Using Origin Analysis to Detect Merging and Splitting of Source Code Entities

Michael W. Godfrey *Member, IEEE* and Lijie Zou

***Abstract*— Merging and splitting source code entities 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 context of the design changes is often lost. That is, it may be obvious which characters of which files have changed, but it may not be obvious where or why moving, renaming, merging, and/or splitting of design elements has occurred. In this paper, we discuss how we have extended *origin analysis* [1], [2] 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 some information about the context of the design change. We also describe a case study of these techniques (as implemented in the Beagle tool) using the PostgreSQL database system as the subject.**

***Index Terms*— Software evolution, origin analysis, restructuring, reverse engineering, and re-engineering**

I. 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]–[5] 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?"*. A software developer who requires answers to these questions must either hope that previous developers have kept accurate and up-to-date documentation, or must make use of tools that can help to extract information about the system's evolution after-the-fact.

Effective reconstruction of the evolutionary history of a software system can also benefit other types of analysis in software evolution research, such as growth analysis based on counting software entities. Most researchers assume that a software entity is uniquely identified by its name (and, perhaps, its logical location within the source code). However, when structural changes, such as moving, renaming, merging, and splitting have occurred, the evolutionary history may appear to be discontinuous, although this is not the case. Having an accurate evolutionary history that takes structural changes into account is thus also of aid to the research community.

In this paper, we propose an approach to detecting function and file merges and splits that 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* [1], [2]; our original formulation of origin analysis did not consider the possibility that program entities might be merged or split between versions.

Ultimately, our goal is to aid the user in gaining a better understanding of the original context of an apparent design change. Origin analysis uses the results of syntactic and semantic analyses to attempt to determine "what happened" to various source code entities from one version to the next. In turn, this information may be used as a basis to infer knowledge of broader or higher level design changes that may have occurred, such as the adoption of a new naming convention, the implementation of a recognized design pattern, or a attempt to consolidate duplicated code within a particular subsystem. By using the results of the origin analysis system model together with experience, domain knowledge, and common sense, the user can build intuition about the context, rationale, scope, and intent of the original design changes. However, in this paper we do not explicitly address how this might be systematically achieved, as it is beyond of the scope of our current work.

The remainder of this paper is structured as follows: In Section II, 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 III, we describe how we have implemented origin analysis and various cases of merging and splitting can be

M. W. Godfrey and L. Zou are members of the School of Computer Science at the University of Waterloo.
Email: {`migod`, `lzou` }`@uwaterloo.ca`

detected. In Section IV, we describe a case study performed on the source code for PostgreSQL, an open source database system that is in wide use. In Section V, we discuss related work, and finally, in Section VI we summarize our results.

II. Origin analysis and merges/splits

A. Definition of origin analysis

We 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 system version, call it V_{old}, 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_{old}, 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.

B. 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 often 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 N-way 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.

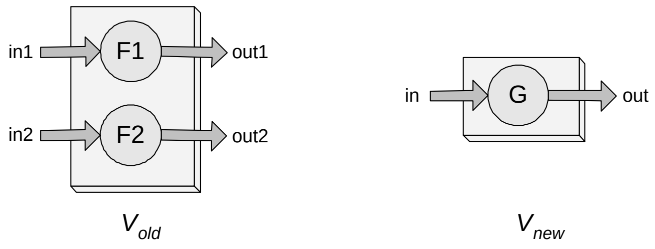


Fig. 1. Canonical two-way function merge.

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.

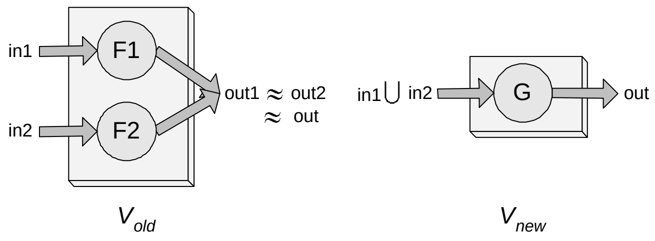


Fig. 2. Clone elimination.

An indicator of this phenomenon is
   - $in_1 \cap in_2 \approx \emptyset \quad \wedge \quad in_1 \cup in_2 \approx in$
   - $out_1 \approx out_2 \approx out$

   where $\approx$ means two sets are approximately the same. 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.

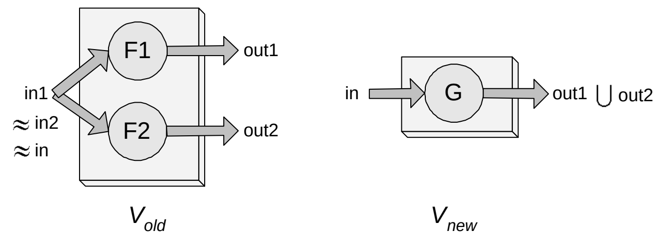


Fig. 3. Service consolidation.

An 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.

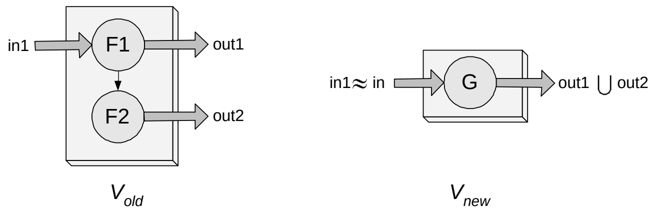


Fig. 4. Pipeline contraction.

An 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 two sections 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 are the ones we had conceived of based on informal experiments with other subject software systems, but before attempting the case study on PostgreSQL; in Section IV, we shall discuss two additional patterns that we discovered to have occurred in the evolution of PostgreSQL.

III. Detecting merging and splitting

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 [2]. 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 [1], 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 our approach to support more matching techniques, and to be more flexible and interactive so as to better support exploratory strategies for detecting merges and splits. The major improvements we have made include:

1) We have generalized the notion of matching. Different matching techniques implemented as "matchers" can be plugged into a common infrastructure. In addition to enhancing the previously mentioned metrics matcher and call relation matcher, we added three new matchers: a name matcher, a declaration matcher, and an expression matcher. The results from the different matchers can also be used in combination.
2) We support semi-automatic and incremental discovery of structural changes. By applying matchers in multiple iterations on selected software entities, the user can identify and informally reason about merging and splitting from historical data, even in complex situations such as sequences of structural changes that are "chained" together.

In the remainder of this section, we first briefly discuss each of the matchers and their running times. Then we give an overview of the tool Beagle from the point of performing origin analysis. Finally we describe how to detect merging and splitting under various circumstances.

A. Matchers

1) Name matcher: The $name\_matcher$ calculates the *longest common substring* (LCS) of the name of the target entity against the names of each of the members of the candidate set, and normalizes the value against the average length of the two entity names.

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

The user may optionally specify if the comparison should be case sensitive, and if a string transformation should apply to one of the names (in the case that a naming convention has been changed between versions).

2) Declaration matcher: The $declaration\_matcher$ performs a normalized LCS calculation similar to that of the name matcher above. By default it removes the parameter types from the declaration string, and then lexically sorts the parameter names. This ensures that changing parameter types or the order of the parameters will not affect the search results; the disadvantage of this approach is that may return more false positives, and it will not handle the case where parameter names really have changed.

As future work, we intend to implement several options for this matcher, whereby the user can specify:

- the behaviour described above, or
- to keep the parameter types and ignore the names, or
- to keep both the parameter types and names.

3) Metrics matcher: The $metrics\_matcher$ compares the differences in various metric values and returns a normalized result. Currently, we use five metric values in our calculations; our preliminary investigations suggest that "more is better" but also that no single metric is a more reliable indicator than any other. The basic formula is:

$$Ans = \sum_{i=1}^{n} M_i$$

where n equals five in our current selection of metrics, each M_i is non-negative, and the sum of the M_is is guaranteed to be at most 1. An example metric we currently use is:

$$M_{LOC} = \begin{cases} 0.2 & \text{if} \quad |LOC(e_1) - LOC(e_2)| \leq 5 \\ 0.1 & \text{if } 5 < |LOC(e_1) - LOC(e_2)| \leq 10 \\ 0 & \text{if} \quad |LOC(e_1) - LOC(e_2)| > 10 \end{cases}$$

where $LOC(e)$ is the number of lines of code in entity e. The choice of which metrics to use and how to weight them is a subject of ongoing research; we initially tried the approach suggested by Kontogiannis *et al.* [6] and are now experimenting with other formulations.

4) Call relation matcher: The (call) $relation\_matcher$ returns a normalized value indicating how closely the caller/callee sets of two entities match; in our experience, it is also the most useful matcher for detecting merges and splits. The matcher computes the similarity of two entities by comparing: (a) both callers and callees, (b) only callers, or (c) only callees. The calculation of similarity for considering both callers and callees is as follows:

$$2 \times \frac{\#(caller(e_1) \Cap caller(e_2)) + \#(callee(e_1) \Cap callee(e_2))}{\#caller(e_1) + \#caller(e_2) + \#callee(e_1) + \#callee(e_2)}$$

where $S_1 \Cap S_2$ is the set of elements in S_1 that have been matched to elements in S_2 (*i.e.,* it is effectively set intersection that incorporates knowledge of any previous origin analysis matching). Similar definitions hold for queries considering only callers and only callees.

Comparing entities 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 II-B 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.

5) Expression matcher: The $expression\_matcher$ differs from the other matchers in several ways: it allows the user to create composite queries (using the other matchers), it allows thresholds to be specified, it allows multiple targets to be checked against candidate sets, and it returns a 2D matrix of all of the results. For example, one could ask for the results of matching all entities in a given file (the targets) against all other unmatched entities (the candidate set), where $name \geq 0.8 \wedge metrics \geq 0.5$. One could then see if any obvious matches are apparent, enter those matches into the system model, possibly remove the matched targets from the target set, and then perform subsequent queries changing the thresholds and/or matchers used. Since the similarities are computed only once and then can be queried in multiple iterations, using this matcher can speed up the analysis process.

B. Matcher running times

For the $name\_matcher$, the running time of computing LCS is $\mathcal{O}(pq)$, where p and q are the lengths of the two strings. If we assume that there is an upper bound for the name length, then the running time is $\mathcal{O}(1)$. In each iteration of origin analysis, we match m entities from version 1 and n entities from version 2, thus the whole matching process is $\mathcal{O}(mn)$.

The $declaration\_matcher$ works similarly to the name-matcher, and its running time is $\mathcal{O}(mn)$ for analogous reasons.

For the $metrics\_matcher$, comparing a pair of entities is constant time, since the individual metric values are precomputed during system check-in. Finding matches from m entities in version 1 and n entities in version 2 is therefore $\mathcal{O}(mn)$.

For the $relation\_matcher$, comparing a pair of entities is $\mathcal{O}(pq)$, where p and q are respectively the combined number of callers and callees of the two entities. If we also assume that p and q have a constant upper bound, then the running time becomes $\mathcal{O}(1)$. Thus, an iteration of origin analysis for m and n entities from two versions is $\mathcal{O}(mn)$.

However, we note that in practice, all of these values are fairly small and the automated parts of origin analysis are therefore almost instantaneous. The time the user takes to browse the version spaces and decide on strategies is the bottleneck for origin analysis.

We note that the detailed syntactic and semantic analysis that occurs when a system version is checked is much more complex and time consuming, but it is performed only once per system version. In "real world" terms, the amount of time this step takes is comparable to a full system compile, which is not surprising, given that the analysis tools are fundamentally special-purpose compilers.

C. The Beagle tool

The matching techniques described above to support origin analysis have been implemented in a prototype tool named *Beagle*.[1] Beagle is a research platform that is intended to help developers gain an understanding of a software system's evolutionary history [1], [7]. It 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. Here we only give an overview of Beagle from the point of performing origin analysis. Details about its architecture and other functionalities are described elsewhere [7].

Before origin analysis can be performed, the user needs to check in versions of the software system into the Beagle repository. For each version, Beagle uses the external tool SWAGkit [8] to extract static relations between program entities, and *Understand for C++* [9] to calculate various metrics. The whole check-in process is supported by a Java tool.

After the factbase has been built, the user can perform origin analysis for any pair of versions (usually, each pair of consecutive versions is analyzed). The origin information is added as annotations to the entities of the system model that is stored in the Beagle repository. There are nine possible annotation values for each software entity: *unchanged* (by far the most common situation), *deleted*, *added*, *moved* (*i.e.,* to a different structural container), *renamed*, *split*, *merged*, *combined* (*i.e.,* satisfies at least two of the previous categories), and *unknown* (the default value at the beginning of origin analysis). In the case of entities that are annotated as unchanged, moved, renamed, split, or merged, the identities of the entity versions of the other system version are added (*e.g.,* methods `f` and `g` in the new version were split from method `h` in the old version).

Figure 5 shows Beagle's software architecture from the perspective of a user performing origin analysis, and Fig. 6 shows a corresponding view of the Beagle user interface (UI). At this point, the desired system versions have already been checked in, and thus the salient "facts" about its evolutionary history have been added to the the repository. The user first loads the two versions of interest from the repository into the (in-memory) internal model; these appear as navigable tree views in the UI ("entity trees"). Then she iteratively selects entities of interest (the "entity list" in the UI) , applies one or more "matchers" and views the results (the "candidates viewer"). In addition to the text widget view shown at the right side of Fig. 6, other kinds of visualizations are supported including scatter plots (discussed later in this section). The internal model can be incrementally "marked up" (indicated by various icons in the entity trees and lists), and may be committed to the repository.

[1]Beagle is named after the ship Charles Darwin sailed on. The Beagle tool is intended to be a vehicle for exploring the evolution of a software system.

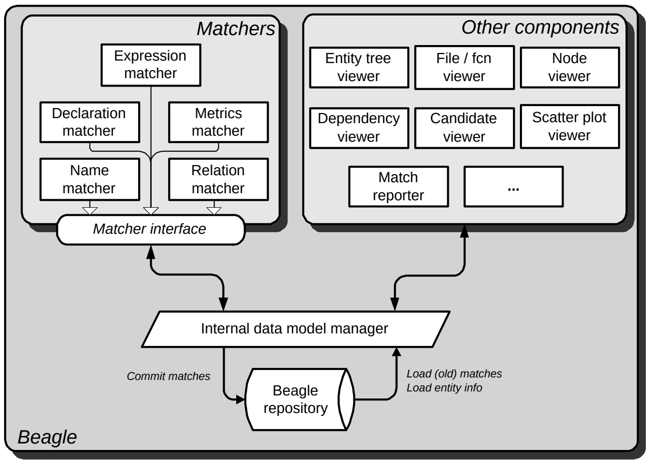


Fig. 5. *Architecture view of Beagle for performing origin analysis.*

D. 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.

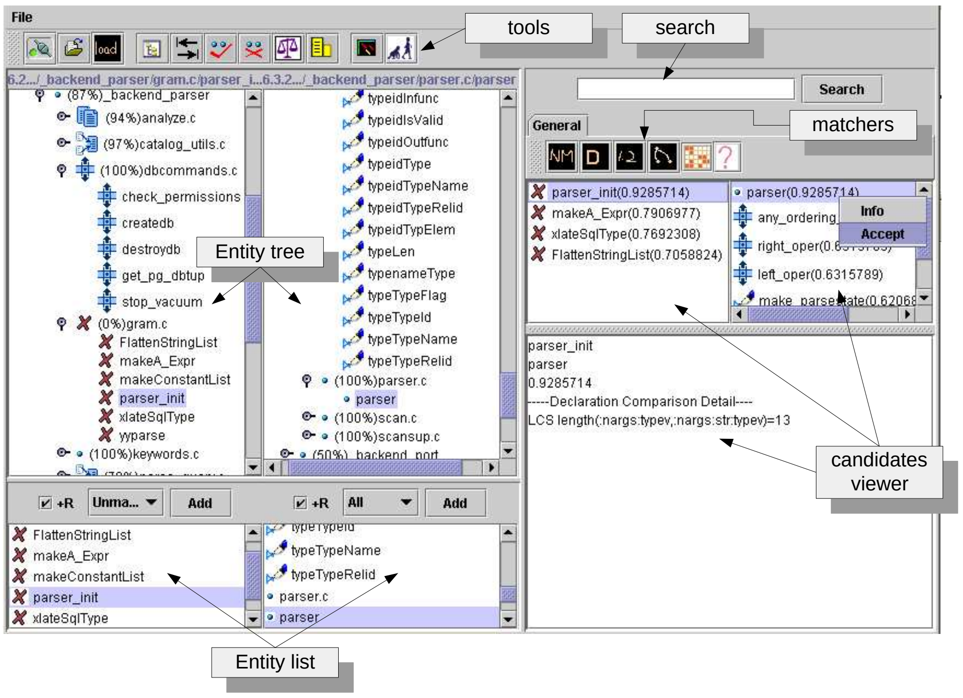
For simplicity of explanation, we will assume the user is considering an "apparently new" entity, which we will call the target. The basic iterative process for performing origin analysis is straightforward:

1) the user decides on a candidate entity set of interest from the old version,
2) she applies one or more "matchers" to the target and the candidates, and
3) she 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, say, 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 plugins to the Beagle tool; currently, there are four basic matchers for functions — $name\_matcher$, $declaration\_matcher$, $metrics\_matcher$, and (call) $relation\_matcher$ — plus a meta-utility called $expression\_matcher$ that allows the results of the others matchers to be combined into a single query.

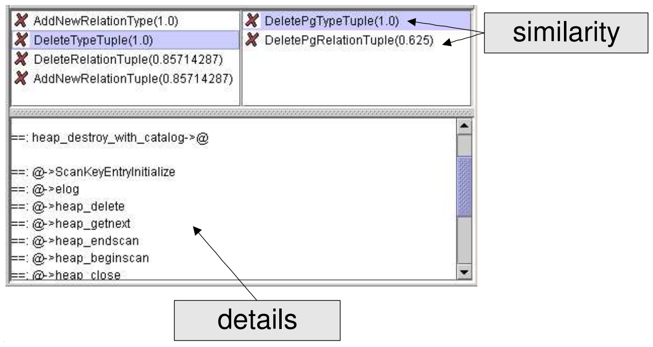
Each of the four basic matchers compares the target entity with each element in the candidate set and computes a number



```
parser_init
parser
0.9285714
-----Declaration Comparison Detail----
LCS length(:nargs:typev,:nargs:str:typev)=13
```

Fig. 6. *A screenshot of the Beagle tool showing the results of a query.*

between 0 and 1, where 1 denotes a perfect match. The candidates in the result set are ranked by computed similarity, and are associated with detailed information of how each similarity is computed. Figure 7 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 in release 6.4.2. For function `DeleteTypeTuple` in release 5.0 (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.


similarity
details
AddNewRelationType(1.0)
DeleteTypeTuple(1.0)
DeleteRelationTuple(0.85714287)
AddNewRelationTuple(0.85714287)
DeletePgTypeTuple(1.0)
DeletePgRelationTuple(0.625)

```
==: heap_destroy_with_catalog->@

==: @->ScanKeyEntryInitialize
==: @->elog
==: @->heap_delete
==: @->heap_getnext
==: @->heap_endscan
==: @->heap_beginscan
==: @->heap_close
```

Fig. 7. Candidate list produced by relation_matcher

In Step 3, the user decides if she 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 that 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 is used in this phase.

E. Detecting chained merges/splits at the 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 can help 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 functions had actually been merged into the file `joinrels.c` in the `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 Fig. 8 and Fig. 9 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.

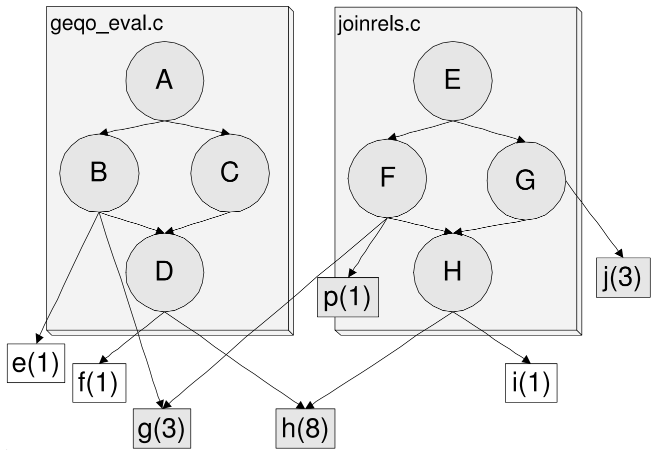


Fig. 8. Call relations in release 6.4.2

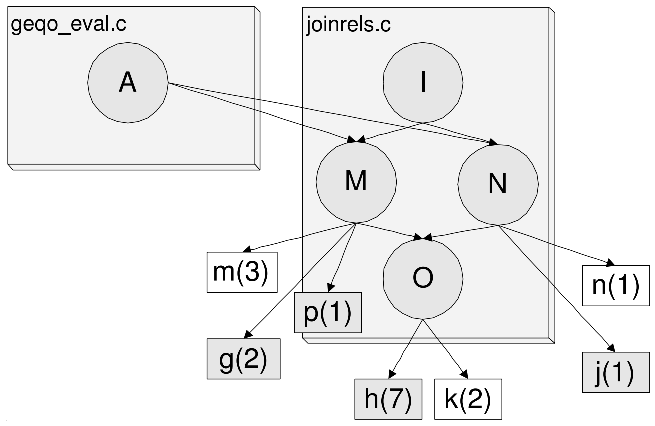


Fig. 9. Call relations in release 6.5

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 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.

F. Detecting merges/splits at the file and subsystem level

Merging and splitting can also occur at the file and subsystem levels. For example, if a new file G is found to be composed of functions from two old files F_1 and F_2, then we can consider that files F_1 and F_2 have been merged into G (a similar statement holds for splits at the file level, and for merges/splits at the subsystem level). Once detected, the user can enter this information as attributes of the files/subsystems in question into the model of the system version in the Beagle repository.

Currently, file- and subsystem-level merges/splits are detected manually in Beagle. We have not automated the detection of file- and subsystem-level merging/splitting as, in our experience, it invariably requires the user's common sense to decide if merging/splitting really has occurred. While we have informally experimented with threshold values (*e.g.,* a merge has occurred if more than 50% of the functions in a "new" file were in a different file in the old version) and made use of scatter plot visualizations [10] (see Section III-G), detecting file- and subsystem-level merging/splitting remains an area of active research for us.

G. Visualization

Beagle supports a variety of visualization tools for browsing the evolutionary history of a software system [7]. Among these is a scatter plot viewer, as shown in Fig. 10. Scatter plots are well known in clone detection research [11]–[13]; 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 10 shows an example from 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

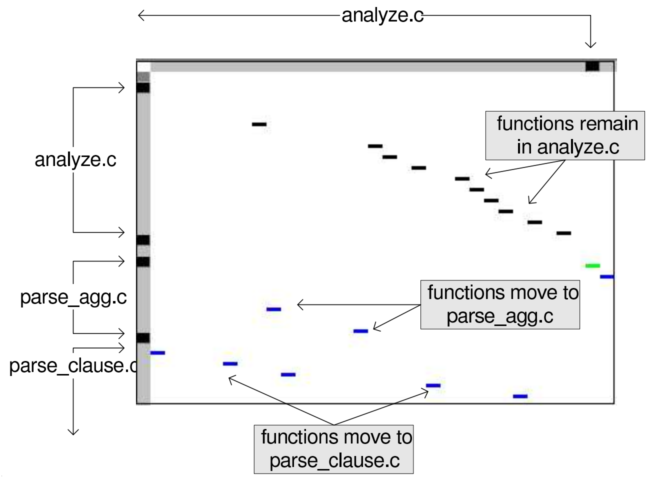


Fig. 10. Scatter plot showing function movement between versions.

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.

IV. Case study: Applying origin analysis to PostgreSQL

We have performed a case study on the use of origin analysis to detect merging and splitting using PostgreSQL as the subject system. 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 that is in wide use.

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 total code base.

Table I summarizes the growth of the backend subsystem of PostgreSQL. The number of subsystems is the number of directories in the source code tree that contain files of source code. The number of .c files was calculated from this Unix command

```
find . -name "*.c" | wc -l
```

after compilation. The number of functions reported here are those that were successfully analyzed by SWAGkit and loaded into Beagle; it ignores a very small number of functions that the SWAGkit extractor was unable to parse. Finally, LOC refers to the sum of the lines of code (LOC) of all of the .c and .h files after compilation.

TABLE I
Growth of PostgreSQL

Release	Date	# subsys	# files	# fcns	# LOC
6.2	1997-10-02	50	328	3262	186,037
6.3.2	1998-04-7	49	346	3321	193,980
6.4.2	1998-12-20	49	358	3372	203,875
6.5	1999-06-9	49	365	3529	216,433
6.5.1	1999-07-15	49	365	3533	216,533
6.5.2	1999-09-15	49	365	3536	215,644
6.5.3	1999-10-13	49	365	3536	215,815
7.0	2000-05-8	49	380	3974	244,362
7.0.3	2000-11-11	49	380	3980	244,561
7.1	2001-04-13	49	384	4191	257,906
7.1.3	2001-08-15	49	384	4202	258,393
7.2	2002-02-4	50	388	4531	279,385

Over the four and a half years of the case study window, the number of subsystems was almost constant while at the same time, there was a significant increase in the number of source (.c) files (18%), functions (39%), and LOC (50%). This suggested to us that the system architecture of the backend subsystem was fairly stable, while at the same time significant changes and additions were occurring within. That is, the backend subsystem of PostgreSQL seemed to be a good candidate for studying origin analysis.

A. 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 detected in each release in Table II. The second through fourth columns show the situation before performing origin analysis: the number of matched program entities (we considered subsystems, files, and functions in our study), the number of apparently deleted entities, and the number of apparently added entities. The fifth through eighth columns show the number of matched, added, and deleted entities after origin analysis; we note that the number matched in V_{old} and V_{new} may no longer be the same, since merging and splitting may have been found to have occurred. The ninth through twelfth columns show the numbers of each kind of broad change that were observed.

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 were no merges or splits observed in three minor releases. These observations conformed to our expectation that most structural changes occur during major release updates.

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 IV-B) combined with *pipeline expansion*. These splits resulted from the introduction of a standardized way of expression tree walking using the Template Method design pattern [14];

TABLE II
SUMMARY OF ORIGIN ANALYSIS RESULTS FOR POSTGRESQL

$V_{old} \rightarrow V_{new}$	BEFORE ORIGIN ANALYSIS			AFTER ORIGIN ANALYSIS				ORIGIN ANALYSIS RESULTS			
	Matched	**Deleted**	**Added**	**Matched** V_{old}	**Matched** V_{new}	**Deleted**	**Added**	**Moved**	**Renamed**	**Merged**	**Split**
6.2 → 6.3.2	3389	219	286	3517	3521	91	154	95	29	13	3
6.3.2 → 6.4.2	3467	208	263	3515	3515	160	215	0	48	0	0
6.4.2 → 6.5	3517	213	368	3616	3601	114	284	20	70	22	2
6.5 → 6.5.1	3876	9	13	3878	3888	7	1	0	2	0	10
6.5.1→ 6.5.2	3886	3	6	3886	3888	3	4	0	0	0	2
6.5.2→ 6.5.3	3892	0	0	3892	3892	0	0	0	0	0	0
6.5.3 → 7.0	3383	509	954	3583	3596	309	741	86	114	2	14
7.0 → 7.0.3	4336	1	7	4336	4336	1	7	0	0	0	0
7.0.3 → 7.1	3640	703	913	3725	3726	618	827	19	65	1	2
7.1 → 7.1.3	4547	6	17	4551	4551	2	13	4	0	0	0
7.1.3 → 7.2	4365	199	533	4420	4427	144	471	12	42	0	11

TABLE III
PERCENTAGE CHANGE IN APPARENTLY ADDED AND DELETED ENTITIES AS A RESULT OF PERFORMING ORIGIN ANALYSIS.

$V_{old} \rightarrow V_{new}$	**% change in # deleted**	**% change in # added**
6.2 → 6.3.2	58	46
6.3.2 → 6.4.2	23	18
6.4.2 → 6.5	46	23
6.5 → 6.5.1	22	92
6.5.1→ 6.5.2	0	33
6.5.2→ 6.5.3	0	0
6.5.3 → 7.0	39	22
7.0 → 7.0.3	0	0
7.0.3 → 7.1	12	9
7.1 → 7.1.3	67	24
7.1.3 → 7.2	28	12
Overall	30	19

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 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.

Table III further summarizes the results in terms of how much "noise" (*i.e.,* false positives) was eliminated from the set of entities that appear to have been added or deleted. Overall, we were able to reduce the number of "apparently deleted" entities by 30% and the number of "apparently added" entities by 19%. We consider that these are significant results.

B. Two more patterns

In addition to the merge/split cases patterns described in section II-B, 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.
   An indicator of this phenomenon is
   - $in_1 \cup in_2 \approx in$
   - $out_1 \approx out_2 \approx out$
   - $decl_1 \sim decl_2 \quad \wedge \quad decl_1 + param_{new} \sim decl$

   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.
   An example of this pattern (Table IV) occurred when the functions `RelationSetLockForRead` and `RelationSetLockForWrite` in release 6.4.2 were merged to form `LockRelation` in release 6.5. The two old functions set a lock on a relation in either read/write mode. The new merged function added a parameter to distinguish the different lock mode.
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.

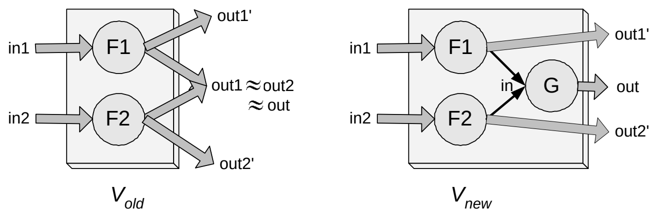


Fig. 11. Partial clone elimination

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

An example of this pattern (Table IV) occurred when a common recursive walking idiom of an expression structure in functions _finalize _primnode , fix _opid , and other functions in release 6.4.2 were extracted out to form a new function expression _tree _walker in release 6.5.

Partial clone elimination differs from clone elimination in the amount of duplicated code that the parent functions share. If we attempted to define a hard boundary to distinguish between the two cases, the distinction would be fairly arbitrary. The reason that we consider it worthwhile to distinguish between them is that the intents behind the the two patterns are different. In clone elimination, the original parent functions are clones and perform the same tasks, while in case of partial clone elimination, they are not clones and merely perform the same subtasks. Considering that maintainers usually add or modify the code after they clone from somewhere else, it would not be surprising to see, as we did in our case study, that instances of partial clone elimination outnumber those of clone elimination.

C. Combination of patterns

As is common with the application of design patterns [14], 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 expansion* (although we counted it only as one instance in Table. III). 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 expansion* 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 adaptor [14] 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. As we adopt a semi-automatic approach, we are still able to explore complex situations. Without losing any existing flexibility, we intend to add more automated support for pattern detection in the future.

D. Instances of different patterns

In Table IV, 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* tended to have a relatively large number of instances. This suggests that much effort had been invested in eliminating duplicate code fragments, routines, and idioms in the PostgreSQL source. It also suggests that clones are good starting points for merge/split detection, and that *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 entities themselves often provided clues about the type of the change, such as when gettypelem and typtoout were merged to form getTypeOutAndElem , and when appendStringInfo split off enlargeStringInfo . This indicates that entity names are a valuable source of information in merge/split detection.

E. Groups of merges/splits at the function level

Three major 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 the 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 owner of a large number of files to restructure modules relatively easily without worrying about "breaking" what other developers might be doing. It would be interesting to investigate whether the group change phenomena are different in other OSS projects without a core development team. We intend to investigate this in the future.

F. Merges/splits at the file and subsystem level

We found three groups of merges/splits at the file level. The first group consisted of three splits (the files catalog _utils.c , analyze.c , and parse.c in the old system version) and seven merges (files named parse _XXX.c for various values of XXX in the new system version), and 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. It is interesting to note that the functions themselves were not merged or split, but were left intact. For example, the functions in analyze.c were moved into seven files, and a new file parse _agg.c was formed from the function agg _error from catalog _utils.c , four

TABLE IV
OCCURRENCES OF PATTERN INSTANCES FOUND IN POSTGRESQL.

			EXAMPLES			
Pattern	**Merge/Split**	**# found**	**Functions in V_{old}**		**Functions in V_{new}**	**Version IDs**
clone elimination	merge	7	`getAttrName`, `get_attname`	→	`get_attname`	6.2 → 6.3.2
service consolidation	merge	1	`gettypelem`, `typtoout`	→	`getTypeOutAndElem`	6.4.2 → 6.5
pipeline expansion	split	6 (+ 23)	`appendStringInfo`	→	`appendStringInfo`, `enlargeStringInfo`	6.4.2 → 6.5
parameterization	merge	3	`RelationSetLockForRead`, `RelationSetLockForWrite`	→	`LockRelation`	6.4.2 → 6.5
partial clone elimination	merge	27	`_finalize_primnode`, `fix_opid`, ...	→	`expression_tree_walker`	6.5 → 6.5.1

functions from `analyze.c`, and the function `ParseAgg` from `parse.c`. So in this case, it is a regrouping at the file level.

The second group consisted of two merges, and resulted from a "clean up" of the `optimizer` subsystem from release 6.4.2 to 6.5; most of the functions in the files `geqo_eval.c` and `geqo_paths.c` in the `optimizer_geqo` subsystem were merged into other files in the subsystem `optimizer_path`. In this case, the regrouping occurred at the function level.

The third group consisted of one split and two merges, and corresponded to a redesign of the `utils/adt` subsystem from release 6.5.3 to 7.0. In detail,

- the file `dt.c` was deleted, and its functions were distributed (with some renaming) between the files `datetime.c` and `timestamp.c`,
- all of the functions in the old version of the file `datetime.c` were merged (with some renaming) into the new versions of the files `date.c` and `nabstime.c`, and
- all of the functions in the old version of the file `date.c` were merged (with some renaming) into the new version of the file `nabstime.c`.

We can see that file-level merges/splits occur both in significantly smaller numbers and much less frequently than function-level merges/splits. Furthermore, we note that file-level merges/splits seem to occur only during major releases; this is perhaps unsurprising since structural changes at the file level often represent a major design-level change in the code, and the resulting "ripple effect" may have a large impact on the rest of the system. Such changes upset the stability of the system as a whole, and so are less likely to be effected during a minor release cycle.

Finally, we note that we did not notice any instances of subsystem merging/splitting in the case study. This is because the subsystem structure was almost constant throughout the history of PostgreSQL.

G. Summary: Merging and splitting in PostgreSQL

In summary, we found that merging and splitting accounted for about 12% of the total number of structural changes in our case study of 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.

We found that we generally spent between two and four hours performing origin analysis on a pair of system versions between major releases, less time for minor releases. Detecting moves was the easiest, while detecting merges and splits and chained changes were the most time consuming. We could have spent more time on each pair of versions, but we stopped when we felt that our efforts were no longer being rewarded. It is not clear to us if a practitioner would be willing to spend this amount of effort to gain an accurate view of a software system's evolutionary history, but we note that it need only be done once per release, and an active developer would have a much easier time performing origin analysis on his/her own system than we did as outsiders to the PostgreSQL project. That is, ignoring the learning curve for the tool itself, we consider our results to be an upper bound on the cost of performing origin analysis.

H. Discussion and future work

The empirical data and our experience with the case study has provided some insight into the advantages and potential disadvantages of our approach, which we now discuss.

We found that our approach was simple but flexible, which was an advantage in exploring this new conceptual terrain. It did not require a structural change pattern to be specified beforehand, and this enabled us to discover patterns that were actually used by developers, not conceived by researchers. The flexibility was also useful in detecting chained changes. How changes are chained together depends largely on the relationship between the software entities involved, thus is hard to model. By including the user as an active part of the analysis process with strong tool support, it makes the complex change situations easier to understand.

Our approach is mostly programming language independent. Only the (external) analysis tools that parse the source code are dependent on the programming language. The output from the analysis tools is converted into a general procedural/object-oriented model, and the Beagle tool (*e.g.,* the matchers, the visualizers, the system models) uses this abstracted schema as its internal meta-model for the systems being examined. Currently, only C/C++ is fully supported, but work on a Java extractor for SWAGkit was recently completed by one of the authors, and a Java version of the metrics tool *Understand* exists.

Matching techniques work best when the attributes they are comparing — *e.g.,* name, declaration, metrics, call relations — are highly similar. In our case study, we found that if one (non-trivial) attribute of a pair of entity versions was exactly the same, such as both have the same non-empty declaration, often the candidate turns out to be a "real" match. Also if more than one of the attributes are highly similar, such as both name and call relations, often the candidate turned out to be a real match. For this reason, we are developing an expression matcher that supports queries of similarities from multiple matchers. We are also investigating heuristics for what combinations work well together.

When software entities undergo significant changes, which is common when merging and splitting occur, we have found the matchers that compared semantic attributes — such as the name, declaration, and call relations — to be the most useful. This might be because developers tend to preserve these semantic attributes as much as possible in maintenance activities. In particular, the call relation matcher often provided not only the origin information about a software entity, but also its change of interaction with other entities, which in turn enabled us to discover some of the context behind the design change. Also, the candidate list produced by the call relation matcher is usually small, as the callers and callees have to be matched to be considered as a vote in matching, which restricts the candidates to a small subset of entities. As future work, we plan to include more relations into our study, such as global variable use.

One potential disadvantage of our approach is that it requires significant human interaction, including choosing entities to be matched and deciding which of the candidates is the "real" match. Although in many cases, such as moving and renaming, it is easy for the user to decide on the real matches, there were cases for which the reasoning about the relations between different software entities involves considerable work. We are considering two ways to improve this: (a) to reduce the time window of origin analysis, such as for every month instead of every two releases, and (b) to provide more support for reasoning, such as assigning software entities different weights according to significance and visualizing relation differences.

V. Related work

A. Refactoring

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. Opdyke was the first to use the term in its present sense [4], but Fowler's book is the best known distillation of this body of knowledge [3]. 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" to look for in code, as well as a set of appropriate actions (refactorings) to take in each case. There are several research tools that perform "bad smell detection" on software systems based on this catalogue [15], [16].

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* and serve as intellectual tools for the software developer/maintainer. Our patterns are more low level, and correspond more closely to *what-would-a-developer-do-to-the-code*; consequently, they may be easier to detect semi-automatically, and this is why we have chosen this approach.

B. Detecting refactorings via changes in metrics values

Demeyer *et al.* have also investigated the idea of trying to discover refactoring activities that have taken place within a software system [17]. While their broad goals are similar to ours, their approach is quite different. They analyze the source code to generate a set of ten characteristic metrics about the classes and methods, such as number of lines of code, number of method calls within a method (*i.e.,* fan-out), or the number of methods defined in a class. They then use this information together with knowledge of the source code inheritance hierarchy to guess where particular refactorings might have occurred.

For example, if a set of methods each experience a similar reduction in the number of method calls, lines of code, and number of statements, then this makes them a good candidate for the *Split Method / Factor Out Common Functionality* refactoring (similar to our *clone elimination* pattern). With this knowledge in hand, the developer then browses the source code to find where the missing functionality might have been moved to.

Fundamentally, their approach relies on comparing metric values of program entities (based on a syntactic analysis of the program) and then browsing the source code to decide if a given refactoring did indeed occur. Our approach incorporates semantic knowledge of the source code, and allows the user to ask semantically richer questions such as *Do f and g call the same functions?* rather than *Do f and g have the same fan-out?*. Additionally, as they point out, their approach is susceptible to the renaming of program entities, whereas discovering and recording incidents of renaming is a key activity of our approach. Once a rename or move has been detected, we take this into account in any future queries on the code.

We note that their ultimate goal, refactoring detection, is different from our goal of finding out *"Are these new/deleted entities really new/deleted, or did they come from somewhere else within the system?"* Thus, detecting entity renaming is not a high priority for them, and they chose not to take entity names into account in their analysis of call sets, *etc.* However, if they had performed a semantic analysis to determine which entities were involved in which relations, this additional information could have reduced the number of false positives in their queries, and also allowed them to more easily examine entities with highly similar call sets where the cardinalities were not identical. We note that based on their reported results, their approach of using metrics values appears to be both reasonable and effective for their purposes; however, a metrics-only approach is clearly insufficient for our goal of origin analysis.

Finally, we note that their approach focuses on object-oriented software systems, whereas so far we have concentrated on procedural languages.

C. Clone detection

Origin analysis borrows some techniques from software clone detection [6], [18], [19]; however, the aim of origin analysis — to detect where, how, and why structural changes have occurred — 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. We note that Johnson, one of the earlier researchers in clone detection, did include an example of using a scatter plot to show how versions of a system differ [12].

As mentioned above, origin analysis consists of entity and relationship analysis. We chose to use an existing metrics-based clone detection technique [6] for entity analysis as it was simple and fast to implement and made version comparison a trivial matter of computing a few numerical results. In principle, any clone detection technique could be substituted, although adopting a finely-grained approach based on comparing Abstract Syntax Trees (ASTs) or Program Dependence Graphs (PDGs) [19] would mean that comparing a target entity to a candidate set would go from a near-constant time operation to a linear or polynomial time operation, depending on the technique used.

Finally, we note that as far as we are aware, our relationship analysis technique is novel, and could be applied as a clone detection technique in its own right, although we have not yet done so.

D. Intelligent merging and renaming detection

The software configuration management (SCM) community — both research and industry — has developed "intelligent" tools for performing versioning and merging. Such tools are capable of modelling software systems as more than just sets of files of text characters; typically, they have some understanding of the structure of the source code, and may be able to model changes to, say, individual functions or model "change-sets" (changes to a set of files that implement a single bug fix, for example). However, while these ideas have been discussed and even implemented within the SCM community for several years now [20], [21], the use of tools that implement them is not yet standard industrial practice. Consequently, we decided to investigate techniques for reconstructing information about structural changes that did not assume the use of such tools.

Of the research in this area, the work that is the most closely related to origin analysis is that of Hunt *et al.* , who have investigated the problem of *renaming detection* of identifiers (including functions, global and local variables) between successive versions of a software system [22], [23]. Their goal is to aid in the well-known SCM problem of merging program variants. This problem arises when two (or more) variant branches of a software system are created from a common parent; if changes are made to the same parts of the system in both variants, then extreme care must be taken when the variants are merged back into a common baseline. In particular, if identifiers of the parent version have been renamed in one of the variants and code has been added or changed in the other variant that uses the old identifier names, then the merging must take this into account, and the merging process itself will likely be very complex and error prone.

Traditionally, text-based differencing and merging algorithms are used to perform the merging of the variants; that is, the SCM tool treats the program sources as plain text in this context, and does not use any special knowledge of the rich syntactic and semantic information that source code contains. However, text-based approaches obviously do not fare well in the presence of identifier renaming.

The approach of Hunt *et al.* involves scanning and (limited) parsing of the system code, and followed by a kind of multi-phase token-based clone detection. First, the source files are converted to token streams, and the places where identifiers are declared and used are noted. Next, they compare the identifiers in the base and variant versions by performing a straightforward comparison of the tokens sequences around the identifier declaration for variable and method signatures ("declaration similarity") and method definitions ("implementation similarity"). Then they compare the token sequences around the vicinities of the uses (or references) of the identifiers in both versions ("reference similarity"); this bears some explanation. Suppose that there is an identifier id_{old} that is defined and referenced several times in the parent version. For each such reference in the parent, they look for a corresponding reference in the variant at approximately the same place in the code (as determined by similar neighbouring token sequences). If such an identifier reference is found, call it id_{new}, then a match is registered between (the definition of) id_{old} in the parent and id_{new} in the variant. At the end of the pass, the number of pairwise matches are recorded. Finally, they use an expert system to weigh the relevance of each suggested matching, based on the declaration, implementation, and reference similarities computed above.

Hunt's *renaming detection* is similar in its practical goal to *origin analysis*: it attempts to find instances of identifiers having been renamed. However, the underlying problem that renaming detecting tries to aid in — source variant merging in the presence of identifier renaming — is different from that of origin analysis — long- and short-term program comprehension in the presence of refactoring. Hunt's approach is, by design, lower level and computationally more expensive in that it extracts and compares token streams of source code variants. In origin analysis, the "expensive" analysis is performed only once, when each system version is checked into the repository (and the analysis is roughly comparable to the complexity of compilation); when versions are compared, entity analysis involves comparing vectors of (pre-computed) numbers, and relation analysis involves comparing (relatively small) sets of program entities. We also note that we consider origin analysis to be a semi-automatic comprehension task, and we have designed the Beagle tool to support trial-and-error, incremental system model building. Finally, we note that Hunt's approach does not explicitly consider merging, splitting, and cloning of software entities, which is the focus of our work here.

E. Gold's conceptual change framework

Gold *et al.* have proposed a framework for understanding conceptual changes in evolving software systems [24]. The framework attempts to characterize types of conceptual changes via transformations in the locality and interpretation of regions of source code. Their work is derived from a case study of commercial COBOL systems.

Gold's approach is to create a set of "concepts" for a given software application, and then create a set of source code "indicators" for the concepts. The source code is then analyzed to look for indicators of the various concepts; if one is found and matched, then the position in which it occurs is noted and a "hypothesis" for the concept is generated. This model can then be applied to different versions of the software system, and can be used to trace the "morphological evolution" of the system's concepts. Often, this means that an initially well-designed software system can be seen to exhibit a visible loss in conceptual integrity as it ages; this is because concept indicators tend to become scattered as the code segments that contain them are added, deleted, moved, split, and merged.

We note that Gold's approach is aimed at high-level comprehension of software system evolution, a higher level than that of origin analysis. It attempts to track the morphological evolution — including merging and splitting — of system "concepts" over time, whereas origin analysis addresses the morphological evolution of files and functions.

VI. Summary

Merging and splitting source code artifacts are common activities during development and maintenance, yet it is common for the original context of these design changes (the scope, rationale, intent, *etc.*) 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. In turn, we found that this knowledge was helpful in discovering some of the original context for these design changes.

Acknowledgements

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.

An earlier version of this paper appeared in the proceedings of the *2003 IEEE Working Conference in Reverse Engineering* [25].

References

[1] 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, pp. 127–136.
[2] 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, pp. 117–119.
[3] M. Fowler, *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.
[4] W. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign., 1992.
[5] T. Mens, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, February 2004.
[6] 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, pp. 577–586.
[7] L. Zou, "Toward an improved understanding of software change," Master's thesis, University of Waterloo, 2003.
[8] SWAGkit homepage, Website, http://www.swag.uwaterloo.ca/swagkit/.
[9] ScientificToolworks, "Understand for C++," Website, http://www.scitools.com/ucpp.html.
[10] K. W. Church and J. I. Helfman, "Dotplot: A program for exploring self-similarity in millions of lines of text and code," *Journal of Computational and Graphical Statistics*, vol. 2, no. 2, pp. 153–174, June 1993.
[11] 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, pp. 86–95.
[12] J. H. Johnson, "Substring matching for clone detection and change tracking," in *Proc. of the 1994 International Conference on Software Maintenance (ICSM-94)*, Victoria, BC, September 1994, pp. 120–126.
[13] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proc. of 1999 Intl. Conference on Software Maintenance (ICSM-99)*, Oxford, UK, September 1999, pp. 109–118.
[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.
[15] 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, pp. 227–237.
[16] T. Tourwe and T. Mens, "Automatically identifying refactoring opportunities using logic meta programming," in *Proc. of 2003 European Conference on Software Maintenance and Reengineering (CSMR-03)*, Benevento, Italy, March 2003, pp. 91–100.
[17] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," in *Proc. of 2000 Intl. Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA-00)*, Minneapolis, USA, October 2000, pp. 166–177.

[18] G. Antoniol, U. Villano, E. Merlo, and M. D. Penta, "Analyzing cloning evolution in the Linux kernel," *Information and Software Technology*, vol. 44, no. 13, pp. 755–765, October 2002.

[19] J. Krinke, "Identifying similar code with program dependence graphs," in *Proc. of 2001 Working Conference on Reverse Engineering (WCRE-01)*, Stuttgart, Germany, October 2001, pp. 301–309.

[20] R. Conradi and B. Westfechtel, "Version models for software configuration management," *ACM Computing Surveys*, vol. 30, no. 2, pp. 232–282, June 1998.

[21] T. Mens, "A state-of-the-art survey on software merging," *IEEE Transactions on Software Engineering*, vol. 28, no. 5, pp. 449–462, May 2002.

[22] J. J. Hunt and W. F. Tichy, "Extensible language-aware merging," in *Proc. of 2002 Intl. Conference on Software Maintenance (ICSM-02)*, Montréal, Canada, October 2002, pp. 511–520.

[23] G. Malpohl, J. J. Hunt, and W. F. Tichy, "Renaming detection," in *Proc. of 2000 Intl. Conference on Automated Software Engineering (ASE-00)*, Grenoble, France, September 2000, pp. 183–202.

[24] N. Gold and A. Mohan, "A framework for understanding conceptual changes in evolving source code," in *Proc. of 2003 Intl. Conference on Software Maintenance (ICSM-03)*, Amsterdam, Netherlands, September 2003, pp. 431–439.

[25] L. Zou and M. W. Godfrey, "Detecting merging and splitting using origin analysis," in *Proc. of 2003 Working Conference on Reverse Engineering (WCRE-03)*, Victoria, BC, November 2003, pp. 146–154.

PLACE PHOTO HERE

Michael W. Godfrey Michael W. Godfrey is an assistant professor in the School of Computer Science at the University of Waterloo. He is the associate chairholder of the Industrial Research Chair in Telecommunications Software Engineering sponsored by Nortel Networks, the (Canadian) National Science and Engineering Research Council (NSERC), and the University of Waterloo. He holds a Ph.D. in Computer Science from the University of Toronto (1997).

PLACE PHOTO HERE

Lijie Zou Lijie Zou is a Ph.D. candidate in the School of Computer Science at the University of Waterloo. She holds an M.Math. in Computer Science from the University of Waterloo (2003) and a B.Sc. from Fudan University, China (1997). Her research interests include software evolution, software architecture and program comprehension.