Towards a Generic Extraction Technique

Tobias Rötschke
Philips Research Laboratories
Prof. Holstlaan 4
5656 AA Eindhoven, The Netherlands
tobias.roetschke@philips.com

April 20, 2000

Abstract

Within Philips Research, we currently look for a uniform and reproducible way to deal with different programming languages, architecure models, and output formats in the extraction phase of reverse architecting. Graph grammars provide an excellent means to specify these aspects of extraction both formally and understandably. Although existing graph rewriting systems are too inefficient to be applied directly for large projects, some major ideas can be adopted. This paper motivates why a more generic extraction approach is needed and sketches some ideas how graph grammars can contribute to reach this goal.

1 Introduction

A broad range of different product families is currently developed and maintained within the Philips company. Because the requirements of our systems vary widely and evolve over time, there are different approaches to software architecture[1] as well. Accordingly, a noticable large variety of architectural concepts, programming languages, and design tools is used. This makes extracting architectural information a tedious job and extraction tools have to be built almost from scratch for each system.

The aim of our current work is to provide a generic extraction technique, which deals explicitly with different programming languages, architectural concepts, and data formats. Of course, this approach would save no time when applied to just one or two systems, but a flexible solution would significantly ease the effort of building extraction tools for different systems.

[1] In this paper, the notion of "architecture" meets the definition of a "module architecture view" in [HNS99]. Other views are likely to be considered in future.

2 Analysis of existing reverse architecting approaches

Within Philips Research, reverse architecting follows the extract-abstract-present paradigm [Kri99]. Relation Partition Algebra (RPA) provides excellent formal means to process architectural information within the *abstraction* phase, based on the simple mathematical concepts of sets and relations [PS99]. The results are easily *presented* by existing tools [B+00, FJ98]. Until now, we lack an elegant *extraction* technique, that can be used for a broad range of systems.
Although we have some working extraction tools designed for the solution of particular problems, they are not suitable for many other systems. To make reverse architecting applicable for a wide range of product families, a more flexible extraction approach is needed, that meets the following requirements:

- Support for all relevant[2] *programming languages.*
- Independence of concrete *architectural concepts.*
- *Scalability* up to several million lines of code.
- *Automation* of the extraction process to eliminate unnecessary user interaction.
- *Connectivity* with other reverse engineering tools.
- *Maintainability* of implemented extraction tools.

Related work None of the existing reverse engineering tools (e.g. [BG98]) meets all of these requirements. Many tools support only one language (often C or C++) or rely on a fixed architectural model. Even when multiple languages are supported, only minor architectural concepts like classes and methods are handled explicitly.
The reason is that in most present programming languages high-level architecural concepts like subsystems, components, layers etc. can not be specified.

[2] Relevant means used for development of our products.

Consequently, this information resides implicitly in the file system or in proprietary project files. Architecture description languages (ADLs) [Cle96] could be used in combination with existing programming languages to make the implementation of higher-level architectural elements explicit. That would result in a consistent view of the system's implementation and enable true architecture verification, i.e. comparing implemented and specified architecture.
An important preliminary step to make reverse architecting more flexible and to benefit from the effort of different research sites is the idea to define an open exchange format as in [BGH00]. The proposed format is based on the highest possible abstraction of an architectural concepts consisting of entities and relationships, which corresponds to RPA sets and relations used in [Kri99] for software rearchitecting. Provided that it retrieves all required architectural concepts, one could instantly use any existing extraction tool.

3 Conceptual solution

The conceptual solution combines ideas from the field of compiler construction with graph rewriting techniques. Separation of concern should be achieved by modularizing independent pieces of information required for the extraction process. In order to keep the development and maintenance effort of extraction tools as low as possible, code generation is used wherever possible to realise the technique.

The complete extraction process consists of two major phases: A *generation phase* and the actual *extraction phase*, as depicted in Figure 1. The latter is somewhat simplified in order to focus on the important ideas. The choice of proposed input documents is the prelimary result of ongoing discussion and might change in the future. Language description and mappings actually consist of multiple related documents.

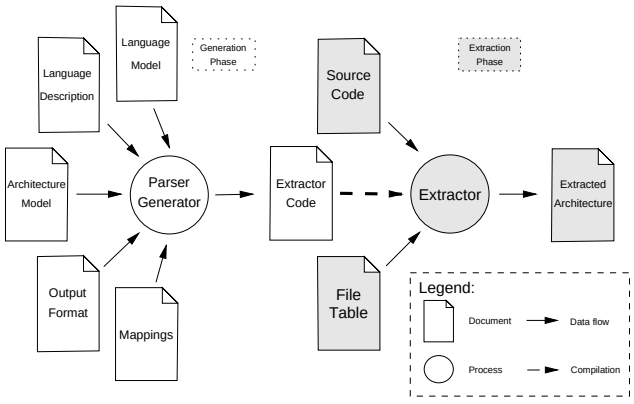

Figure 1: The generic extraction approach

3.1 The generation phase

The generation phase produces source code for an extractor dedicated to the system to be analysed. While the generation framework is considered stable, all diversity is handled explicitly by refining or modifying at least some of the five different input documents.

Language description The context-free syntax of many languages can be described by means of an EBNF grammar, which is easily understood. Results from the IPSEN project [Nag96] show that rules for the generation of a parse-tree can automatically be derived from a given EBNF in Schäfer-Engels normal form [ES89]. Graph grammars allow to formally specify name and type binding rules in a easy understandable manner. Tools like PROGRES [SWZ95] can even generate prototypes for this kind of specification. We currently try to tailor these ideas to our needs, using the more efficient ANTLR parser generator toolkit [Par95].
Although only a few number of different programming languages are used within Philips, practically every project makes use of certain naming conventions or places architectural information within special comments. In principle, the different projects use *refined* instances of the original language. The language description document for the extraction tool will specify this refinement relation explicitly.
Larger architectural entities are often *implemented* by placing *files* in a *directory structure* according to certain programming conventions, that are usually settled in some pieces of system documentation, which might be inaccessible for automatic extraction. In order to let the source code describe virtually all aspects of implementation, we use a *project description language*, which is deemed an *extension* of the programming language.

Language model Of course, the resulting parse-tree is huge even for medium-size projects – Parsing a 110 KLOC Java project with an additional 360 KLOC for JDK and Swing sources resulted in more than 6 Million nodes. However, most of the information stored in the parse-tree is useless from an architectural point of view. For instance, the precedence of operators is an irrelevant artefact that results in tens of nodes for every expression. The goal is to derive a more compact representation, reducing the number of required nodes by two orders of magnitude. This representation will contain only significant artifacts specified in a simplified *language model*, similar to the approach of *Resource Graphs* as proposed in [EKP99].

Architecture model The architecture model defines the conceptual entities of the software architecture by means of a graph scheme. In spite of the fact that it is rather similar to the language model, they can change independently. For instance, Java packages can

realise system, subsystems, and components according to one architecture model, while another consists of uniform building blocks [LM95].

Mappings The three documents described above must be glued together by means of specifying which parts of the language model represent which concepts of the context-free syntax, and how architectural model concepts are realized by language model concepts. This document integration can be formally described with triple graph grammars [Sch94].

Output formats This document specifies how the entities and relationships are represented in terms of the architecture model as a result of the extraction process.

3.2 Extraction phase

The extraction phase starts with generating a project description from analysing the file system. After parsing the project description, which forms the top of the parse-tree simplified according to the language model, all files of the project are parsed by the extractor and the resulting trees are attached to the already existing one. After name and type binding has been performed, the complete structure is transformed to the output format according to the architectural model.

4 Conclusion

Much work still is to be done to realise a generic extraction technique, mostly because we need efficient, incremental graph transformation algorithms to achieve the scalability requirements. Nevertheless, we would benefit from a flexible generic extraction technique as presented in this paper, because it would enable us to deal with architectural diversity in different systems more easily. Solutions of other research sites could be used much easier, so that more human resources would be free to spent effort on other, often even more compelling fields of architecture analysis and verification. Hence, the quality of our products could be further improved.

References

[B+00] Reinder Bril et al. Maintaining a Legacy: Towards Support at the Architectural Level. *Software Maintenance*, February 2000.

[BG98] Berndt Bellay and Harald Gall. An Evaluation of Reverse Engineering Tool Capabilities. *Software Maintenance: Research and Practice*, 10:305–331, 1998.

[BGH00] Ivan Bowman, Michael Godfrey, and Richard Holt. Connecting architecture reconstruction frameworks. *Information and Software Technology*, 42(2):91–102, 2000.

[Cle96] Paul Clements. *A Survey of Architecture Description Languages.* In *Proc. 8th International Workshop on Software Specification and Design*, Paderborn, March 1996.

[EKL99] J. Ebert, B. Kullbach, and F. Lehner, editors. *Proc. Workshop Software-Reengineering.* Universität Koblenz-Landau, 1999.

[EKP99] Thomas Eisenbarth, Rainer Koschke, and Erhard Plödereder. *Projekt Bauhaus: Interaktive und inkrementelle Wiedergewinnung von SW-Architekturen.* In [EKL99], pages 17–26, 1999.

[ES89] G. Engels and W. Schäfer. *Programmentwicklungsumgebungen, Konzepte und Realisierung.* B. G. Teubner Verlag, 1989.

[FJ98] L. Feijs and R. de Jong. 3d Visualization of Software Architectures. *Communications on the ACM*, 41(12):73–78, December 1998.

[HNS99] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture.* Addison-Wesley, 1999.

[Kri99] René Krikhaar. *Software Architecture Reconstruction.* PhD thesis, Philips Electronics N.V., Eindhoven, 1999.

[LM95] F. van der Linden and J. Müller. Creating Architectures with Building Blocks. *IEEE Software*, pages 51–60, November 1995.

[Nag96] M. Nagl, editor. *Building Tightly Integrated Software Development Environments: The IPSEN Approach.* Springer, Berlin, 1996.

[Par95] Terence Parr. Language Translation using PCCTS and C++. Technical report, Automata Publishing, San Jose, CA, 1995.

[PS99] André Postma and Marc Stroucken. *Applying Relation Partition Algebra for Reverse Architecting.* In [EKL99], pages 27–32, 1999.

[Sch94] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. Technical Report Technical Report AIB 94-12, RWTH Aachen, December 1994.

[SWZ95] Andreas Schürr, Andreas Winter, and Albert Zündorf. Graph grammar engineering with progres. In W. Schäfer and P. Botella, editors, *Proc. European Software Engineering Conference (ESEC) '95*, volume LNCS 989, pages 219–234, Berlin, 1995. Springer.