What architects should know about reverse engineering and reengineering

Rainer Koschke
University of Bremen, Germany
http://www.informatik.uni-bremen.de/st/~koschke
koschke@informatik.uni-bremen.de

Abstract

Architecture reconstruction is a form of reverse engineering that reconstructs architectural views from an existing system. It is often necessary because a complete and authentic architectural description is not available.

This paper puts forward the goals of architecture reconstruction, revisits the technical difficulties we are facing in architecture reconstruction, and presents a summary of a literature survey about the types of architectural viewpoints addressed in reverse engineering research.

1 Introduction

In software development, one of our most powerful tool is abstraction. Software architecture is an abstraction. Figure 1, for instance, shows a reference architecture of compilers as you find it in textbooks [6]. Given this description, you can explain the individual steps in the transition from source text to object code in a compiler. The dashed lines indicate the logical flow of information. The solid lines show how this flow is actually realized: by way of two data structures connecting the processing steps: the abstract syntax tree (AST) and the associated symbol table. The description abstracts from the details of the processing steps and helps to understand the overall structure.

Such architectural descriptions are often incomplete. It is interesting to note, for instance, that the picture does not show a connection between the symbol table and the AST, although the symbol table is a mapping of symbols onto nodes in the AST. This description would be sufficient to explain certain aspects of a compiler, but it would be insufficient for most maintenance tasks. For instance, if a maintenance programmer wants to analyze the potential ripple effects of changing the AST, he will not find the relation between AST and symbol table in this architectural description.

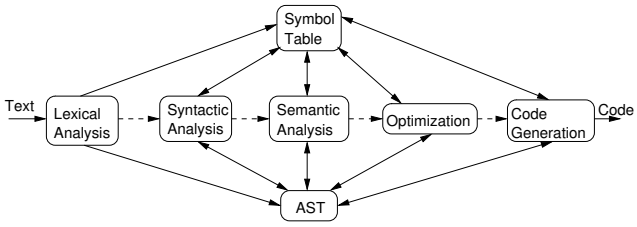

Figure 1. Software Architecture in Textbooks

In practice, architectural description are rarely complete if they exist at all. "The truth is in the code" many people say. They are right to the extent that the code usually tells us nothing about rationals for design decisions nor the alternatives that have been considered and dismissed. However, even if a lot of truth is in the code, it is typically well hidden and the connection between low-level facts you can extract from a system and the abstract architectural description are non-obvious. In such cases, you must reconstruct this connection.

Architecture reconstruction is a form of reverse engineering that reconstructs architectural views from an existing system, where reverse engineering is defined as the process of analyzing a subject system to identify the system's components and their interrelationships and to create representations of the system in another form or a higher level of abstraction [1]. As an incidental remark, it is interesting to see the common elements in the definition of reverse engineering and software architecture: we have components and interrelationships and we have abstraction.

2 Interrelationships and Abstraction

People not involved in reverse engineering research are often mislead by available commercial tools about what reverse engineering can offer. These tools typically recover modules and their dependencies from code through some static analysis. Most UML tools, for instance, are capable of creating class diagrams for object-oriented code. The result is often disappointing; what you get has little abstraction. For instance, these tools do not infer the different types of associations (composition, aggregation, or general association). These tools even miss classes and dependencies sometimes [4].

What these tools actually perform is what we call *extraction*: inference of information from source code or other artifacts or from the execution of the program. Extraction—although often hard enough—is just the first necessary step in reverse engineering. Once we have the information, we need to abstract it, infer new knowledge, and present it to a human analyst [8].

Extraction in itself can be difficult if it comes to large programs and subtle interrelationships. Commercial tools often do little more than the semantic analysis of a compiler. Many give you, for instance, a call graph. Extracting such a call graph is relatively easy if you take only named calls into account. Yet, such call graphs are typically incomplete because the tools do not handle indirect calls through function pointers or dispatching calls in object-oriented programs. Furthermore in software security, researchers try to detect the many tricky ways to manipulate or hide control flow through malicious attacks. These things are extremely difficult to detect.

But even in a friendly environment with advanced pointer analysis, you will still miss dependencies. Code can be used by calling it, or code can be used by simply copying the source text. Such copying creates code clones. Code cloning is not covered by call graphs; yet, they do create subtle dependencies.

To give you another example for extraction and abstraction, let me turn to inheritance in object-oriented programs. Extracting the inheritance relation is simple only at first sight. What UML tools give you is just syntactic inheritance. Imagine someone would collapse all your classes into one. Syntactically, you have just one class, where in fact you have many logical classes.

There is a technique by Snelting and Tip [7] that reconstructs a precise inheritance hierarchy from the code irrespectively of the syntactic inheritance.

The technique determines the required class attributes (i.e., data and function members) for every local, global, or heap variable that has a class type and every pointer that points to a class type. So, on one hand, you have all variables and on the other hand the class attributes they require. That obviously constitutes a binary relation. This binary relation is analyzed by a precise mathematical technique called formal concept analysis. The result is a lattice that describes the actual inheritance relation.

The technique combines pointer analysis and formal concept analysis, two relatively complex techniques.

Reverse engineering attempts to find such subtle interrelationships and to raise the level of abstraction.

3 Technical Challenges

In reverse engineering, we use the whole spectrum of program analyses ranging from static to dynamic analyses. Static analyses are typically preferred for structural aspects of the architecture whereas dynamic analyses are often used for behavioral aspects. Dynamic analysis, however, is not the only way to tackle behavior. Optimizing compilers know quite a lot about the behavior of a program. And they do perform static analysis.

For many people, dynamic analysis appears as a quick win. But dynamic analysis is easy only at first sight. Dynamic analysis is related to testing and shares the same disadvantages. All the conclusions you draw are valid only with respect to the given input. When it comes to architecture, however, we are generally interested in all possible behavior.

And dynamic analysis can create a huge amount of data if you instrument a program blindly. In fact, in the reverse engineering context, gathering dynamic information is even more difficult than in testing. In glass-box testing, you may assume prior knowledge on the system. In black-box testing, you can start from a specification. In reverse engineering, we have neither specification nor prior knowledge.

Yet, static analysis is neither simple. Fortunately, we can leverage many advanced techniques from the compiler community. But unlike compilers, we generally need whole-program analyses. Such analyses are quite expensive and the programs we analyze tend to be huge, often written in multiple and awkward languages.

Compilers are batch-oriented tools. In reverse engineering, the user is usually in the loop. So, our analyses are typically interactive.

Another difference to compilers is the question "how conservative must we be?" We all agree that compilers should make conservative assumptions in case of uncertainty. Because of exaggerated pessimism, a compiler

may refuse to make a certain optimization. The cost, then, is a somewhat slower program.

In our context, the costs are much higher. Pessimism may result in massive false positives to the extent of uselessness of the analysis to an analyst. For me, it is still an open question how much optimism we can and should afford.

Then, there are additional problems a compiler does not care about. Consider, for instance, the ambition to analyze non-preprocessed code. The syntax tree may be completely different depending on how certain macros are expanded. Yet, if you do code transformation in the reengineering context, you must analyze all possible configurations of the source text and you have to re-generate the code as close as possible to the original text. A compiler does not care about such nasty things.

This dilemma leads us to the question of software quality. There are many quality factors for an architecture. One of them is analyzability. It seems as if analyzability falls victim to flexibility these days. Current movements in Java, for instance, such as loading classes at runtime or modifications and control flow through reflection, remind us of the days of self-modifying code. How will we maintain these dynamic systems if we cannot analyze them?

4 Architectural Views

Architecture reconstruction creates architectural views for existing systems. A *view* is a representation of a whole system from the perspective of a set of related concerns [3]. Such views are formalized through viewpoints. A *viewpoint* specifies the kind of information that can be put in a view [3].

Viewpoints are very popular in forward engineering. Zachman was one of the first authors on viewpoints. He proposed 6×6 different viewpoints. Perry and Wolfe proposed a simplified version of these views, distinguishing only three viewpoints. Then you have the 4+1 viewpoints by Philippe Kruchten and the four Siemens viewpoints.

The number of viewpoints is confusing, in particular, because many of them are very similar. Recently, the book by Clements and colleagues brought some order to this sea of viewpoints [2]. They distinguish the following categories of viewpoints:

M: Module viewpoints show static structure and describe the decomposition, layering, and generalization of modules and their use dependencies. A module is a code unit that implements a set of responsibilities.

Cat.	Style	Content	#
M	decomposition	part-of	43
	feature location	implements	16
	design patterns	element participates-in pattern	12
	class diagrams	association, aggregation	10
	conformance	conforms-to, deviates-from	7
	interfaces	requires, provides	3
	use cases	implemented-by	2
	configuration	varies-with	2
	class hierarchies	inherits, attribute-of, method-of	2
CC	object interaction	interacts-with	12
	process interaction	interacts-with	10
	component interaction	interacts-with	3
	conceptual viewpoint	implemented-by	3
	object traces	applied operations	2
A	responsibilty	responsible-for	1
	build process	generated-by	1
	files	described-in, stored-in	1
–	view integration	element corresponds-to Element	5

Table 1. Adressed Viewpoints

CC: Component-and-connector viewpoints express runtime behavior described in terms of components and connectors. A component is one of the principal processing units of the executing system; a connector is an interaction mechanism for the components.

A: Allocation viewpoints describe mappings of software units to elements of the environment (the hardware, the file systems, or the development team).

The interesting question now is which viewpoints do we as reverse engineers address? In Table 1, I am relating reverse engineering papers that address architectural views to the category by Clements et al. The last column of this table contains the number of papers that address this viewpoint. The number stems from a recent comprehensive literature survey of mine that covered all relevant conferences and journals of the last 10 years [5].

The vast majority of research papers is devoted to

finding modules or subsystems, generally through software clustering. These techniques indeed typically extract dependencies and try to find cohesive elements.

There is also a bunch of papers that try to locate functionality in code. Such techniques establish traceability links between requirements, architecture, and code.

Also, design patterns are being searched for quite frequently. It is interesting to note that we have seen a similar movement in the 90ies. It was called plan or cliche recognition, which looked for code idioms.

Moreover, various heuristics are investigated to distinguish aggregation, composition, and general association in reverse engineering UML class diagrams from source code.

And there are a few more papers on checking architecture conformance. Typically, however, these are structural checks such as the reflection method by Gail Murphy.

Then, there are papers that try to find component-and-connector viewpoints. Interestingly enough, dynamic analyses are not dominating this area of research, although you indeed find much more than for module viewpoints. They differ in the types of runtime entities for which they reveal interaction patterns.

Finally, you get only very few papers that try to embed software into its environment. Relating software to its environment seems to be either not a popular topic or too difficult. One of these papers clusters modules according to their ownership, the so-called ownership architecture. Another one, on the other hand, describes ways to model and reconstruct the build architecture.

In addition to that, there are a few papers that try to find links between different viewpoints. Such viewpoints could be considered meta viewpoints as they map views onto each other. That is why they do not fit into the category by Clements and colleagues.

5 Conclusions for Future Research

We have proposed many clustering techniques in our research. They are all more or less based on structural information. Some of them take similarity of identifiers into account additionally. Usually, they try to cluster according to the notions of cohesion and coupling. But these are not necessarily the human criteria for groupings. Humans cluster things because they are semantically related. Yet, it is difficult to extract this kind of semantics from code. We cannot perform miracles.

Moreover, it is an open issue on how we relate these clusters to a model of the intended architecture if we have one. Identification of aspects is another issue. As opposed to normal feature location, aspects – by definition – are cross-cutting concerns and cannot be mapped onto a single element.

Then we have the adoption problem. As in many other disciplines, it takes us too much time to transfer our research tools to practical use.

There are also open management issues such as suitable process models and cost estimation. I think that is a completely underdeveloped area.

Similarly, me must intensify our effort in architecture conformance checking that goes beyond structural checks.

Fowler and others have presented a catalog of bad smells and associated refactorings to eliminate them. Could something similar be written for architectural refactorings? What are the bad smells at that level? What are the transformations to cure them? In the end, are architectural refactorings any different from code refactorings?

And last but not least, how do we analyze these dynamically reconfigurable systems that are being used, whose number will even increase in the near future?

References

[1] E. J. Chikofsky and J. H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, Jan. 1990.

[2] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architecture*. Addison-Wesley, Boston, 2002.

[3] IEEE P1471. IEEE recommended practice for architectural description of software-intensive systems—std. 1471-2000, 2000.

[4] R. Kollman, P. Selonen, E. Stroulia, T. Systä, and A. Zündorf. A study on the current state of the art in tool-supported uml-based static reverse engineering. In *Working Conference on Reverse Engineering*, pages 22–31. IEEE Computer Society Press, 2002.

[5] R. Koschke. Rekonstruktion von software-architekturen – ein literatur- und methoden-Überblick zum stand der wissenschaft. *Informatik Forschung und Entwicklung, Springer-Verlag*, 19(3):127–140, 2005.

[6] M. Shaw and D. Garlan. *Advances in Software Engineering and Knowledge Engineering*, chapter An Introduction to Software Architecture. World Scientific Publishing Company, River Edge, NJ, 1993.

[7] G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In *Proceedings of the ACM SIGSOFT sixth international symposium on Foundations of software engineering*, pages 99–110. ACM Press, Nov. 1998.

[8] S. Tilley, S. Paul, and D. B. Smith. Towards a framework for program understanding. In *Proceedings of the International Workshop on Program Comprehension*, pages 19–28. IEEE Computer Society Press, 1996.