Four Interesting Ways in Which History Can Teach Us About Software

Michael Godfrey Xinyi Dong Cory Kapser Lijie Zou
Software Architecture Group (SWAG)
School of Computer Science
University of Waterloo
Waterloo, Ontario, CANADA
email: {migod,xdong,cjkapser,lzou}@uwaterloo.ca

Abstract

In this position paper, we outline four kinds of studies that we have undertaken in trying to understand various aspects of a software system's evolutionary history. In each instance, the studies have involved detailed examination of real software systems based on "facts" extracted from various kinds of source artifact repositories, as well as the development of accompanying tools to aid in the extraction, abstraction, and comprehension processes. We briefly discuss the goals, results, and methodology of each approach.

1. Introduction

This position paper describes four broad approaches to studying software system evolution that yield different perspectives on how and why a system has evolved. These approaches are:

- coarsely-grained longitudinal case studies of growth and evolution,
- finely-grained case studies of *origin analysis* between consecutive versions of a system,
- case studies of code cloning within families of related systems, and
- tracking how build architectures and software manufacturing-related artifacts change over time.

Each of the above approaches involves three basic steps:

1. extraction of raw "facts" from various kinds of source artifact repositories,
2. automated, semi-automated, and manual analysis techniques performed on the facts, and
3. tool-supported exploration, navigation, and visualization to aid in comprehension.

We note that there is sufficient space in this position paper only to outline our results and methodologies. The reader is referred to listed references for more details.

2. Longitudinal case studies of growth

2.1. Goals and Results

Previously, we have studied growth and evolution of several open source software systems, including the Linux operating system [2]. Our original goal was to track how the growth patterns of large open source systems compared to previous studies on (non-open source) industrial systems; in particular, we wished to determine if Lehman's Laws of Evolution [8], which had been derived based on studies of (closed source) industrial software systems, also held for open source systems. We found that they did not hold in several instances; perhaps the most surprising result was that the Linux kernel continued to grow at a geometric rate even after surpassing two million lines of code (2 MLOC) (Fig.1). Lehman's empirical model predicts slowing growth as a software system becomes "large".[1]

2.2. Methodology

For these studies, we analyzed the source code of the Linux operating system in a semi-automated manner. This involved manually downloading and unpacking 96 versions of the kernel source code "tarball", running a set of hand-crafted `bash` and `awk` scripts over them to measure their

[1] Lehman has personally acknowledged that this study does indeed contradict some of his laws, and has said that he will need to reformulate them to take into account the various phenomena of open source software development.

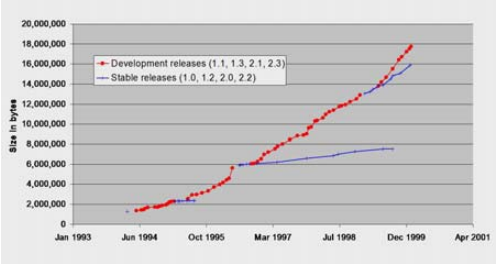

Figure 1. Growth of the compressed `tar` file distribution for the Linux kernel source release; measuring size as lines-of-code, number of modules, *etc*.showed roughly the same geometric growth pattern [2].

size in various ways and at various levels of abstraction, and then analyzing and exploring the results within a spreadsheet.

At the time the study was done, we did not have access to (or know of) a "live" CVS repository that could have aided in automating the analysis tasks. The size of the Linux kernel source itself (over 2 MLOC) and its relative fragility (it can be difficult to configure, build, and compile in an automated way) limited the amount and kinds of analysis we were able to perform. Its fragility meant that we could not reliably use our preferred static analysis tools to get a more sophisticated understanding of its complexity, and its size meant that we could not easily store more than a few unpacked kernels at a time on the file system. Effectively, we "boiled the ocean" of source code down to sets of numbers that could be stored inside a few spreadsheets, and used those as the basis for our analysis.

3. Case studies of *Origin Analysis*

3.1. Goals and Results

A particular problem for program comprehension is accurately modelling the *ontology* of a system's components. That is, the identity of a component is often equated with the name of the containing file or programming language entity (possibly together with its location within, say, a directory hierarchy). If a component is renamed or moved, it is considered that the old component has died and a new one has been born. However, as a system is redesigned and refactored over time, it is fairly common for components to be renamed, moved to another container, and merged into or split from other components. While the name/location-based identity assumption has the advantages of being simple and easy to implement in a tool, a lot of useful knowledge about the system can be lost if the system has undergone internal change and refactoring.

We have therefore sought to develop a set of techniques for performing what we call *origin analysis*, as well as a supporting tool called Beagle [11, 12]. That is, when confronted with a set of programming language entities that appear to be new to a particular version of a software system, we try to determine which of these components really *are* new and which are in fact derived from entities in the previous version of the system.

Origin analysis continues to be an ongoing area of research for us, but we have already performed two detailed case studies. The first explored the incidence of moving and renaming of functions within the GCC compiler suite [11]; Table 1 shows the results of applying origin analysis of the `parser` subsystem of version 1.0 of the EGCS variant of GCC. This subsystem does not exist in the previous evolutionary ancestor (GCC version 2.7.2.3), yet we were able to determine that approximately 46% of the 848 functions originated from various places within the ancestor. The second case study, which is not detailed here, examined the incidence of merging and splitting in the PostgreSQL relational database [12].

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

Table 1. Summary of origin analysis results for the (apparently) all new parser subsystem of the EGCS 1.0 compiler relative to its immediate evolutionary ancestor (GCC 2.7.2.3).

3.2. Methodology

Origin analysis is comprised of two basic techniques: *entity analysis* and *relationship analysis*. Entity analysis, which is similar to software clone detection, attempts to match entities of the two system versions based on similarity of the entities themselves. We have implemented this as a metrics-based "fingerprint" [7]. A set of metric values (*e.g.,* cyclomatic complexity, fan-in/out) for functions are

precomputed on system check-in by the commercial tool Understand for C++, and the results are stored in a relational database. At querying time, the candidates with the closest Euclidean distance to the metric tuple of the target entity are returned.

Relationship analysis is based on the assumption that if an entity is moved or renamed, there is a high likelihood that it will still engage in many of the same relationships with the same entities as before (*e.g.*, calls, called-by, inherits, uses-var). Various relations are extracted and recorded at system check-in by the cppx fact extractor for C/C++ systems [9], and stored in a relational database. At querying time, the candidates that have the most similar relational images (*e.g.*, call the same functions) are returned.[2]

Effectively, origin analysis reduces the source code down to sets of numerical values and abstracted facts about the software entities and their inter-relationships; we store this information in a relational database, and use a graphical tool to perform directed queries to help establish the most likely "origin" of software entities that appear to be "new".

4. Case studies of code cloning

4.1. Goals and Results

While algorithms and tools for code duplication detection (*i.e.*, "clone detection") have been well studied by the research community, there has been relatively little investigation into what types of clones might exist and how often and in what context they occur within industrial software systems. We feel that these questions are important as they can help us to develop criteria to evaluate the effectiveness of current detection techniques, and provide insight into how these tools might be improved.

Recently, we performed a case study on the incidence of code cloning within the file system component of the Linux operating system [6]. Our initial goal was to investigate how code duplication occurred in a well known industrial software system, and we began to classify the types of clones we found.

Our study led to several broad observations about cloning within the candidate system, and to hypotheses about cloning in general. For example, we found that files that belonged to the same subsystem (in this case, a particular file system implementation, such as `ext3`) often had many instances of cloning within them. Overall, we found that 78% of clone-pairs occurred within the same subsystem; this led us to hypothesize that some degree of system structure might be determinable based on the clone relationships. We also found that subsystems that share higher-than-average amounts of code duplication between them were often in a relationship of one being derived from the other, or the creation of one was heavily based on the other. In Figure 2, we have labelled three points in the graph where the subsystems were in such a relationship.

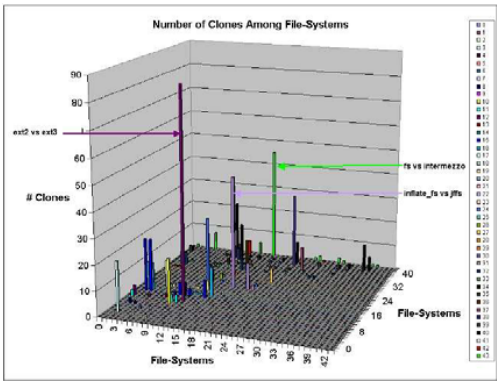

Figure 2. Number of clone pairs between file systems (excluding themselves).

We also observed that clones that existed between files in the same subsystem were usually function clones (according to the definition in the next section), whereas clones between files in different subsystems were usually not function clones. This is another indication that information about system structure may reside within the cloning relationship. We hypothesize that cloning activity may provide strong clues about file relationships.

Finally, we found that clones between files of different subsystems were often the "remains" of functions that had been cloned, but had been changed substantially enough that they could no longer be easily found as function clones. This insight has led us to consider a new vein of investigation; we plan to use change data from CVS repositories to profile how function clones change over time. Our analysis will address discovering which developers tend to introduce cloned code in a software system, who makes changes that drive them to become different, and if bug fixes are consistently made across all clone instances.

4.2. Methodology

To detect clones, we used both parameterized string matching [5, 4] and metrics-based string matching [1].[3] For brevity we will not describe these techniques here; the reader is referred to the cited references. After the initial extraction, we found that the tools returned a total of 5000

[2] Relationship analysis is also key to detecting merging and splitting, but the details are more subtle [12].

[3] We used the CCFinder tool to perform parameterized string matching [5]; we implemented our own metrics-based tool, following the design of others [1, 7].

clone pairs; of these, we determined through inspection that 1996 of these pairs were clearly false-positives and so were removed.

In the study, we identified five types of clones: *function clones*, *initialization clones*, *finalization clones*, *cloned blocks*, and *cloned blocks within the same function*. For the first four clone types, we further subdivided each group into clones in the *same file*, the *same subsystem*, and *different subsystems*. *Function clones* were functions where a minimum of 60% of the code of each function was duplicated between the two; a function clone can be composed of several smaller clone pairs. *Initialization clones* are pieces of source code at the beginning of a function that allocate space for and initialize variables; these clones must start in the first five lines of the function and end within the first half of the function. Analogously, *finalization clones* are pieces of source code which deallocate space and massage data for returning; these clones must start in the last half of the function and end in the last five lines of the function. *Cloned blocks of code* are segments of code that do not fall into any of the other types of clone

After categorization, and for any other empirical results we have presented, we performed manual inspection of a large percentage of the clone pairs in the given study to ensure that they were within the criteria that we specified and that they were accurately found as clones.

5. Longitudinal case studies of software manufacturing-related artifacts

5.1. Goals and Results

Software manufacturing — that is, the creation of software deliverables from source artifacts — is an important part of industrial software development. Large software systems often have complex subparts that engage in subtle relationships with the underlying technologies from which they are built; consequently, many such systems have complex and interesting architectural properties that can only be understood in the context of the various phases of system construction [10]. We consider that modelling and extracting build-time architectural properties of such systems are key to the software comprehension process, and so we have begun studying characteristics of development and maintenance activities that are related to software manufacturing (SM).

Recently, we have begun a project to study the maintenance effort of six open source projects[4] from a software manufacturing perspective (Table 2). We have attempted to measure the maintenance effort of SM-related artifacts for each project along three dimensions: the authorship, the size of the changes, and the frequency of the changes.

[4]We note that one of the systems studied —Apache Ant —also happens to be a system building tool.

Project	*Period studied from*	*# files*	*# CVS records*	*# authors*	*Build tools used*
midworld	05/2002	199	1,677	8	SCons
mycore	07/2001	343	2,116	12	Ant
Apache Ant	01/2000	1,791	28,888	32	Ant
kepler	08/2003	412	1,129	6	Ant + Make
PostgreSQL	07/1996	2,093	59,815	22	Make
GCC	08/1997	17,378	150,423	204	Make

Table 2. Case study project information.

The results of the study suggest that the development and maintenance of SM-related artifacts is a significant activity during software evolution. For example, we found overall that more than half of the project developers contributed changes to the SM-related artifacts (Table 3). This may indicate that changes to the source artifacts often require changes to the SM-related files. In all but one of the systems we studied, SM-related files changed much more than often than the other kinds of source files. Overall, changes to the SM-related files accounted for non-trivial percentages of total system changes, from 3–10% depending on the project.

Project	*# authors*	*# authors who changed SM artifacts*	*Percentage*
midworld	8	5	62.50%
mycore	12	7	58.33%
Apache Ant	32	26	81.25%
kepler	6	4	66.67%
PostgreSQL	22	15	68.18%
GCC	204	154	75.49%

Table 3. Author involvement in SM activities.

As more evidence of the significance of the evolution of SM-related artifacts, we found several versions of systems where insertions and deletions of lines in SM-related artifacts accounted for up to 30% of the total insertion and deletion of lines in the system (Table 4).

Project	*LOC of SM files / total LOC*	*Changed LOC in SM files / total changed LOC*	*Peak*
Apache Ant	0.70%	1.59%	11.92%
PostgreSQL	2.33%	3.70%	24.99%
GCC	4.64%	20.05%	30.24%

Table 4. LOC and changed LOC in SM files.

In an attempt to find causes of the changes to the build system, we calculated the correlation of number of changes to SM-related artifacts to the number of times the file count changed. For half of our case studies, we found correlation stronger than 0.7. We found that none of systems showed a strong correlation to the size of the changes in SM-related artifacts and change in file counts.

5.2. Methodology

For our study, we chose six open source software systems that used CVS as their versioning system. This allowed us to analyze the CVS logs — particularly of those artifacts that are related to SM — and perform statistical analysis on the data. The first step was to separate the SM-related artifacts from other source artifacts. The SM-related artifacts includes configuration scripts and system description files, such as `Makefiles`, Ant `build.xml` files, and `SConscripts`. We considered each CVS "commit" record to be one change, and the number of lines in each CVS commit to be the size of the change. By examining the size and frequency of changes in SM-related artifacts and source artifacts, we are able to address questions such as:

1. How much effort is put into SM-related artifacts?

2. What is the relation between evolution of SM-related artifacts and evolution software system as a whole?

A paper detailing the results of this study is under development.

6. Summary and challenges

This position paper has outlined four approaches to studying software systems using historical data extracted from various kinds of source artifacts. In each case, we performed automated analysis of the artifacts, then used various intermediate tools (such as `grok` [3], `awk`, and `bash` scripts) to created abstracted views that can be explored, navigated, and visualized for program comprehension purposes.

We conclude by noting that performing evolutionary studies of software systems presents several challenges that must be addressed by researchers in the field:

- *Scale* — In our experience, existing analysis tools often function "at the bleeding edge" of what is practical with respect to internal (and external) storage and processing. A typical static analysis performed on a single version of a system often produces voluminous detail and is computationally intensive; performing the same analysis across multiple versions of a system requires that scale issues be addressed seriously.

- *Automation* — Again, in our experience many software analysis tools require significant user intervention to extract accurate facts. This problem is exacerbated over multiple versions, and as new problems arise and have to be dealt with.

- *Artifact linkage and analysis granularity* — The canonical version control repository system, CVS, typically stores only source code, which it treats as plain text. A sophisticated tool for exploring software system evolution requires easy access not only to the "facts" and statistics about the source code that result from the analysis tools, but also to the source code entities themselves; CVS simply does not understand the notion of "method" or "function" embedded within a file.

References

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

[2] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proc. of 2000 Intl. Conference on Software Maintenance (ICSM-00)*, San Jose, California, October 2000.

[3] R. C. Holt. An introduction to the Grok language. Available at http://plg.uwaterloo.ca/~holt/papers/grok-intro.html, 2002.

[4] T. Kamiya, S. Kusumoto, and K. Inoue. A token-based code clone detection tool: CCFinder and its empirical evaluation. Technical report, Osaka University, 2000.

[5] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. In *Transactions on Software Engineering 8(7)*, pages 654–670. IEEE Computer Society Press, 2002.

[6] C. J. Kapser and M. W. Godfrey. Toward a taxonomy for source code cloning: A case study. In *Presented at First Intl. Workshop on Evolution of Large-scale Industrial Software Applications (ELISA-03)*, Amsterdam, September 2003.

[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] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution —The nineties view. In *Proc. of the Fourth Intl. Software Metrics Symposium (Metrics-97)*, Albuquerque, NM, 1997.

[9] A. Malton and T. Dean. The CPPX homepage: A fact extractor for C++. Website. http://www.swag.uwaterloo.ca/~cppx.

[10] Q. Tu and M. W. Godfrey. The build-time software architectural view. In *Proc. of 2001 Intl. Conference on Software Maintenance (ICSM-01)*, Florence, Italy, October 2001.

[11] 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.

[12] 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.