Tracking Structural Evolution using Origin Analysis

[Position Paper]

Michael Godfrey and Qiang Tu
Software Architecture Group (SWAG)
University of Waterloo
Department of Computer Science
Waterloo, Ontario, Canada N2L 3G1
{migod, qtu}@swag.uwaterloo.ca

ABSTRACT

Many long term studies of software evolution have made the simplifying assumption that the system's architecture and low-level structure is relatively stable. In our past work, we have found that this is often untrue; therefore, we have sought to investigate ways to detect and model structural change in software systems through a technique we call *origin analysis* [6] and supported a tool called Beagle [7]. In this position paper, we present a summary of our recent and ongoing work in this area, and we argue that more attention needs to be paid to techniques for understanding architectural and structural evolution of software systems.

Keywords

Software evolution, structural change, software architecture, supporting environments

1. INTRODUCTION

It is well accepted that software systems must change over time if they are to be successful [3]. As bugs are discovered, as new features are requested by users, and as the environment in which the software system itself operates evolves, so must a software system be able to adapt to these circumstances or risk losing user satisfaction.

Maintaining a good state of "health" — that is, keeping a system agile and adaptable for future desired change — is therefore a key goal for most software systems. Developers who maintain older systems must be able to understand what their system does and why it does it in the way it does. They must be able to understand the rationale for past architectural decisions, and have an appreciation for the software system's evolutionary history, especially at the architectural level.

1.1 The nature of software change

Some changes to software systems are mostly additive in nature; for example, new features maybe added to an existing functional infrastructure, or an entire new piece of infrastructure maybe added to support a new deployment environment. By definition, these changes require relatively little intervention in the existing codebase, and may be easily understood by a software maintainer trying to learn about the evolutionary history of the system. Other changes are more invasive but are still easily understood, such as small bugfixes that require only simple tweaks to the codebase. However, a large percentage of software change is highly *invasive*, in the sense that it involves refactoring, redesign, and rearchitecting of (parts of) the system's basic structure.

It is important that software maintainers be able to comprehend systems that have undergone substantial structural and architectural evolution, yet there is a dearth of tools that aid maintainers in understanding and reasoning about invasive change. Most program comprehension tools that are able to model evolving systems make the simplifying assumption that the system's structure — from its architecture to its low-level design — is relatively stable. At the other end of the granularity spectrum, most `diff`-like tools are very low level; they are intended mainly to aid developers in merging code chunk variants into a single coherent whole, rather than as an aid to understanding system evolution at higher levels of abstraction. Thus, there is a need to investigate tools and techniques that can aid in reasoning about how and why structural changes have occurred in software systems.

2. MODELLING STRUCTURAL CHANGE

A particular problem for program comprehension tools 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 to another container, it is considered that the old component had died and a new one has been born. While this approach has the advantages of begin simple and easy to implement, a lot of useful knowledge about the system can be lost; this is especially problematic when trying to build an accurate evolutionary history of a software system.

We have therefore sought to develop a set of techniques for performing what we call *origin analysis*. 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. We do this by performing a kind of directed clone detection that uses both established [2] and novel [6] techniques.

3. ORIGIN ANALYSIS

3.1 Goals for origin analysis

We have recently created a prototype of a tool called BEAGLE [7], which is intended to provide an extensible framework to aid software maintainers in comprehending the evolutionary history of a software system. In particular, BEAGLE supports the extraction of "facts" about a software system, permits graphical browsing and querying of a system's evolutionary history at various levels of abstraction, and the creation of new views and the addition of new information to evolutionary models.

As part of this work, we have examined the problem of how best to reason about structural change and how best to perform origin analysis on software components. Our current approach works at the level of functions and variables, but also allows for entity and relationship information to be automatically abstracted to the level of files and subsystems via the `grok` relational calculator that is part of the PBS toolkit [1].[1]

Our other goals for origin analysis include:

- The generation of "fingerprints" for program entities should be performed only once per version, when it is added into the BEAGLE repository.
- The comparison of two or more versions of a system should be computationally cheap to perform.
- The supporting environment should allow for different fingerprinting strategies to be used.
- The reasoning should be semi-automatic; users should be presented with several choices and be allowed to pick the most appropriate option.

So far we have chosen to limit ourselves to function definition boundaries, as "facts" about functions are easily derived using external source code analysis tools.

3.2 Definition of origin analysis

We define origin analysis as follows:

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

We note that while simple renaming and moving of entities are easy to define formally and fairly easy to detect, the idea that F *is a changed version of* G 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 F is a changed version of G.

[1] The user must provide a subsystem hierarchy model for the system; the dependencies are then inferred to the level of containers, supercontainers, *etc.* automatically. The PBS visualization engine can be used to navigate the system structure and query the (original and derived) information at various levels of abstraction.

3.3 Origin analysis and clone detection

Origin analysis borrows some techniques from software clone detection; however, 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. As such, it is typical to compare all program entities of the system (function, classes, and/or files) to each other. However, if an expensive comparison technique, such as comparing ASTs, is used then the naive approach of comparing every entity to every other entity is impractical on large systems. Consequently, it is common to perform some sort of hashing-type calculation to identify "likely suspects", and apply the expensive matching algorithm only in those cases.

With origin analysis, it is important to use computationally inexpensive techniques and to pre-compute as much as possible, since users will typically be examining several versions of systems with many subpieces. Fortunately, origin analysis usually entails examining only a small subset of a system's entities, *i.e.*, those that did not "exist" in the previous version. Furthermore, the most obvious candidates to be the "origin" of these new entities are those that appear to have died out in the previous version, and this is also a relatively small set of the system's entities.[2] Thus, although our approach may initially seem expensive, in fact most of the computation involves a relatively small number of software entities being compared to each other.

Our approach to origin analysis has two main phases: entity analysis and relationship analysis. Entity and relationship information about a single version of a system is extracted and computed once, when the version is checked into the BEAGLE repository. Subsequent comparisons requested by users are performed by contrasting the various results between entities in the new version versus entities in the old version.

3.3.1 Entity analysis

Entity analysis generates a kind of fingerprint for each program entity; the current implementation of BEAGLE considers only function entities as we have exploited the use of third-party analysis tools, including `cppx` [4] and Understand for C++ [5]. For each function in each version of the system, we computed basic metrics such as lines of code, lines of comments, cyclomatic complexity, code nesting, fan-in, fan-out, number of global variables accessed and updated, number of parameters, number of local variables, the number of input/output statements. In addition, we compute the following composite metrics: S-complexity, D-complexity, Albrecht's metric, and Kafura's metric; we provide a fast comparison between two entities by borrowing the approach of Kontogiannis *et al.* [2] of taking the Euclidean distance of the 5-tuples consisting of the four composite metrics mentioned above plus cyclomatic complexity. Since the metric values for each system version are computed only once and the values stored in the BEAGLE repository, comparing two entities (in different versions) consists of a simple database query and a cheap numerical calculation.

All of the information derived about the entities is stored in the BEAGLE repository. When a user asks for a comparison between two entities, the computed values are used to by the system to sug-

[2] It is also possible that code cloning may have occurred; that is, the "parent" of the new entity may have another legitimate "heir", and thus may not be considered using our current implementation of origin analysis. We intend to address this possibility in the future, but we consider that it is reasonable to ignore it in most cases.

gest a set of "most likely suspects" based on the Euclidean distance described above. However, a user can request a summary of the various pre-computed metrics values (as well as looking at code `diff`s) when trying to make a decision. Again, we emphasize that we consider that to be effective, origin analysis should be a semi-automated technique.

3.3.2 Relationship analysis

Relationship analysis is based on the idea that if an entity changes its name, there is still a high likelihood that its relationships to other entities in the system (*e.g.*, the functions it calls or is called by, the files it includes, the global variables it uses, *etc.*) will not change. Consequently, comparing the pre- and post- relational images of various relations with respect to a "new" entity might further aid in determining its origin.

More formally, suppose that F is a "new" function in V_{new}. Then for each G that is a function in V_{orig} but not in V_{new} and for each relationship R that is supported by BEAGLE[3], we compute the following two sets:

$$\begin{aligned} Pre_R(G,V) &= \{H \in V \mid R(H,G)\} \\ Post_R(G,V) &= \{H \in V \mid R(G,H)\} \end{aligned}$$

That is, if R is the *calls* relationship, then $Pre_{calls}(G, V_{orig})$ is the set of all functions that call G in version V_{orig} and $Post_{calls}(G, V_{orig})$ is the set of all functions that G calls in version V_{orig} . We then compare these sets with $Pre_{calls}(F, V_{new})$ and $Post_{calls}(F, V_{new})$ to see how much overlap there is; if F is simply a renamed G, then the sets should be very similar.[4]

3.3.3 Summary

Once the results of the entity and relationship analysis have been computed, this information may be combined with a comparison of the names of the entities. As yet, no automatic string-based name ranking is performed by the BEAGLE system, but this is projected as future work. Currently, the names of the various candidates are shown together with the their ranking according to the results of entity and relationship analysis. The user can then browse the code of the various candidates, perform a `diff` on them, and request details of the precomputed values of the metrics.

4. CURRENT WORK, OPEN QUESTIONS

In a previous case study [7], we found the BEAGLE tool and its implementation of origin analysis to be very useful in building an evolutionary history of the GCC compiler suite. We extracted full information about 29 versions of GCC, and we have performed a detailed analysis of the difference between several pairs of versions, including GCC version 2.7.2.3 and EGCS version 1.0. The latter version represents a major redesign of the former, including a new naming convention that made most of its entities appear to be "new". Table 1 shows a summary of the results of performing origin analysis on the parser subsystem of EGCS; after detailed examination using BEAGLE, we concluded that out of its 848 functions, 460 were truly "new" to EGCS and the remainder were very similar to functions in the older version.[5]

[3] Currently, only function calling is modelled.

[4] It should be noted that the members of $Pre_{calls}(G, V_{orig})$ and $Post_{calls}(G, V_{orig})$ might themselves "change" between versions, and thus appropriate substitutions may be required. BEAGLE does not yet provide automated support for this.

[5] EGCS incorporated support for then-recent changes to C++ as a result of the ANSI standardization; it was therefore not surprising that there should be a lot of change to the parser subsystem of EGCS relative to its immediate ancestor.

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

Table 1: Summary of origin analysis results for the parser subsystem of EGCS 1.0 compared to its immediate predecessor (GCC 2.7.2.3).

Although we consider that our preliminary case study in using BEAGLE was successful, we have not yet systematically or comparatively validated the usefulness of our particular techniques. Many open questions remain.

Our current implementation considers only entity and relationship information about functions; information about files and subsystems is inferred and abstracted from the information about functions they (transitively) contain.

The particular metrics we decided to model — lines of code, cyclomatic complexity, fan-in *etc.*— were chosen because the subtools we had access to supported them. The use of Euclidean distance of five entity characteristics is borrowed from other research [2]; we have not conducted experiments to see if other approaches may yield more accurate results.

5. REFERENCES

[1] P. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Müller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4), November 1997.

[2] 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.

[3] 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.

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

[5] ScientificToolworks. Understand for C++. Website. http://www.scitools.com/ucpp.html.

[6] Q. Tu. On navigation and analysis of software architecture evolution. Master's thesis, University of Waterloo, 1992.

[7] 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. (To appear).