GASE: Visualizing Software Evolution-in-the-Large

Ric Holt and Jason Y. Pak
Computer Systems Research Institute
University of Toronto
Toronto, Ontario, Canada
{holt,pak}@cs.toronto.edu

Abstract

Large and long lived software systems, sometimes called legacy systems, must evolve if they are to remain useful. Too often, it is difficult to control or to understand this evolution. This paper presents an approach to visualizing software structural change. A visualization tool, called GASE (Graphical Analyzer for Software Evolution), has been used to elucidate the architectural changes in a sequence of eleven revisions of an eighty thousand line industrial software system.

1 Introduction

A considerable amount of research on "programming in the large" [1] has focused on the study of software architecture [4, 8]. However, not much research has concentrated on understanding changes (evolution) of software architecture [2]. Since large software systems inevitably evolve over time, and tend to deteriorate structurally [5], we need better means of visualizing this kind of change.

This paper presents the **GASE (Graphical Analyzer** for **Software Evolution)** tool. This tool inputs descriptions of successive versions of a legacy software system and produces diagrams that highlight architectural changes.

This paper is organized as follows. First we describe the organization of the GASE tool. Next we describe our scheme for using colors to highlight architectural change. Then we describe an experiment in which we use GASE to visualize changes in the architecture of a large industrial software system. We first show how a single version of the software can be viewed and then give diagrams that contrast two quite similar and two quite different pairs of versions. We conclude by giving observations from our experience in using GASE to analyse this industrial software system.

2 Organization of GASE

The GASE tool consists of four major phases: an Extractor, an Analyzer, a Mapper and a Viewer (see Figure 1). This organization, as well as much of GASE's actual code, is based on Landscape software [9, 6].

The Extractor parses the source code for target system that is to be analyzed, and determines the relevant "facts" about that system. These facts typically include the set of modules and subsystems in the target system as well as relations such as "calls" or "includes" among these. Since we are only interested in the large scale structure of the target system, we ignore lower level entities such as statements and variables.

The Analyzer consumes the Extractor's output for one or more versions of the system. In the common case, the Analyzer inputs facts about the parts of two versions of a system; let us call these two versions V1 and V2. From a graph theory point of view, the parts are either nodes (modules or subsystems) or edges (relations such as "call"). Each part is flagged as (a)

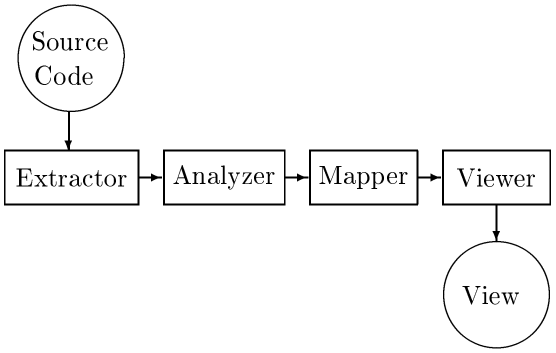

Figure 1: Organization of the GASE Tool

unique to V1 (eliminated in V2), (b) common to V1 and V2, or (c) unique to V2 (did not exist in V1).

The Mapper translates the flagged facts into a corresponding colored diagram, using boxes for nodes and arrows for edges. This translation includes laying out the boxes in a way that attempts to minimize edge crossings [10].

The Viewer provides a convenient graphical user interface that is used interactively to select versions and version pairs to be inspected. The Viewer uses colors to highlight changes from version to version. The Viewer supports a rich set of navigational and querying facilities including zooming, panning, opening and exploding of (sub)views [6].

3 Pair-Difference Coloring

GASE uses color to show changes in architectural structure. It uses **red** to represent recent change, **grey** for intermediate status, and **blue** for older change. In accordance with accepted graphics practice, we use red for "hotter" or recent activity and blue for "colder" or older activity.

In this paper, we will present a scheme for contrasting pairs of versions, in which we use only three shades (red, grey and blue), but in principle the scheme can be extended to n versions displayed by a "spectrum" of shades running from red through grey to blue [7].

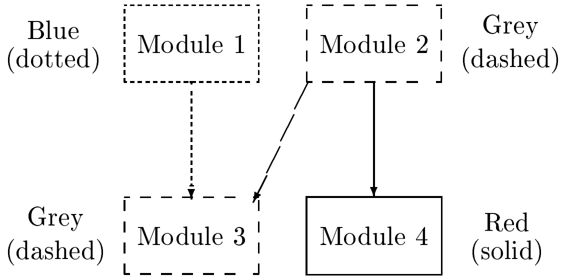

Figure 2: The Part-Difference Coloring Scheme

Figure 2 gives a simple example in which version V1 of system has evolved to version V2. Since the diagrams given here are not rendered in color, we are using **solid** lines for red, **dashed** lines for grey, **dotted** lines for blue. Blue (dotted) parts (boxes and arrows) exist in V1 but were deleted in V2. Grey (dashed) parts are common to both V1 and V2. Red (solid) parts did not exist in V1, and were added in V2. We call this the **Pair-Difference** coloring scheme because, for a pair of versions, it highlights the parts that are either common among or unique to the versions.

In Figure 2, Module 1 and edge (1,3) are blue (dotted), meaning they existed in V1 but were deleted in V2. Modules 2 and 3 and edge (2,3) are grey (dashed), meaning they exist in both V1 and V2. Module 4 and edge (2,4) are red (solid), meaning that they did not exist in V1, and were created in V2.

4 An Experiment with Industrial Software

We were fortunate to have access to the architectural facts (subsystems and relations) for a sequence of eleven versions, which we will call V1 to V11, of an industrial software system [3]. The system is a commercial configuration management system, which is used by software develop-

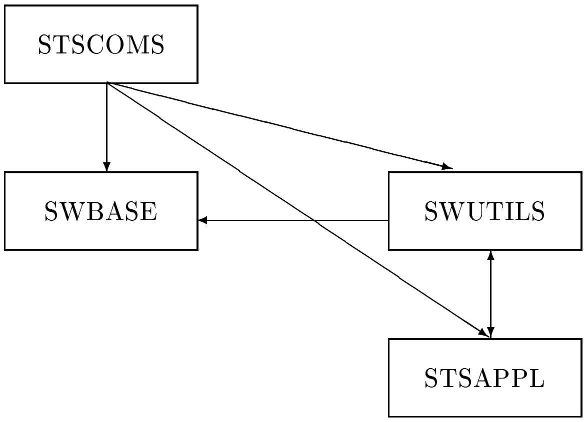

Figure 3: Version 1 of the Industrial System

Figure 4: Versions 1 and 2 Showing Common Dependencies

ers to keep track of their software artifacts. The system, developed over a period of four years, is written in the C programming language. A typical release contains roughly 80,000 lines of code.

4.1 Viewing a Single Version

Figure 3 shows the first version, V1, of the industrial system. The figure shows that the system consists of four subsystems, connected by static dependency relations. The relations correspond to calls or accesses to global variables. Since only one version is shown, we do not need to use colors (or dashes and dots). Each subsystem is shown "closed" meaning that its contents are not shown. GASE provides the user with GUI facilities to "open" each subsystem to reveal its contained modules along with their interdependencies.

4.2 Contrasting Two Similar Versions

Figure 4 is a GASE screen dump showing both versions V1 and V2 of the industrial system. In this figure, all subsystems are "open", so we can

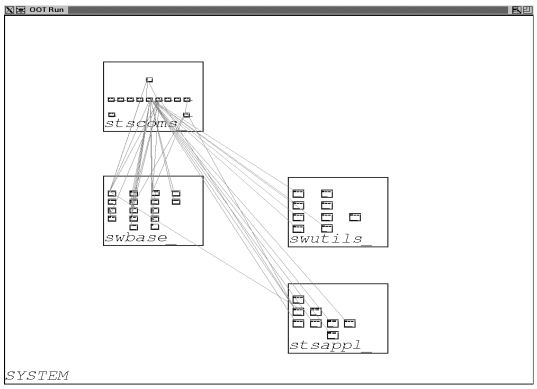

Figure 5: Versions 1 and 2 Showing New Dependencies

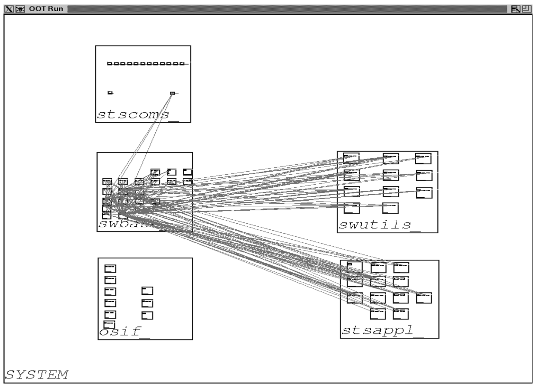

Figure 6: Versions 10 and 11 Showing Old Dependencies

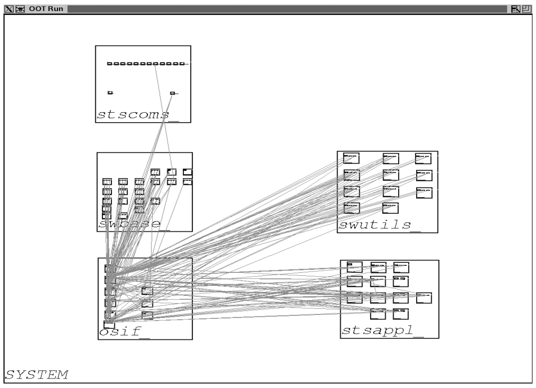

Figure 7: Versions 10 and 11 Showing New Dependencies

see the modules within them.

If we had colors available, the figure would use red, grey and blue to show new, common and deleted parts. It turns out that from version V1 to V2, the subsystem structure remained constant and no modules were deleted, but some modules were created. As a result, some modules (the new ones) should be shown in red and the rest in grey. Instead of using color, we use two figures to selectively show common edges (in Figure 4) and new edges (in Figure 5). As can be seen, most edges are common (Figure 4), and a fewer number are new (Figure 5).

From Figures 4 and 5, it is easy to see that the architectural structures of V1 and V2 are quite similar, and it is easy to see the new dependencies.

4.3 Viewing a Major Architectural Change

From Figure 4, we can directly observe that the high level structure, at the level of subsystems, of the industrial system did not change from version V1 to V2. From using GASE, we observed that the high level structure remained constant until V11. GASE views contrasting V10 and V11 (Figures 6 and 7) show that there is a significant change. Figures 6 and 7 show five subsets instead of the previous four. Careful inspection suggests that one subsystem, SWBASE, was split into parts called SWBASE (again) and OSIF.

In discussions with the developers of the industrial system we learned that in V11 they restructured their system to isolate the operating system interface in a new subsystem called OSIF. The GASE diagrams neatly portray this evolution. Figure 6 contrasts V10 and V11 by showing the deleted (blue) dependencies while Figure 7 shows the added (red) dependencies. (In a sense these figures show too many changed dependencies in that a dependency is flagged as changed if it occurs in a new subsystem, even though it still connects the same modules.)

5 Observations

In our experiment we made a number of observations, some of which we had not expected.

The most gratifying was that the significant restructuring from V10 to V11 was immediately obvious from the GASE views, and correlated with the developers' description of this architectural change.

Another observation was that most changes occurred at a low level (among modules) and not at a high level (among subsystems). This phenomena suggests a "software rule" that predicts that the rate of change is proportional to structural depth. Of course our simple experiment simply hints at the existence of such a "rule". (It may be that metrics researchers may want to try to validate this rule by tabulating rates of architectural change versus depth.) The interesting part from our point of view is that GASE views of successive versions immediately made clear that this particular industrial system follows this rule.

Another observation was that the industrial system consistently grew and almost never shrank. This phenomena was immediately obvious from the GASE views because boxes were commonly red (new) but almost never blue (deleted).

A final observation was made by a developer, who noticed from looking at a view containing V11 that the OSIF subsystem depends upon another subsystem. To him, this was an unexpected and unintended dependency, because OSIF was intended to be a reusable subsystem, independent of other subsystems.

6 Conclusions

We developed the GASE tool to explore an approach to visualizing software evolution. This approach uses colors to contrast new, common and deleted parts of a software system. With this scheme, the developer can easily see structural change that might otherwise be difficult to discern.

We tried out GASE on an 80,000 line industrial system that had a history of 11 versions. GASE views of this history made possible a number of observations about the industrial system. These observations included the following. The industrial system underwent only one major restructuring, which was immediately obvious from GASE views. The views made clear (a) that the system changed mostly at low levels and (b) that it grew but rarely shrank. The GASE view following the major architectural change revealed a dependency that the developer had not intended.

From our experience using the GASE tool, we feel that this approach has considerable promise for helping developers understand and control architectural change.

References

[1] DeRemer, Frank and Kron, Hans H., *"Programming in the Large Versus Programming in the Small"*, IEEE Trans. on Software Engineering, Vol. SE-2 No. 2, June 1976.

[2] Eick, S.G., Steffen, J.L., Summer, E.E. Jr., *"Seesoft – a tool for visualizing line oriented software statistics"*, IEEE Trans. on Software Engineering, Vol. 18, Nov 1992, pp. 957-968.

[3] Farah, J., Private correspondance, Dec 22, 1995.

[4] Garlan, David and Shaw, Mary, *"An Introduction to Software Architecture"*, *Advances in Software Engineering and Knowledge Engineering*, World Scientific Publishing Company, Vol 1, 1993.

[5] Lehman, M.M. and Belady, L., *"Program Evolution. Process of Software Change"*, London: Academic Press, 1985.

[6] Holt, R.C. and Mancoridis, S., *"A framework for Specifying and Visualizing Architectural Designs"*, Tech. Report #300, Computer Systems Research Institute, University of Toronto, 1994.

[7] Pak, J.Y., *"Towards Visualizing the Evolution of Software Architecture"*, Master's thesis, Department of Computer Science, University of Toronto, 1996.

[8] Perry, D.E. and Wolf A. L., *"Foundations for the Study of Software Architecture"*, ACM SIGSOFT, Software Engineering Notes, Vol. 17 No. 4, Oct 1992, pp. 40-52

[9] 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.

[10] Sugiyama, K., Tagawa, S. and Toda, M., *"Methods for Visual Understanding of Hierarchical Systems"*, IEEE Trans. on Systems, Man and Cybernetics, Vol. SMC-11, No. 2, 1991, pp. 109-125