 6.3.2, we found 11 function names were changed to conform to format "inputOutput", where "input" is the data to be processed and "Output" is the returned data.

5. *Change to a different naming convention* — Different developers may adopt different naming conventions. When a different developer becomes the maintainer, (s)he may change the name so that it conforms to another naming convention. It may also happen when a function moves to another file that uses a different naming convention. Example: `makeParseState` → `make_parsestate`.

6. *Result from data structure change* — Data structure name is embedded in function name. When the data structure is renamed, the function is renamed too. We found a huge number of instances fallen into this category. Example: `_copyHInfo` → `copyHashInfo`.

7. *Result from functionality changes* — Function changes what it does, thus the name changes too to reflect increased or decreased functionality.

Multiple cases can occur at the same time for one rename instance. For example, a name is changed to follow a new naming convention; while at the same time, abbreviation in the name is replaced with a full word.

In summary, we found function renamings occurred often in the evolution of PostgreSQL. Different situations caused these renamings. Some of them could be avoided easily, *e.g.*, in the case of fixing spelling error, spelling checking on function names can be performed. Some require better management in the development team, *e.g.*, in the case of 4 and 5, a unified naming convention for the whole project might be helpful to avoid them. Some do not have straightforward solutions yet, such as function renaming resulted from data structure renaming. We feel that problems still exist as how to construct a good name for a software entity.

4.6 Summary and Analysis of Merging/splitting

Merges/splits reflect more complex structural changes than moves or renames. The number of total instances we have identified in our case study is much less than other types of changes. However, we found that we learned interesting and important information about the system's evolution by studying them. For example, we notice a "ripple effect" of merges/splits: ten splits from 6.5 to 6.5.1 were caused by introduction of a standardized way of expression tree walking; from release 6.5.1 to 6.5.2, one more split occurred for the same reason; from release 6.5.3 to 7.0, six more splits were detected due to the same walker mechanism. We have observed similar "ripple effect" in renames, as discussed in Section 4.5.

4.6.1 Two More Patterns

In addition to the merge/split cases patterns described in Section 3.8.2, we discovered two more patterns in the course of our case study:

1. *Parameterization* — Two similar functions F_1 and F_2 are combined into a new function G by adding a parameter to distinguish different functionalities.

   A strong indicator of this phenomenon is

   - $in_1 \cup in_2 \approx in$
   - $out_1 \approx out_2 \approx out$

- $dec1_1 \approx decl_2 \wedge decl_1 \cup decl_2 \cup param_{new} \approx decl$

  where $decl_1$, $decl_2$ and $decl$ are function declarations for F_1, F_2 and G respectively, and $param_{new}$ is the new parameter in $decl$ and "$\approx$" denotes lexical similarity.

2. *Partial clone elimination* — A chunk of code found in two functions F_1 and F_2 are clones. These clones are extracted out to form a new function G, which is called by its parent functions F_1 and F_2.

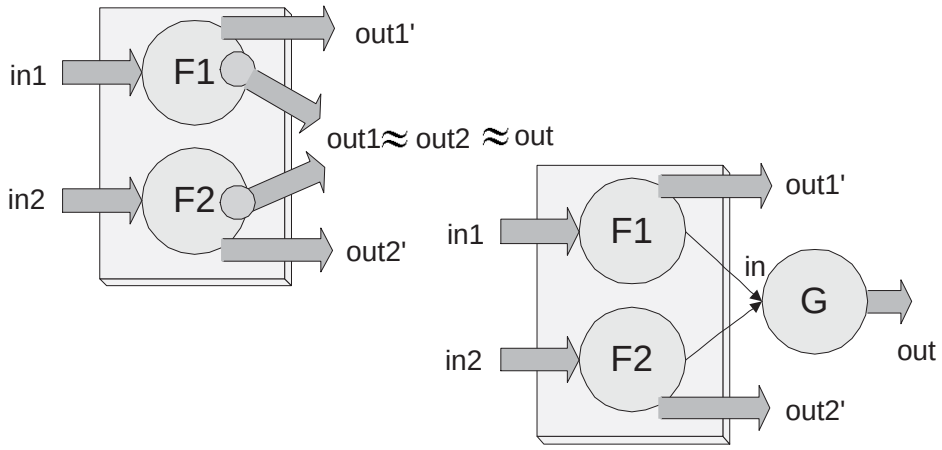


Figure 4.4: *Partial clone elimination*

A strong indicator of this phenomenon is

- $in_1 \cap in_2 \approx \emptyset$
- $out_1 \approx out_2 \approx out$
- $in = (F_1, F_2)$

where out_1, out_2 and $decl$ are the callee sets of the common clone segment within F_1 and F_2.

4.6.2 Combination of Patterns

As is common with the application of design patterns [18], we found that multiple patterns of merging/splitting may be applied on the same entities at the same time. For example, the creation of a standardized expression tree walker mechanism mentioned above involved a combination of *partial clone elimination* and *pipeline extraction* (although we counted it only as one instance in Figure 4.2). In this combination, *partial clone elimination* was first applied on functions that share common code for visiting an expression tree, which resulted the creation of a new function called `expression_tree_walker`. Then, *pipeline extraction* was further applied on the "parent" functions (where the clones had just been removed from): each of the parents split off the logic that detailed the peculiar way it walked the tree into a new adapter [18] function, call it `my_walker`, which the new, slimmed-down parent became the sole client of.

When different merge/split patterns are applied at the same time, the change of call relations can be complex and hard to reason about. Our current approach favors flexibility and querying over automated pattern detection; we intend to add more automated support for pattern detection in the future.

4.6.3 Instances of Different Patterns

In Table 4.6, we list the number of instances we found for each merge/split pattern as well as some examples.

We were surprised to find only one instance of *service consolidation* in our case study. A possible reason for this is that situations for this change to occur are relatively rare and developers may not wish to merge different services after-the-fact if they are not quite sure that these services should be combined into one. Patterns that relate to removing code duplicates, including *clone elimination*, *parameterization*, and *partial clone elimination* have a relatively large number of instances. This indicates that much effort had been put to eliminate duplicate codes, routines, and idioms in PostgreSQL. It also suggests that clones are good starting points for merge/split detection, and clone detection, although it is different from origin analysis, can help to improve techniques in origin analysis.

When we considered these instances as a group, we found that the names of the en-

Pattern(#)	Examples	Releases
clone elimination (7)	getAttrName,get_attname→ get_attname	6.2 → 6.3.2
service consolidation (1)	gettypelem,typtoout → getTypeOutAndElem	6.4.2 → 6.5
pipeline extraction (6+*23*)	appendStringInfo → appendStringInfo, enlargeStringInfo	6.4.2 → 6.5
parameterization (3)	RelationSetLockForRead, RelationSetLockForWrite → LockRelation	6.4.2 → 6.5
partial clone elimination (27)	_finalize_primnode,fix_opid,... → expression_tree_walker	6.4.2 → 6.5

Table 4.6: Merge/split instances of different patterns

tities themselves often gave out information about the type of the change, such as when `gettypelem` and `typtoout` were merged to `getTypeOutAndElem`, and `appendStringInfo` split off `enlargeStringInfo`. This indicates that name is a valuable source of information in merge/split detection.

4.6.4 Group of Merges/splits

Three groups of splits were detected:

1. 17 splits in ten files from four subsystems were caused by the introduction of the walker mechanism mentioned above,
2. six splits in four files from two subsystems were caused by a mutator mechanism that supports a standard way to modify an expression tree, and
3. four splits in four files in subsystem access were caused by a callback mechanism that allows tuple processing during index building.

All three groups were caused by the introduction of a new mechanism. We wondered how a group of changes scattered in different subsystems spanning multiple releases could

be achieved. After we examined the CVS log of PostgreSQL, we found that all these changes had the same author. This reminded us the fact that PostgreSQL has a core development team, which enables the common author of a large number of files to restructure modules relatively easily without worrying about "breaking" what other developers were doing. It would be interesting to investigate whether the group change phenomena is different in other OSS projects without a core development team.

4.6.5 Merges/splits at the File Level

We found two groups of merges/splits at the file level. The first one corresponded to a large-scale restructuring in the parser subsystem from release 6.2 to 6.3.2. Functions in the "old" files were redistributed throughout the subsystem; some were placed in existing files, while others were grouped into "new" files. The functions themselves were not merged or split, but left intact. For example, functions in `analyze.c` were moved into seven files and a new file `parse_agg.c` in release 6.3.2 was formed from function `agg_error` in `catalog_utils.c`, four functions in `analyze.c` and `ParseAgg` in `parse.c`. The second group resulted from the cleaning up of `optimizer` subsystem from release 6.4.2 to 6.5; most functions in file `geqo_eval.c` and `geqo_paths.c` in `optimizer_geqo` subsystem merged to subsystem `optimizer_path`.

We can see that total number of merges/splits at file level and the frequency appears to be much smaller than that at function level. They happened only during major releases changes, which is reasonable because structural changes at file level usually reflect a big change to overall design, which is typically implemented only during major release changes.

4.7 Summary

In this chapter, we describe our case study of structural changes in the evolution process of PostgreSQL. We have performed origin analysis on 12 releases of PostgreSQL over four years, including six major release changes and five minor release changes. In overall, we detected a large number of structural changes (679). We found in major release changes, on average, at least one third of the functions that seemed to be deleted from old version and 22% that seemed to be inserted into new version, were actually performing structural

changes, including renaming, moving, merging and splitting. The structural changes that actually existed might be much more than what we have detected, thus it seems that more research on studying structural changes is necessary.

Based on the changes we have identified, we observed many interesting phenomena: We found groups of changes occurred during restructuring; function merging and splitting exhibited patterns, and similar to design patterns, they can be used in combination at the same time; a set of changes caused by the same reason were scattered in multiple release changes.

We also categorized function moves and mined more knowledge out of historical data. We found 74% function moves occurred between existing files, and 71% occurred within subsystems. We also summarized different causes for function renames. We found many renames were caused by lack of unified naming convention. Many resulted from data structure renaming.

In the next chapter, we conclude this thesis and discuss future work.

Chapter 5

Summary and Future Work

5.1 Summary

In this thesis, we present our approach to achieve an improved understanding of structural changes during software evolution. We propose an improved version of origin analysis, as well as its tool support in Beagle, to detect structural changes, including function/file move, rename, merging and splitting. In our approach, we generalize the notion of matching and adopt a flexible process, so that the detection can be guided by the user, and new matching techniques can be incorporated as plug-ins. Compared to the initial version of origin analysis, we are able to detect merging and splitting. We can also deal with changes in more complex situations, such as when they are chained together. To better understand structural changes, we use a technique called scatter plot to visualize changes between two versions. Scatter plot is also useful in detection of changes. The tool Beagle supports origin analysis in various ways, including navigation between entities involved in changes, transaction support for change identification and building a persistent repository of changes identified.

In our case study of 12 releases of PostgreSQL over four years, we detected a large number of structural changes. We found that in six major release changes, on average, at least one third of the functions that seemed to be deleted from the old version and 22% that seemed to be inserted into the new version, were actually caused by structural changes. We also observed many interesting phenomena, such as a set of similar changes

spanning multiple releases, group of function renamings caused by data structure renaming, and patterns of merging/splitting. After categorizing function moves we have identified, we found that most function moves were between existing files, and that most functions moved within subsystems.

We believe our approach is promising towards a better understanding of software structural changes.

5.2 Future Work

One possible improvement to our approach is to investigate new sources of evidence. In our current approach, the main evidence for change identification is drawn from source code. We use only CVS log as an information source for verification. However, as the CVS log records detailed historical data, and programmers often commit structural changes as one operation, including when they perform renaming, moving, merging and splitting, it would be interesting to investigate how the CVS log can act as a new source of evidence.

Another future project is to encode heuristics and build them into tool support. Heuristics about change identification and change patterns are only "formless" information derived from experiences in our current approach. As it is expensive to derive a good heuristic, it is valuable to encode and add automation for it in tool support, so that this knowledge can be reused to speed up analysis process. For example, patterns of merging/splitting are detected manually in our current approach. Automation can be added if we can formally expressing constraints that define this pattern.

There are some other parts of our approach that need to be improved in the future, including incorporating type-based comparison into `declaration-matcher`, and optimization of `metrics-matcher` *etc.*.

We plan to perform more case studies to evaluate and provide feedback to improve our current approach. We would like to know whether other software systems share the same behavior with PostgreSQL. If there exist differences, then how to explain them.

Bibliography

[1] http://www.dur.ac.uk/CSM/.

[2] Feast feedback, evolution and software technology, 1999. http://www-dse.doc.ic.ac.uk/~mml/feast/.

[3] Graphviz. http://www.research.att.com/sw/tools/graphviz/.

[4] Grok. http://swag.uwaterloo.ca/swagkit/.

[5] Jdbc. http://java.sun.com/products/jdbc/.

[6] Open source initiative. http://www.opensource.org/docs/definition.php.

[7] Postgresql. http://www.postgresql.org/.

[8] Swagkit. http://swag.uwaterloo.ca/swagkit/.

[9] Understanding for c++. http://www.scitools.com/downloadc.shtml.

[10] Annie I. Antón and Colin Potts. Functional paleontology: System evolution as the user sees it. In *Proceedings of the International Conference on Software Engineering*, 2001.

[11] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of Second Working Conference on Reverse Engineering*, pages 86–95, 1995.

[12] Evelyn J. Barry, Chris. F. Kemerer, and Sandra. A. Slaughter. On the uniformity of software evolution patterns. In *Proceedings of the International Conference on Software Engineering*, 2003.

[13] Ira D. Baxter, Andrew Yahin, Leonardo M. De Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, pages 368–377, 1998.

[14] Elizabeth Burd and Malcolm Munro. An initial approach towards measuring and characterising software evolution. In *Proceedings of the International Conference on Reverse Engineering*, 1999.

[15] James Westland Cain and Rachel Jane McCrindle. Making movies: Watching software evolve through visualisation. *Lecture Notes in Computer Science*, 2074:738–??, 2001.

[16] Harald Gall, Mehdi Jazayeri, Rene Kloesch, and Georg Trausmuth. Software evolution obervations based on product release history. In *Proceedings of the International Conference on Software Engineering*, 1997.

[17] Harald Gall, Mehdi Jazayeri, and Claudio Riva. Visualizing software release histories: the use of color and third dimension. In *Proceedings of the International Conference on Software Maintenance*, pages 99–108, 1999.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1994.

[19] Michael W. Godfrey and Qiang Tu. Evolution in open source software: a case study. In *Proceedings of the International Conference on Software Maintenance*, 2000.

[20] Michael W. Godfrey and Qiang Tu. Tracking structural evolution using origin analysis. In *Proceedings of the International Workshop on the Principles of Software Evolution*, 2002.

[21] R. Holt and J.Y. Pak. Gase: visualizing software evolution-in-the-large. In *Proceedings of Second Working Conference on Reverse Engineering*, pages 08–10, 1996.

[22] Bell Canada Inc. Datrix - abstract semantic graph reference manual version 1.2. 1999.

[23] J. H. Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the International Conference on Software Maintenance*, pages 120–126, 1994.

[24] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[25] Chris F. Kemerer and Sandra Slaughter. An empirical approach to studying software evolution. In *Proceedings of the International Conference on Software Engineering*, pages 493–509, 1999.

[26] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Proceedings of Fourth Working Conference on Reverse Engineering*, pages 44–54, 1997.

[27] Michele Lanza. The evolution matrix: recovering software evolution using software visualization techniques. In *Proceedings of the International Workshop on the Principles of Software Evolution*, pages 37–42, 2001.

[28] M M Lehman, D E Perry, and J F Ramil. Implications of evolution metrics on software maintainance. In *Proceedings of IEEE Special Issue on Software Maintainance*, pages 208–217, 1998.

[29] M M Lehman, J F Ramil, P D Wernick, and D E Perry. Metrics and laws of software evolution - the nineties view. In *4th International Software Metrics Symposium*, 1997.

[30] M. M. Lehman and Juan F. Ramil. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 11(1):15–44, 2001.

[31] Lehman M. M. Programs, life cycles, and laws of software evolution. In *Proceedings of IEEE Special Issue on Software Engineering*, pages 1060–1076, 1980.

[32] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. Evolution patterns of open-source software systems and communities. In *Proceedings of the International Workshop on the Principles of Software Evolution*, 2002.

[33] David Lorge Parnas. Software aging. In *Proceedings of the Conference on The Future of Software Engineering*, 1994.

[34] Dewayne E. Perry. Dimensions of software evolution. In *Proceedings of the International Conference on Software Maintenance*, pages 296–303, 1994.

[35] Juan F. Ramil and Meir M. Lehman. Metrics of software evolution as effort predictors - a case study. In *Proceedings of the International Conference on Software Maintenance*, 2000.

[36] Qiang Tu and Michael W. Godfrey. An integrated approach for studying software architecture evolution. In *Proceedings of the International Workshop on Program Comprehension*, 2002.

[37] W. M. Turski. Reference model for smooth growth of software systems. In *Proceedings of the International Conference on Software Engineering*, pages 599–600, 1996.

[38] Yunwen Ye and Kouichi Kishida. Toward an understanding of the motivation of open source software developers. In *Proceedings of the International Conference on Software Engineering*, 2003.

[39] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146–170, 1995.