Detecting Merging and Splitting using Origin Analysis

Lijie Zou and Michael W. Godfrey
Software Architecture Group (SWAG)
School of Computer Science, University of Waterloo
{lzou, migod}@uwaterloo.ca

Abstract

Merging and splitting source code artifacts is a common activity during the lifespan of a software system; as developers rethink the essential structure of a system or plan for a new evolutionary direction, so must they be able to reorganize the design artifacts at various abstraction levels as seems appropriate. However, 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. In this paper, we discuss how we have extended origin analysis *[10, 5] to aid in the detection of merging and splitting of files and functions in procedural code; in particular, we show how reasoning about how call relationships have changed can aid a developer in locating where merges and splits have occurred, thereby helping to recover information about the intent of the design change. We also describe a case study of these techniques (as implemented in the Beagle tool) using the PostgreSQL database as the candidate system.*

1 Introduction

Merging and splitting source code artifacts — such as files and functions — are commonly performed activities during both active development and maintenance. These refactoring techniques [3, 8] can be used to keep the codebase in a healthy and agile state; software maintainers often use merging and splitting to reduce the complexity of the software system, making it more comprehensible and easier to evolve.

Although the effects of merging and splitting are plainly evident in the source code version histories, the merging and splitting actions themselves typically are not. That is, it may be clear what is contained in successive versions of a set of files, but it may not be clear that between versions 4.2 and 4.3 one function from each of the files `scsi.c`, `atapi.c`, and `usb.c` were merged into a common utility function that was added to the file `storage.c`.

Detecting where merges and splits have occurred can help software maintainers to better understand the change history of a software system. In a typical development environment, system changes are tracked by a version management system, and detail which characters in which files have changed since the last check-in. They usually do not provide answers to such questions as *"why was this new function added?"*, *"where did the XXX functionality disappear to?, "how much additive versus invasive change occurred?"*, or *"how much restructuring of source code occurred?"*. If a software developer requires answers to these questions, (s)he must either hope that previous developers have kept accurate and up-to-date documentation, or (s)he must make use of tools that can help to extract information about the system's evolution after-the-fact.

In this paper, we propose an approach to detecting function and file merges and splits that may have occurred between versions of a software system. Our approach is based on a detailed analysis of call relations and various attributes of the function entities themselves. This work is an extension of our previous work on *origin analysis* [5, 10]; our original formulation of origin analysis did not consider the possibility that program entities might be merged or split between versions.

The remainder of this paper is structured as follows: In Section 2, we define what we mean by *origin analysis*, and show how we have extended the definition to include the detection of merging and splitting of source code artifacts. In Section 3, we describe how we have implemented our techniques in the Beagle tool, and we describe a case study performed on the source code for PostgreSQL, an open source database system that is in wide use. In Section 4, we discuss related work, and finally, in Section 5 we summarize our results and sketch future research.

2 Origin analysis and merges/splits

2.1 Definition of origin analysis

Let us begin with an informal definition of origin analysis:

> *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*

1

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.

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. This is why we consider that origin analysis must be a semi-automated approach to be useful. The user must apply experience and common sense to decide if the similarity is strong enough to consider that *G is a changed version of F*.

While this informal definition helps to show the intuition behind our research, *origin analysis* — as we have implemented and investigated it — 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?"* Our original implementation of origin analysis in the Beagle tool considered only the first question, but the new version of the tool supports 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).

We shall concentrate our discussions on the phenomena of merging and splitting in this paper.

2.2 Implementing origin analysis

Previously, we have described how we used a combination of *entity analysis* and *relationship analysis* techniques to help to detect structural changes that have occurred between versions of a software system [5]. In brief,

- [Entity analysis] we created a kind of hash value or fingerprint of each function in a (procedural) system based on its various attributes, such as number of lines of code, fan-in/out, number of local/global variable used, its cyclomatic complexity, *etc.*, and

- [Relationship analysis] we considered the set of callers and callees of each function.

In our original implementation of origin analysis in the Beagle tool [10], the analysis routines were performed all at once in a batch, and the results combined into a single list. That is, for each function G that appeared to be "new" in V_{new}, and for each function F that appeared to be "deleted" in V_{old}, we:

1. compared the entity analysis fingerprints,

2. compared the *calls* and *called_by* relational images, and

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

The results were then combined into a linear ranking, and the user was able to examine the best "match candidates".

Recently, we have sought to improve the design of Beagle by making it more flexible and more interactive. We have generalized the notion of matching, we have devised a plug-in infrastructure that allows for the easy addition of new "matcher" subtools, and we have provided support for building incremental persistent models of a system's change history. However, we will not discuss that work here as it is detailed elsewhere [12]. Instead, we will discuss how detection of merges and splits has been incorporated into Beagle.

2.3 Merge/split matching

Merging and splitting can occur at different architectural granularities. When these actions are performed 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 merging and splitting 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 inferred from changes at the lower levels, we have concentrated our efforts on extracting and modelling information about merges and splits at the function level.

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 $N = 2$ and consider only merging of functions for now.
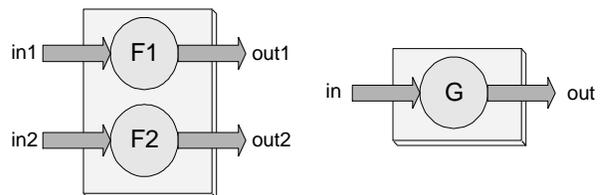

Figure 1: *Canonical two-way function merge.*

Figure 1 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 *caller*s (clients) and out_1, out_2, and out denote the *callee*s of F_1, F_2, and G respectively.

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.
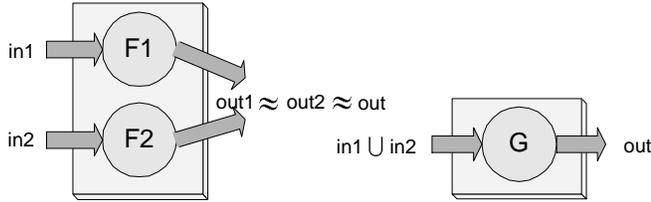

Figure 2: *Clone elimination.*

A strong indicator of this phenomenon is

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

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.

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.
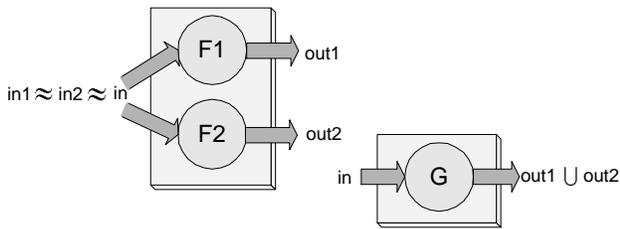

Figure 3: *Service consolidation.*

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 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 out_1 and out_2.

3. *Pipeline contraction* — A function (the service provider) is only ever called by a single client. In the new version, either the client consumes functionality of the service provider directly, or a new function is created that merges both the client and service provider.
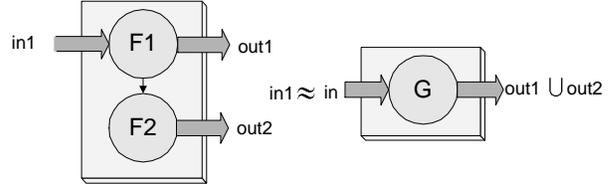

Figure 4: *Pipeline contraction.*

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 "second" 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*

may also be detected easily. We have built some of this knowledge into the new version of the Beagle tool; in the next section we will describe our experiences in using Beagle to look for merges and splits in a large software system.

We conclude this section by noting that merging and splitting may occur for a number of other reasons, in addition to those cases we have described above. Those patterns listed above were the ones we had conceived of before attempting the case study on PostgreSQL; in the next section, we shall discuss more patterns that we observed to occur in the evolution of PostgreSQL.

3 Case study: PostgreSQL

3.1 Using Beagle

Beagle is a research tool that is intended to help developers gain an understanding of a software system's evolutionary history [10, 12]. Beagle incorporates various techniques and subtools from software metrics, software visualization, and relational databases into a unified framework; this framework allows users to query, visualize, and navigate through a

system's change history, and allows users to build persistent, annotated models of how structural changes have impacted the design of the system. Details of its various functionalities are described elsewhere [10, 12]. In this paper, we concentrate on how origin analysis in Beagle can be used to detect merges and splits of functions and files.

3.1.1 Detecting basic merges/splits at function level

As noted above, the purpose of origin analysis is to decide if a software entity that appears to be either new (in the new version of the system) or deleted (from the old version) really *is* new or deleted, or if it should more accurately be viewed as having come from or disappeared into some other software entity.

For simplicity of explanation, we will assume the user is considering an "apparently new" entity, which we will call the target. The basic iterative process for performing origin analysis is straightforward:

1. the user decides on a candidate entity set of interest from the old version,

2. the user applies one or more "matchers" to the target and the candidates, and

3. the user examines the ranked list and the detailed matcher output and decides which, if any, of the candidate matches to accept as the "correct" origin of the target entity.

Step 1 helps to reduce the computation time by allowing the user to decrease the number of entities used in performing the matching algorithms; this is particularly useful in cases where we know (or hypothesize) that changes must have happened within the `parser` subsystem, or that only "deleted" entities could be involved.

In Step 2, the "matchers" are applied to produce a ranked list of the "best" matches together with details of the matcher output and hyperlinks to the appropriate source code locations so that the user can browse and compare. The matchers are plug-ins to the Beagle tool; currently, there are four matchers for functions: name, declaration, implementation, and call relations. The last one, called *relation_matcher*, is the most useful for detecting merges and splits.

The *relation_matcher* computes the similarity of two functions by comparing three sets of call relations: both callers and callees, callers only, and callees only. 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 as the patterns discussed in Section 2.3 illustrate. 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.

The candidates are ranked by computed similarity, and are associated with detailed information of how each similarity is computed. Figure 5 shows an example of candidates produced by *relation_matcher*. Here, we were trying to find the origin of apparently deleted functions in `heap.c` in PostgreSQL in release 5.0. For function `DeleteTypeTuple` (highlighted on the left), two functions in release 6.4.2 on the right were found to be very similar to it. The closest match was `DeletePgTypeTuple` and why its similarity was 1.0 was displayed at the bottom: their call relations were exactly the same.
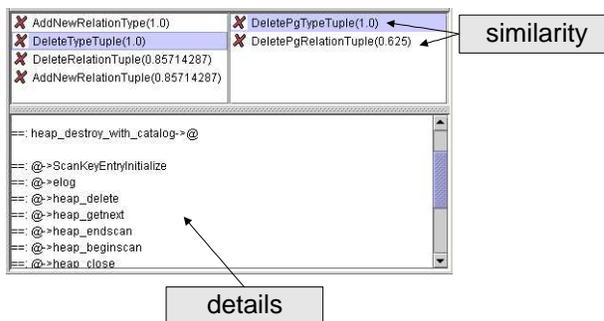

Figure 5: *Candidate list produced by relation_matcher*

In Step 3, the user decides if (s)he thinks that the current evidence is strong enough to make a commitment. The whole process may be repeated using different candidate sets and/or different matchers until the user is satisfied that the "correct" origin of the target entity has been found or, alternatively, that no such entity exists. In either case, once the decision has been arrived at, it is recorded as an attribute of the target entity and the matching candidate(s) (if any) in the Beagle model of the system.

After several iterations of origin analysis, 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. A small helper tool for comparing call relations of two sets of functions helps us in this phase.

3.1.2 Detecting chained merges/splits at function level

Sometimes we discover that a set of structural changes, including merges and splits, may be "chained" together; that is, the entities involved are heavily interdependent, making it difficult to perform our typical analysis in one iteration. In such a case, performing the analysis iteratively helps to reveal what has occurred.

Here, we present a detailed example taken from our case study of PostgreSQL from release 6.4.2 to 6.5. At first, it appeared that three functions in `geqo_eval.c` within the `optimizer_geqo` subsystem had been deleted. After performing origin analysis, we found that these func-

tions 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 6 and 7 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.
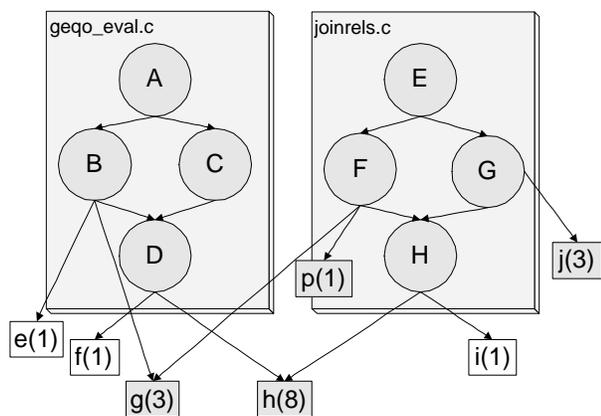

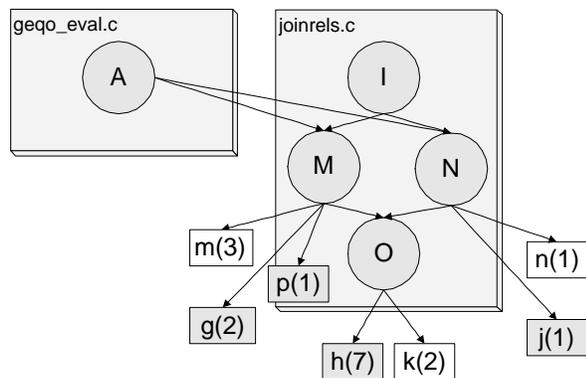Figure 6: *Call relations in release 6.4.2*

Figure 7: *Call relations in release 6.5*

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.

3.1.3 Detecting merges/splits at the file level

File level merges/splits are detected manually in Beagle. If a new file *G* is found to be composed of functions from two old files F_1 and F_2, then we consider that files F_1 and F_2 have been merged into *G* (a similar statement holds for splits at the file level). Once detected, the user enters this information as attributes of the files in question into the model of the system version in the Beagle repository.

3.1.4 Visualization

Beagle supports a variety of visualization tools for browsing the evolutionary history of a software system [12]. Among these is a scatter plot viewer, as shown in Figure 8. Scatter plots are well known in clone detection research [2, 6, 9]; 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 coloured marks are used to indicate the presence of an "interesting property" (or "hit"), usually that there is a non-trivial 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.

In origin analysis, we typically put two different versions of a system along the X and Y axes. Of course, we expect to see a high degree of similarity along the diagonal, but we also expect to see breaks, where functions have been changed, added, or deleted between versions.

Scatter plots can be used in a variety of ways: a "hit" can indicate that the computed fingerprints of two entities are within a given tolerance, or that their caller sets (or callee sets, or both) are highly similar, or that some other interesting relationship holds between the two entities. We have used scatter plots to look for meta-properties and recurring patterns across the system. Figure 8 shows an example from
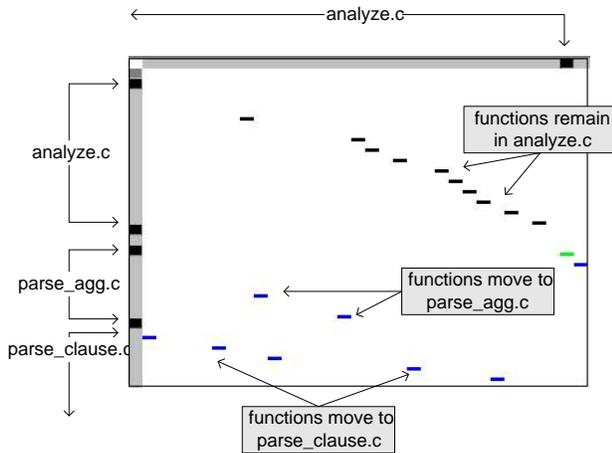
Figure 8: *Scatter plot showing function movement between versions.*

Figure 9: *Growth of PostgreSQL*

the case study of how using a scatter plots can quickly highlight when functions have been moved between files; this is particularly useful for finding file merges and splits. In our case study, we found that looking at a scatter plot after some origin analysis had been performed helped to find further incidents of function merging and splitting, and was also very helpful in detecting merging and splitting at the file level. Being able to see various relationships between many pairs of functions at the same time was an invaluable tool in exploring the evolutionary history of PostgreSQL.

3.2 Applying original analysis to PostgreSQL

PostgreSQL is an open source object-relational database management system (ORDBMS), originally based on the POSTGRES system which was developed at the University of California 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 starting 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
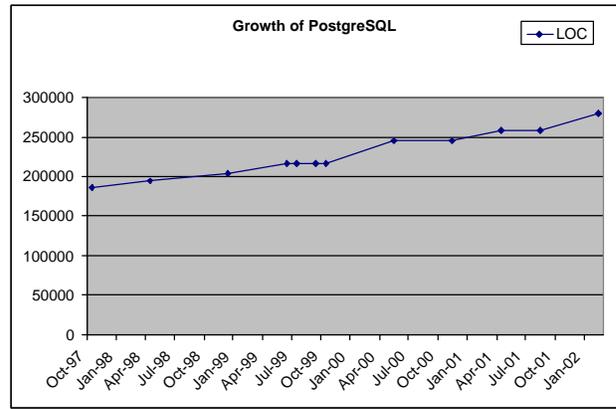
- from 186 KLOC to 279 KLOC,

- from 328 to 388 files, and

- from 3262 to 4531 functions.

Figure 9 shows the LOC (lines of code) of the `backend` subsystem in of the each releases in the study. We can see from this figure that PostgreSQL evolves continually in the four years with an annual growth rate about 10%.

3.2.1 Summary of structural changes

We performed origin analysis on each consecutive pair of the 12 releases, including six major releases (*e.g.,* 6.4.2 to 6.5) and five minor releases (*e.g.,* 7.0 to 7.0.3). We have summarized the number of each type of structural change in each release in Figure 10. Perhaps unsurprisingly, the total number of structural changes that were found to have occurred in the six major releases is much greater than those of five minor releases. Although *move* and *rename* had the largest numbers of instances in overall, merges/splits occurred in seven release changes. The four largest number of merges/splits all occurred in major releases and there was no merge/split in three minor releases. These observations conformed to our expectation that most structural changes occur during major release changes.

From minor release change 6.5 to 6.5.1, however, we found ten splits or, more precisely, ten instances of *partial clone elimination* (we describe this pattern in section 3.2.2) combined with *pipeline extraction*. These splits resulted from the introduction of a standardized way of expression tree walking; this design change eliminated near-duplicate code in many 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 and six more splits occurred in release 6.5.3 to 7.0. This was not the only case we observed of a "ripple effect" where a series of related structural changes spanned multiple
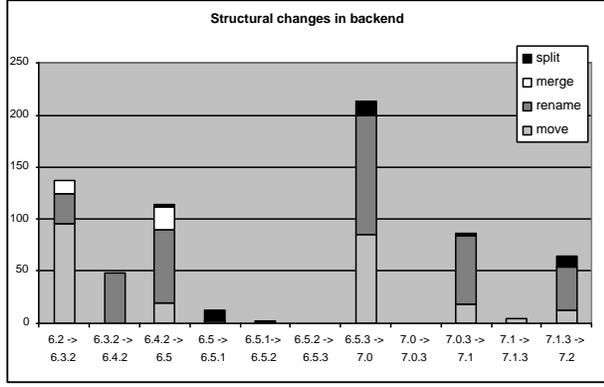
6

Figure 10: *Structural changes detected by origin analysis*

Figure 11: *Partial clone elimination*

releases. We had a similar observation for a series of function renamings, where 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.

3.2.2 Two more patterns

In addition to the merge/split cases patterns described in Section 2.3, we discovered two more patterns in the course of our case study that we had not anticipated:

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

   - $in_1 \cup in_2 \approx in$
   - $out_1 \approx out_2 \approx out$
   - $dec1_1 \sim decl_2 \quad \wedge \quad decl_1 + param_{new} \sim decl$

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

   A strong indicator of this phenomenon is

   - $in_1 \cap in_2 \approx \varnothing$
   - $out_1 \approx out_2 \approx out$
   - $in = \{F_1, F_2\}$

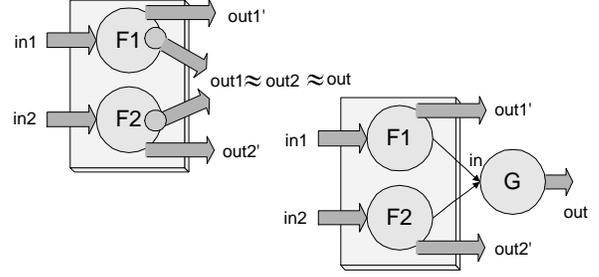   where out_1 and out_2 are the callee sets of the common clone segment within F_1 and F_2.

3.2.3 Combination of patterns

As is common with the application of design patterns [4], 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 10). 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 [4] 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 favours flexibility and querying over automated pattern detection; we intend to add more automated support for pattern detection in the future.

3.2.4 Instances of different patterns

In Table 1, we list the total 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 be hesitant to merge different services after-the-fact if they are unsure 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.

Pattern(#)	Examples	Version
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 1: Pattern instances

When we considered these instances as a group, we found that the names of the entities 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 entity names are a valuable source of information in merge/split detection.

3.2.5 Group of merges/splits

Three groups of splits were detected:

1. 17 splits in ten files from four subsystems caused by the implementation of the walker mechanism mentioned above,

2. six splits in four files from two subsystems 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` caused by a callback mechanism that allows tuple processing during index building.

All three groups were caused by introduction of a new mechanism. We wondered how a group of changes scattered in different subsystems spanning multiple releases could be performed. 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. We intend to investigate this in the future.

3.2.6 Merges/splits at file level

We also found two groups of merges/splits at the file level. The first one corresponded to a large-scale re-structuring 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.

3.2.7 Summary: Merging and splitting in PostgreSQL

In summary, we found that merging and splitting accounted for a non-trivial number of structural changes — both at the function level and at the file level — in the evolution of PostgreSQL. While detecting where merges and splits had occurred required time and effort, it improved our understanding of how and why some of the major design changes had been made to the system.

We note that while we are fairly confident that we have no false positives in our findings, we may have missed some merges and splits also. That is, merging and splitting may be even more widespread than we have found so far.

4 Related work

4.1 Fowler's refactoring catalogue

Refactoring is a commonly performed preventive maintenance activity; its intent is to improve some aspects of the design of an existing system while leaving the outward functionality of the system mostly unchanged.[1] Refactoring can be performed at various levels of detail: within and between functions, classes, packages/subsystems, and up to the architectural level; however, as used in Fowler's widely read book, the term mostly concerns function- and class-level design changes [3]. It presents a catalogue of "bad smells"[2] to look for in code, as well as a set of appropriate actions (refactorings) to take in each case.

---

[1] While Opdyke was the first to use the term in its present sense [8], Fowler's book is the best known distillation of this body of knowledge [3].

[2] van Emden and Moonen have built a tool called jCosmo to aid in the automated detection of "bad smells" [11].

While Fowler's book is aimed mainly at object-oriented systems, many of the refactoring patterns listed in Fowler's book are of interest to our work in origin analysis, as they involve moving, renaming, merging, and/or splitting methods/functions. These include: push down/pull up/move method; hide method; form template method; extract/inline method; rename method; replace method with method object; and parameterize method.

We note that Fowler's catalogue is aimed at the (intentional) design level; his patterns are at the level of *what-would-a-developer-be-thinking*. Our patterns are more low level, and correspond more closely to *what-would-a-developer-do-to-the-code*.

4.2 Clone detection

Origin analysis borrows some techniques from software clone detection [1, 7]; however, the aim of origin analysis is distinct from that of clone detection, and some of the details and tradeoffs are different. Clone detection, *per se*, is usually performed on a single version of a software system to see if any two (or more) components strongly resemble one another. The goal is to detect where cloning may have occurred in the past, with a view to possibly reorganizing the source code and refactoring the commonalities into a single place within the system. Origin analysis is performed across two versions of a system, and the goal is to build a model of where, how, and why structural changes have occurred.

5 Summary

Merging and splitting source code artifacts are common activities during development and maintenance, yet it is common for the rationale of these design changes not to be recorded. In this paper, we have discussed a set of techniques for applying origin analysis to detect instances of merging and splitting in source code. We presented a set of merge/split patterns, and showed how reasoning about call relationships can aid in detecting their occurrence. Finally, we performed a case study on the PostgreSQL system; we found that merging and splitting of functions and files had occurred throughout its history, and that the techniques for detecting merging/splitting that we implemented in the Beagle tool were helpful in building a model of how the system had changed.

Acknowledgments

We thank François Marier for his work on detecting merging and splitting in the source code of the VIM text editor, as well as for feedback on the Beagle tool.

References

[1] G. Antoniol, U. Villano, E. Merlo, and M. D. Penta. Analyzing cloning evolution in the Linux kernel. *Information and Software Technology*, 44(13):755, October 2002.

[2] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proc. of the 1995 Working Conference on Reverse Engineering (WCRE-95)*, Toronto, Ontario, July 1995.

[3] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[5] M. W. Godfrey and Q. Tu. Tracking structural evolution using origin analysis. In *Proc. of 2002 Intl. Workshop on Principles of Software Evolution (IWPSE-02)*, Orlando, Florida, May 2002.

[6] J. H. Johnson. Substring matching for clone detection and change tracking. In *Proc. of the 1994 International Conference on Software Maintenance (ICSM-94)*, pages 120–126, Victoria, BC, September 1994.

[7] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Proc. of 1997 Working Conference on Reverse Engineering (WCRE-97)*, Amsterdam, Netherlands, October 1997.

[8] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign., 1992.

[9] M. Rieger and S. Ducasse. Visual detection of duplicated code. In S. Ducasse and J. Weisbrod, editors, *Proceedings ECOOP Workshop on Experiences in Object-Oriented Re-Engineering*. Forschungszentrum Informatik Karlsruhe, 1998.

[10] Q. Tu and M. W. Godfrey. An integrated approach for studying software architectural evolution. In *Proc. of 2002 Intl. Workshop on Program Comprehension (IWPC-02)*, Paris, France, June 2002.

[11] E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proc. of the 2002 Working Conference on Reverse Engineering (WCRE-02)*, Richmond, VA, October 2002.

[12] L. Zou and M. W. Godfrey. Attribute-based identification of structural changes using origin analysis. In preparation, 2003.