Dynamic Analysis Techniques for the Reconstruction of Architectural Views

Bas Cornelissen
Software Evolution Research Lab
Delft University of Technology
The Netherlands
s.g.m.cornelissen@tudelft.nl

Abstract

Gaining an understanding of software systems is an important discipline in many software engineering contexts. It is essential that software engineers are assisted as much as possible during this task, e.g., by using tools and techniques that provide architectural views on the software at hand.

This Ph.D. research addresses this issue by employing dynamic analysis for the reconstruction of such views from running systems. The aim is to devise new abstraction techniques and novel visualizations, to combine them, and to assess the benefits through substantial case studies and controlled experiments. This paper describes our approach, reports on the results thus far, and outlines our future steps.

1. Introduction

Program comprehension is an important discipline that has been subject to research for a long time. It is a process of which perhaps the most well-known purpose is the facilitation of software maintenance tasks: when modifications to a software system are necessary, the engineer must first familiarize himself with (part of) the system at hand. However, as a software system evolves over time, its architectural documentation (if existent at all) often becomes outdated, which leaves the engineer with no choice than to study such artifacts as source code and system behavior.

Unfortunately, this is a very time-consuming activity and Corbi has reported [4] that up to 50% of maintenance efforts is spent on gaining a sufficient understanding of the system. The use of specialized tools and techniques that provide comprehensible *views* [3] of the software can significantly reduce this effort, particularly when dealing with complex systems. Much attention in this context has been given to the reconstruction of *static* architectural views such as the module viewtype (e.g., [1]), which comprises the visualization of a system's structural information.

1.1. Dynamic Analysis

The component & connector (c&c) viewtype, on the other hand, is based on *dynamic analysis*, which is the analysis of data gathered from a running program. The system is executed according to a certain scenario and, in the process, various runtime data are collected that are subject to (post-mortem) analysis. Effective visualizations of the resulting execution traces provide an insight into the program's inner workings, particularly in terms of its behavior. Through the use of suitable abstractions, the gathered data may be lifted to an architectural level, thus leading to c&c views.

The principal added value over static approaches is the revelation of detailed object interactions and late binding, thus allowing for accurate pictures of a software system. However, these details come at a price: while being dependent on the level of information that is collected, the massive amounts of data that often result from dynamic analyses call for smart abstractions and visualizations [15].

While several common visualization techniques do exist, they typically do not suffice when the system at hand is large and complex. In general, a visualization technique faces a twofold challenge: it must depict (1) structural information and (2) large amounts of runtime information without confusing the viewer, e.g., without the need for excessive scrolling. UML sequence diagrams, for example, present sequences of events in a chronological and therefore intuitive fashion, but tend to become increasingly unreadable when handling large systems or long execution traces. Moreover, the abstractions that are being offered are typically manual in nature (e.g., [13]).

1.2. Research questions

Having illustrated the problems and issues that we are facing, we now formulate four research questions to guide us through the research process:

1. In the context of program comprehension, do *testsuites*

account for suitable scenarios when performing dynamic analysis?

2. Can we come up with (automatic) abstractions to make *existing* visualizations of large executions scalable and human-readable?

3. On the other hand, can we devise *new*, flexible visualization techniques to cope with large amounts of data?

4. Based on the previous two questions, can we lift our visualizations to an architectural level?

Through the use of dynamic analysis, this research seeks to address these research questions by improving existing and devising new abstraction and visualization techniques, while maintaining scalability and interactivity. We will elaborate on the results achieved thus far, evaluate our techniques using open source and industrial systems, and finally, validate them through controlled experiments.

2. Approach

2.1. Visualization Criteria

When discussing the construction of "comprehensible" visualizations, we need criteria that capture comprehensibility. We consider the following two properties from the realm of visual programming languages to be appropriate in determining the comprehensibility of a view [14]:

Accessibility of related information Viewing related information in close proximity is essential. When two screen objects are not in close proximity, there is the psychological claim that these objects are not closely related to each other, or are not dedicated to solving the same problem.

Use of screen real-estate The term "screen real-estate" refers to the size of a physical display screen and connotes the fact that screen space is a valuable resource. Making optimal use of the available screen real-estate is necessary to prevent excessive scrolling and thus reduce the required effort while locating information.

2.2. Techniques

In order to cope with the large amounts of dynamically gathered data, we roughly distinguish between two techniques: efficient abstractions, and novel visualizations. Though being subject to powerful combinations, we plan for our initial efforts to address each of these approaches more or less separately. We will utilize the Symphony framework [7] in that there exists a distinction between data gathering, knowledge inference, and information interpretation (Figure 1).

Sequence diagrams UML sequence diagrams, especially those generated from runtime data, are a straightforward means to visualize program behavior in an intuitive fashion. While the reverse engineering of such diagrams in itself is not new (e.g., [2]), it is our opinion that the abstraction issue in this context has not been sufficiently addressed and evaluated, and that the origin of the necessary execution scenarios often remains unclear. It is our intention:

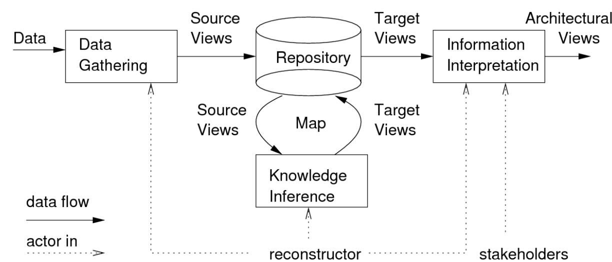

Figure 1. The Symphony reconstruction process.

- to use *testcases* for scenarios, as they tend to trigger specific features and because we consider them to be an ideal starting point when domain knowledge is lacking. Moreover, testcase visualizations may serve as documentation in Agile software development [8, 12];

- to concentrate on *abstraction* mechanisms. We are especially interested in abstractions that can be applied automatically, e.g., based on metrics that are gathered during the testcase execution.

Trace visualization For the effective visualization of execution traces, we need distinct solutions for the size problems of both large systems on the one hand and long traces on the other. We therefore propose to employ two linked views:

- The notion of *hierarchical edge bundles* [9], a visualization technique that bundles the static relationships between a system's structural elements, can be extended with a temporal aspect to visualize execution events. This technique is particularly attractive in our context since the projection of a system's structure on a circle is very efficient in terms of screen real-estate, and because the bundling aspect will improve the readability of the many relationships.

- To provide a navigable overview of the trace, we propose to use an extended version of the information mural by Jerding et al. [11]. We elaborate on this concept by using an improved form of scaling. The resulting view is to be linked to the aforementioned view, and serves to prevent the viewer from getting lost.

2.3. Evaluation and Validation

Evaluation The implementation of our techniques requires a thorough testing phase to assess their usefulness. We will

achieve this by applying tool prototypes on a diverse set of open source systems, among which are medium-sized systems such as JHOTDRAW [1] (300 classes) and CHECKSTYLE [2] (800 classes), and the more complex AZUREUS [3] (4000 classes).

Additionally, our research project's scope grants us access to industrial software such as CROMOD, a complex system that regulates climate conditions in greenhouses. Whenever possible, we will involve the domain experts in determining the effectiveness of our techniques.

Validation Following the development of a series of techniques and tools, we plan to conduct a *controlled experiment*. The purpose is to determine the extent to which visualization tools actually help during certain maintenance tasks. The idea is for heterogeneous groups of graduate and Ph.D. students to perform a series of change requests on an unknown system, with part of the participants having access to our supporting tools while the others are merely dealing with the source code.

2.4. Survey

In order to create a comprehensive overview of related work and to assist the definition and refinement of our future research directions, we conduct a survey of dynamic analysis techniques in parallel with the activities mentioned earlier. The focus in this study is on all papers that use these techniques in the context of software understanding and that have been published in any of the major conferences and journals related to reverse engineering.

3. Preliminary Results

3.1. SDR

We have conducted an experiment [5] in which we have reverse engineered and abstracted a form of UML sequence diagrams, which we call scenario diagrams. At the basis of this process lies a program's testsuite, of which the testcases are used as execution scenarios. Using our Scenario Diagram Reconstruction (SDR) framework, a testsuite is instrumented and executed, during which the events (i.e., method calls) are converted to scenario diagram specifications. In order to make the diagrams more readable, we also collect several runtime metrics to propose sets of (customizable) abstractions on the views.

Figure 2 depicts the reconstruction process. In terms of the Symphony framework, the entities on the top row are the data gathering phase; the usage of abstractions (changing the views) corresponds to knowledge inference; and the visualization of the resulting diagrams comprises the information interpretation phase.

[1] http://jhotdraw.sourceforge.net/
[2] http://checkstyle.sourceforge.net/
[3] http://azureus.sourceforge.net/

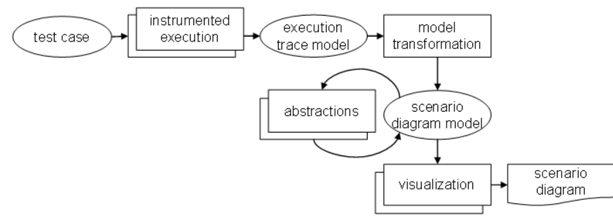

Figure 2. Reconstructing scenario diagrams.

Our approach has been evaluated on a small system consisting of 25 classes, yielding promising results.

3.2. Extravis

In another experiment, we have developed a novel visualization tool to handle large execution traces [10]. The solution that we propose is twofold: the *massive sequence view* uses importance-based anti-aliasing (IBAA) to provide a zoomable overview of the trace, whereas the *circular bundle view* warrants for the grouped visualization of the call relations in the current timeframe. The latter technique is illustrated in Figure 3, in which the calls are shown in a light-to-dark fashion so as to indicate their directions.

The tool, EXecution TRAce VISualizer (EXTRAVIS)[4], is based on the views that were proposed in Section 2.2 and reflects a strong emphasis on scalability. It has proven effective [6] in three distinct program comprehension contexts, among which are trace exploration, feature location, and feature comprehension. The subject systems included JHOTDRAW (150,000 events) and CROMOD (270,000 events).

3.3. Dynamic Analysis Survey

Our survey of dynamic analysis techniques is underway and we are in the process of systematically organizing the material that we have collected, which amounts to approximately 150 papers from eight conferences and four journals. We are using formal concept analysis to group the papers by keywords and research topics addressed.

4. Further Research

4.1. Views and Dualities

The views that were created using the techniques mentioned earlier are subject to improvement. For example, in the context of scenario diagram reconstruction, there is clearly a need for intricate abstraction methods to keep the diagrams sufficiently readable, particulary when dealing with large

[4] Available at http://www.swerl.tudelft.nl/extravis/

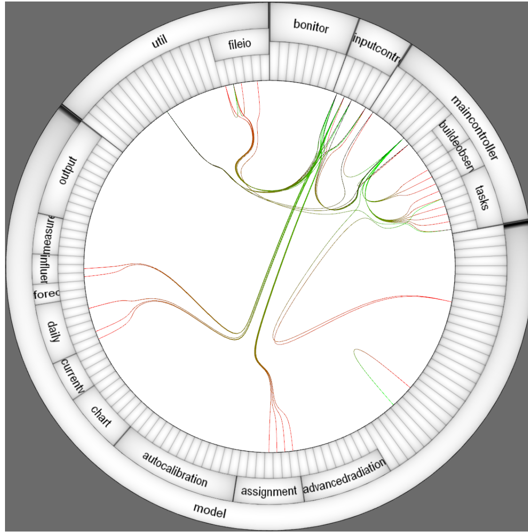

Figure 3. Visualizing execution traces.

software systems. The idea is for the runtime metrics to play a key role in *automatic* abstractions. Moreover, by omission of the more detailed interactions, we hope to attain visualizations at a more architectural level.

With respect to trace visualization, the major issue is the massive amounts of data. Our tool implementation indicated a lack of computational resources, and the limited amount of screen real-estate prohibits the display of all information without (unwanted) abstractions. An alternative would be to introduce a preprocessing step that involves certain abstractions.

Finally, our experiments revealed an interesting *duality*: whereas EXTRAVIS's strength lies in scalability and overview, the views presented by SDR's scenario diagrams are more intuitive because of their chronological ordering. More effort is needed to present new views that combine the best of both worlds, e.g., by combining these techniques or presenting derivations thereof.

4.2. Evaluation

To further evaluate our techniques, we plan to conduct studies on systems that are larger and, thus, more challenging.

The introduction of new abstraction mechanisms in the reconstruction of scenario diagrams, for example, requires extensive testing and tweaking, as was indicated in preliminary experiments with CHECKSTYLE (800 classes). Also, we feel that it must be verified whether our testsuite visualizations actually capture the essence of the testcases at hand; one could think of conducting a controlled experiment or, in an industrial context, consulting the domain experts.

With respect to our trace visualization technique, we expect the bundle view to scale up pretty well due to the circle approach, and we plan to verify this assumption by means of studies on AZUREUS and larger industrial systems.

References

[1] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: Its extracted software architecture. In *Proc. 21st Int. Conf. on Software Engineering (ICSE)*, pages 555–563, 1999.

[2] L.C. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of UML sequence diagrams for distributed java software. *IEEE Trans. on Software Engineering*, 32(9):642–663, 2006.

[3] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.

[4] T.A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[5] B. Cornelissen, A. van Deursen, L. Moonen, and A. Zaidman. Visualizing testsuites to aid in software understanding. In *Proc. 11th European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 213–222. IEEE, 2007.

[6] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. van Wijk, and A. van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proc. 15th Int. Conf. on Program Comprehension (ICPC)*, pages 49–58. IEEE, 2007.

[7] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-driven software architecture reconstruction. In *Proc. 4th Working Conf. on Software Architecture (WICSA)*, pages 122–134. IEEE, 2004.

[8] A. Forward and T. Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proc. ACM Symp. on Document Engineering*, pages 26–33, 2002.

[9] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Trans. on Visualization and Computer Graphics*, 12(5):741–748, 2006.

[10] D. Holten, B. Cornelissen, and J. van Wijk. Visualizing execution traces using hierarchical edge bundles. In *Proc. 4th Int. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 47–54. IEEE, 2007.

[11] D. Jerding and J. Stasko. The information mural: A technique for displaying and navigating large information spaces. *IEEE Trans. on Visualization and Computer Graphics*, 4(3):257–271, 1998.

[12] B. Marick. Agile methods and agile testing. http://testing.com/agile/agile-testing-essay.html (accessed June 8th, 2007), 2004.

[13] R. Sharp and A. Rountev. Interactive exploration of UML sequence diagrams. In *Proc. 3rd Int. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 8–15. ACM, 2005.

[14] S. Yang, M. M. Burnett, E. DeKoven, and M. Zloof. Representation design benchmarks: a design-time aid for vpl navigable static representations. *J. Visual Lang. & Computing*, 8(5-6):563–599, 1997.

[15] A. Zaidman. *Scalability Solutions for Program Comprehension through Dynamic Analysis*. PhD thesis, University of Antwerp, 2006.