A Hybrid Process for Recovering Software Architecture

Vassilios Tzerpos and R.C. Holt
University of Toronto
Toronto, Ontario, Canada
{vtzer,holt}@cs.toronto.edu

Abstract

A large portion of the software used in industry today is legacy software. Legacy systems often evolve into difficult to maintain systems whose original design has been lost or else no longer closely matches the actual structure of the system.

In our paper[1] we present a "hybrid" process in which we combine extracted code facts and information derived from interviewing developers to determine the architectural structure of a legacy system. We introduce the steps of this process using a case study of a large legacy system, an optimizing back end for IBM compilers. These steps include collecting "back of the envelope" designs from project personnel, extracting raw facts from the source code, collecting naming conventions for files, clustering code artifacts based on naming conventions, creating tentative structural diagrams, and collecting more "live" information in terms of reactions to these tentative diagrams, and so on, until we converge to an architectural structure.

Our conclusion is that there is a reasonably well defined sequence of steps that the reverse engineer can go through to extract facts and collect live information in order to create a useful architectural structure diagram.

1 Introduction

Legacy software systems are systems that were designed and developed many years ago. Too often a legacy system evolves into an unmaintainable system whose structure has a little to do with its original conception or the structure described in the system's documentation. It commonly becomes necessary to "reverse engineer" such systems in order to move them to a new platform or to write them in a new programming language.

In this paper we present a "hybrid" process for deriving the architectural structure of a legacy system. We combine extracted code facts with information obtained from the developers in order to determine the architectural structure of a legacy system. The steps of this process are introduced using a case study of a large legacy system of roughly a quarter million lines of code, called Tobey, which is an optimizing "back end" for IBM compilers.

We propose a reasonably well defined set of steps that a reverse engineering project can follow in order to create a useful architectural structure diagram by combining mechanically extracted facts and "live" information.

The rest of the paper is organized as follows. Section 2 gives background on reverse engineering and presents the problem addressed in this paper in greater detail. Section 3 describes the steps we took in our effort to extract the architectural structure of Tobey. Section 4 gives more detail about our "hybrid" approach. Finally, Section 5 concludes the paper.

2 Background

Chikofsky [5] states that "reverse engineering is the process of analyzing a subject system to identify its components and their interrelationships and create representations of the system

[1] This research was, in part, supported by IBM Canada Ltd.
The IBM comtact for this paper is Jacob Slonim, Centre for Advanced Studies, 2G/894, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1H7.

in another form or at a higher level of abstraction".

All too often, when dealing with a legacy system, there is only a modest amount of system documentation and the original developers are no longer accessible. In such cases, the available information about the system's structure lies primarily in the system's actual code or in the heads of the people responsible for maintaining the system.

Many different solutions to the problem of reverse engineering have been suggested [4, 1, 9, 6, 7]. These can be categorized as either top-down or bottom-up approaches.

Working in a top-down fashion, one attempts to recover the overall structure of the system from the people involved in it by interviewing them. The information they know about the system is called "live" documentation. Then, one attempts to match this structure onto the actual implementation.

Conversely, a bottom-up approach extracts low-level facts from the source code by running special programs that determine, for example, call graphs and accesses to global variables, and attempts to deduce the higher-level structure from these facts.

These approaches have limitations that do not allow them to scale up well to deal with large industrial systems that have evolved over a significant time period. As will be discussed later in more detail, we propose a hybrid approach that alleviates many of the problems of the two classic methods.

3 The Tobey story

Tobey is the optimizing back end for IBM compilers. It is a fairly large system that has been constantly under development for more than a decade. Its original developers are no longer associated with it, and new people frequently join the project. The system's architectural structure has evolved significantly since its original conception. As a result, Tobey is an appropriate candidate for a reverse engineering project.

The source code for Tobey is a collection of approximately 1200 source files, stored in two directories.

As has been shown in [10], it is reasonable to assume that files are meaningful clusters for system resources such as procedures and global variables, since developers will generally place related resources in the same file. Therefore, we concentrated on clustering these files into meaningful groups (called subsystems).

The source file inclusion view is sometimes a good basis for extracting the structure of C systems [2], so we tried that first. However, this effort failed to produce interesting results for Tobey, due to the particular inclusion scheme of PLIX (the programming language Tobey is written in).

The next step was to look at Tobey's procedure call graph. That information was extracted from the cross-referencing information provided by the PLIX compiler. Dependencies among source files were induced by the call dependencies of the procedures they contain. Unfortunately, with over a thousand files, we were not able to create a meaningful visualization of call dependencies.

We next attempted to partition the Tobey files into a meaningful set of subsystems,using two different approaches.

The first approach was based on the naming convention (not uniformingly used across the system) for the source file names. In most cases, the first two letters of the file name indicated the subsystem it belongs to. This provided us with a useful, though crude, decomposition of the system.

We attempted to improve on this decomposition by applying a second and more graph-theoretical approach. Based on the assumption that subsystems in Tobey usually have a single point of entry, we identified collections of source files with the following property. If S is the set of the files of the collection, then all calls from files outside S to files in S are calls to only one file in S (the entry point). We say that this one file "covers" the files in the collection. We developed software to automatically find the collection of files covered by a given file.

Based on naming conventions and file covering, we created a second approximation of the system's structure, and were able to visualize it as shown in Figure 1. Although this diagram provides useful information about the subsystems and the interactions between them, it can

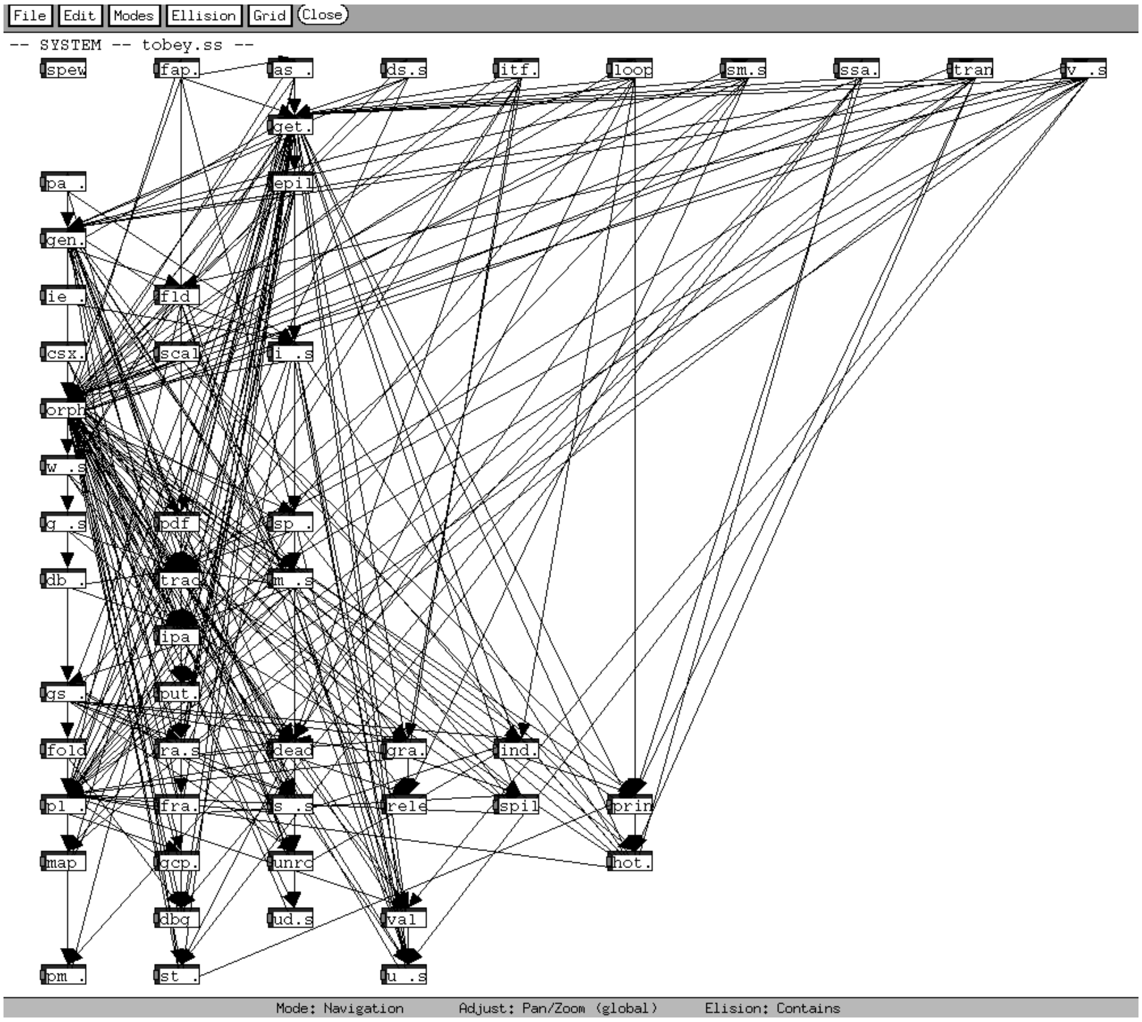

Figure 1: A first visualization of Tobey's architectural structure.

be improved in two ways. First, the layout can be manually manipulated so that it resembles more closely the mental image developers have about their system. Second, there were about 300 source files that we were not able to place in a subsystem using the two automatic methods described above. In Figure 1, they belong to a subsystem called "orphans" (first column, seventh row). These files need to be moved to the appropriate subsystem.

In order to deal with the "orphan adoption" task, we consulted the developers, because they are the source for the semantic knowledge of the system's structure, that is necessary in order to move each orphan file into an appropriate subsystem. We used the Landscape Viewer [3] to manipulate our tentative version of the architectural structure (Figure 1) to incorporate input from the developers.

Coming up with a "meaningful" layout is always a tricky task, since "meaningful" is a subjective notion. However, after a couple of iterations we managed to achieve the picture shown in Figure 2. In this figure, we have a meaningful decomposition for Tobey as well as a layout that decomposes Tobey even further into four big subsystems (its three major phases and a collection of service modules). The layout reflects the developers' convention that left-to-right positioning generally corresponds to the

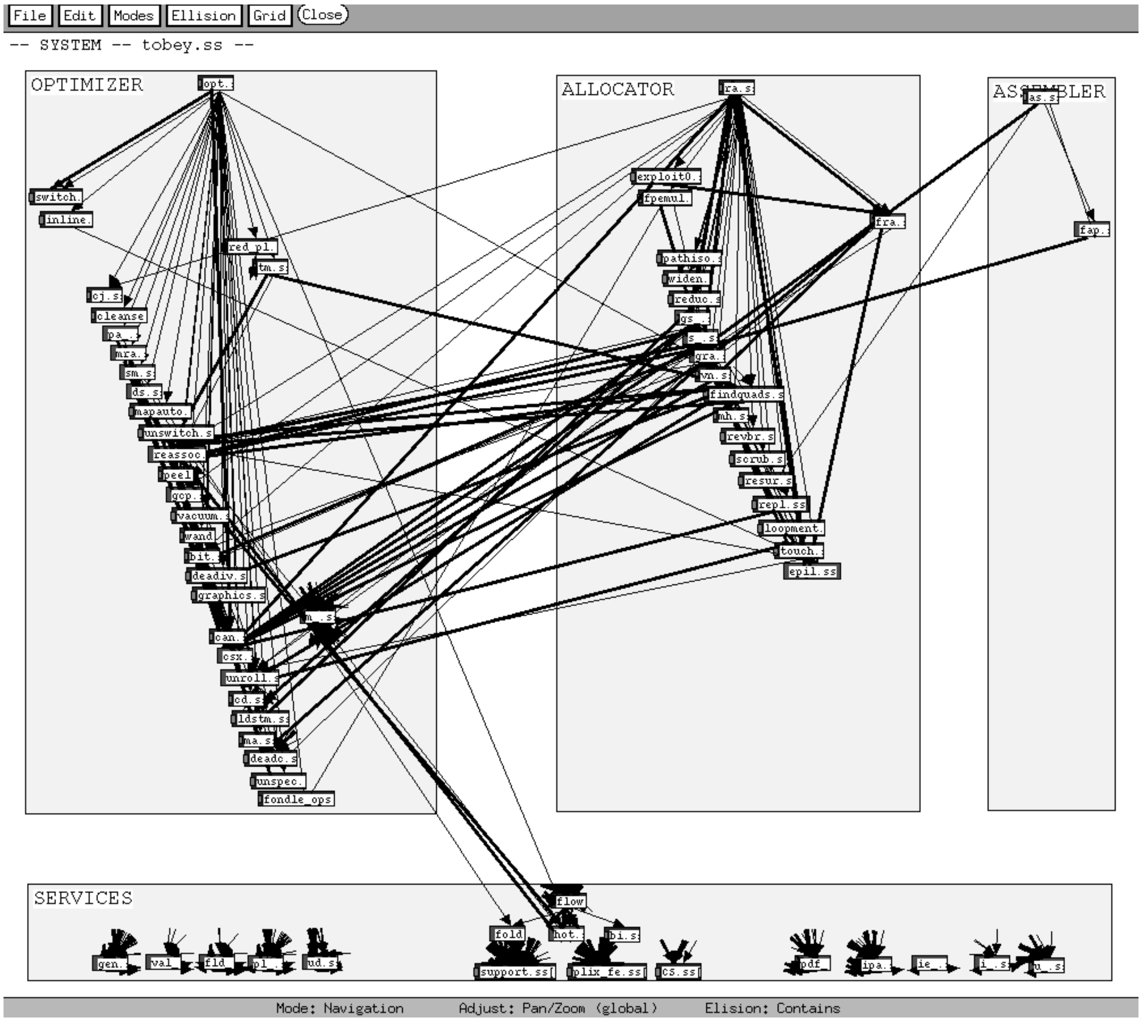

Figure 2: The evolved visualization of the architectural structure of Tobey.

order in which these get executed (this cannot be absolutely true since several subsystems are executed more than once at different times during the execution of Tobey). In achieving the final visualization, we developed a new facility of the Star tool [8] (used as a front end for the Landscape Viewer) that preserves the layout positions from previous versions as much as possible.

When the developers were presented with Figure 2 they thought it closely resembled their mental image of the system's structure and felt that it could become a valuable piece of the documentation of the system.

4 The hybrid approach

Based on our experience with Tobey, we believe that there exists a reasonably well defined set of steps that the reverse engineer can follow to extract facts and collect live information in order to create a useful architectural structure diagram.

These steps are not necessarilyy sequential. Some of them can be executed in parallel or in a different order as seems appropriate. Furthermore, these steps can be executed at more than one level of granularity, for example, they can be used to determine the structure of a subsystem.

The steps are:

- *Choose domain model.* An appropriate domain model has to be chosen, so that it fits the requirements of the system in question and produces meaningful visualizations. For example, inclusion relations among source files was shown to be an inappropriate domain model for Tobey, while procedure calls produced a much better result.

  The appropriate domain model can vary significantly between different systems. An object-oriented system, for example, would require a different domain model from a procedural one.

- *Extract facts from source code.* Mechanically extracted facts must be acquired next. Facts can be extracted from the source code or from other sources, such as cross-referencing listings from the compiler.

- *Cluster into subsystems.* An automatic or "semi-automatic" attempt at clustering must be performed. This can be based on known algorithms for automatic clustering, or can rely on other information, such as naming conventions or directory structure.

- *Refine clustering using "live" information.* The architectural structure obtained in the two previous steps has to be confirmed with the developers. It then has to be manually modified so that it more closely matches reality. More than one iteration of this step may be required.

- *Refine layout using "live" information.* Either in conjunction with the previous step, or separately, an attempt should be made to visualize the system's structure in a meaningful fashion. Automatic layout algorithms do not generally use semantic information, so manual layout is necessary in order for the picture to convey this semantic information.

It is our belief that a reverse engineering project that follows these steps will be to a large extent successful in creating a useful architectural visualization.

5 Conclusions

We have presented a hybrid approach to reverse engineering legacy systems. Our approach is a combination of the classic top-down and bottom-up approaches. It utilizes mechanically extracted facts as well as "live" information from the developers to repeatedly refine the system's architectural structure diagram.

Our approach is based on experience with a large industrial system.

Acknowledgements

The developers of the Tobey project have been extremely helpful in this research, especially Bill Hay, Joanne Minnish, Ian McIntosh and Andrew McLeod.

The Rigi group at University of Victoria handled the fact extraction for the PLIX system.

About the Authors

Vassilios Tzerpos is a Ph.D. student in the Department of Computer Science at the University of Toronto. His research interests include software architecture, reverse engineering and software maintenance.

Tzerpos received his B.Sc. degree from the Electrical and Computer Engineering Department of the National Technical University of Athens, Greece in 1992 and his M.Sc. in computer science from the Department of Computer Science of the University of Toronto in 1995.

R. C. Holt is a professor at the University of Toronto in the the Department of Computer Science. Holt received his Ph.D. degree in Computer Science from Cornell University in 1971. His research has included work in operating systems, compiler development, and software construction methods. He is an author of the Turing programming language. His recent work has concentrated on Software Landscapes, which provide a visual formalism for software architectures. (See WWW http://www.turing.toronto.edu.)

References

[1] Canfora G. Antonini P. and Cimitile A.

"Reengineering legacy systems to meet quality requirements : An experience report". *International Conference on Software Maintenance*, pages 146–153, 1994.

[2] Tzerpos V. Carmichael I. and Holt R.C. "Design Maintenance : Unexpected Architectural Interactions". *International Conference on Software Maintenance*, pages 134–137, October 1995.

[3] Penny D.A. "The Software Landscape: A Visual Formalism for Programming-in-the-Large". *Ph.D. Thesis, Department of Computer Science, University of Toronto*, 1992.

[4] Buss E. and Henshaw J. "A software reverse engineering experience". *CASCON 91*, pages 55–73, October 1991.

[5] Chikofsky E. and J. Cross. "Reverse Engineering and Design Recovery: A Taxonomy". *IEEE Software*, pages 13–17, January 1990.

[6] Gannod G.C. and Cheng B.H.C. "Strongest postcondition semantics as the formal basis for reverse engineering". *Working Conference on Reverse Engineering*, pages 188–197, 1995.

[7] Reubenstein H.B. Harris D.R. and Yeh A.S. "Reverse engineering to the architectural level". *International Conference on Software Engineering*, pages 186–195, April 1995.

[8] Godfrey M.W. Mancoridis S., Holt R.C. " A Program Understanding Environment Based on the "Star" Approach to Tool Integration". *ACM CSC*, 1994.

[9] Choi S.C. and Scacchi W. "Extracting and restructuring the design of large systems". *IEEE Software*, pages 66–71, January 1990.

[10] Tzerpos V. "Visualizing the source file structure of software written in C". *Master's thesis, Department of Computer Science, University of Toronto*, January 1995.