Tool-support for Reengineering of object-oriented systems

Position paper on the FAMOOS-project

UDO GLEICH THOMAS KOHLER

Juli 1997

Abstract

Searching for methods and ways to transform object-oriented legacy systems into new framework-based applications is the main intention of the ESPRIT project FAMOOS. The partners involved expect a tremendous increase in software flexibility from this approach. This results in the understanding that effective reengineering of large-scale applications is not feasible without sufficient tool support. At this time there are already a few considerations about what tool support for reengineering should look like. This paper gives an overview of the work at Daimler-Benz regarding this. It starts with a short summary of the different tasks that could be related to the term reengineering. Subsequently, a generic proposal for a possible tool architecture is given. Finally the paper introduces a few tool-prototypes which are currently being developed at Daimler-Benz.

Contents

1 Introduction

The goal of the FAMOOS Project is to develop methods and tools for the reengineering of **object-oriented legacy systems**. Such software, mainly first-generation oo-systems, should be migrated to a modern framework-based architecture.

A legacy system is a software system which cannot be adapted to new requirements with the usual maintenance costs, although the new requirements are near or within the problem domain of the software. An object-oriented legacy system mainly contains code in an object-oriented or object-based programming language. Non-object-oriented code could also be included in a software like this.

We can say that a lack in flexibility is the main characteristic of a legacy system. Flexibility cannot be achieved for all conceivable dimensions of an application, and it is not sensible to ask for absolute flexibility in a rather complex system. Flexibility should be achieved for the parts of a software which have to be changed during the evolution of a system. Naturally, it is not easy to find these potentially changing components. The objective of a software system may vary during its evolution, therefore the flexibility requirements do not remain the same as in the very beginning. Consequently, a certain level of legacy easily comes into a complex system.

The main reason for maintenance problems is related to the preceding topic. It is the uncontrolled development of a software system. New features are brought into a program without adapting the application model to the new requirements. That leads to solutions which do not fit into the existing system.

Another property of legacy systems which often occurs is the poor or even missing documentation. Documentation comprises not only analysis and design documents, but also the comments in the source code. Nonexistent documentation leads to maintenance problems through the non-understanding of the software. In addition to this, badly-documented systems are in most cases also badly structured. Evolutionary growing systems which are not consequently documented from the very beginning are promising canidates for reengineering. Undocumented systems arise if no software engineering approach is followed, and therefore no analysis and design documents are created.

Early object-oriented software projects were often afflicted with these problems, because the methods and tools were, as a rule undeveloped and the programmers inexpierienced. Now we are facing the problem of reengineering this software.

2 A reengineering environment

2.1 Overview

A focus here at Daimler-Benz lies on the tool support for reengineering. Ideas for an environment for tools and data for reengineering are currently under development.

A reengineering environment should support the whole reengineering life cycle of reverse and forward engineering. The objective of reverse engineering is the redocumentation and finally the understanding of the software and its problems. During forward engineering the software is restructured to fulfill the new flexibility requirements.

A reengineering tool should also have an open architecture in order to integrate third party tools and to adapt it to different approaches to the reengineering life cycle. The tool has to be adaptable to various needs. These are:

- Scaleability
  - to application size
  - to reengineering needs (part of application, whole application)
- Adaptability to reengineering goal
  - changing of functionality
  - inserting of new functionality
  - integration of several systems
  - paradigm change
  - port to another platform

A reengineering environment should include the following components:

- a repository to store all information, which is created during the reengeneering life cycle
- four categories of tools, which are:

  - reverse engeneering tools
  - problem detection tools (metrics, heuristics)
  - browsers/viewers
  - forward engineering tools

The initial idea of an architecture for a reengineering environment is shown in figure 2.1 which will be described in the following sections.

2.2 A Repository for Reengineering

A reengeneering repository should support different abstraction levels for a software system. These are:

- source code information
- language independent symbolic information
- subsystems (groups, clusters[1]) of elements such as classes, methods etc.
- informal documentation

The database has to control matching between the abstraction levels. It has to be possible to find all documentation or subsystems from code or the according code from a higher level. It should be possible to store formal and informal information referring to each other. It should be possible to assign informal documentation to every item on every abstraction level.

The basis of a reengineering repository should be an adequate complete meta model for object-oriented and non-object-oriented software structures. A database which implements this model shoud include three parts:

- the representation of the source-code in a form independent of any programming language,
- language dependent plug-ins and
- groups and clusters of the underlying software structure in order to divide the system into understandable pieces

[1]the term "grouping" is currently used within the FAMOOS-project for this issue

Currently, a UML-repository for an object-oriented database which will be used for the FAMOOS-project is under development here at Daimler-Benz.

The next abstraction layers should be groups or clusters of symbolic information from the database. The best approach for this is probably a graph description language capable of graph-subgraph-relationships.

The informal documentation of the software system is created during reengineering or it is already availabe. Therefore, it is neccesary to integrate various forms of documentation formats. It should also be possible to attach informal documentation to every item on every abstraction level. Consequently, the best way would probably be to organize the documentation in hypertext-based form.

2.3 Tools for reengineering

The first step in reengineering is reverse engineering. The goal of this part is the understanding of the software system.

The understanding of a software system can follow three basic cognitive models for problem solving. The top-down approach starts with a general idea about the system functionality and goes on to associate functionality with components. The bottom-up approach tries to group parts of the code and to assign concepts to those groups. The third approach is to create a set of hypotheses and to adapt them until they are consistent [Doe97]. Tools for reverse engineering should support these approaches.

Parsers and debuggers are needed to take the step from the source code to a language independent-representation. The creation of static information is a solved problem for most programming languages. Dynamic information can be received through code instrumentation or via debuggers. Only one debugger which can handle a program on the level of objects and messages is known to the authors - Look! by OST [loo]. A mixture of newly developed tools and the integration of available solutions would be the proper way here.

Symbolic information should be divided into understandable pieces. This could be done automatically or by hand by a domain expert. The arising subsystems will be adapted and documented to recover the software design. The rules for the cre-

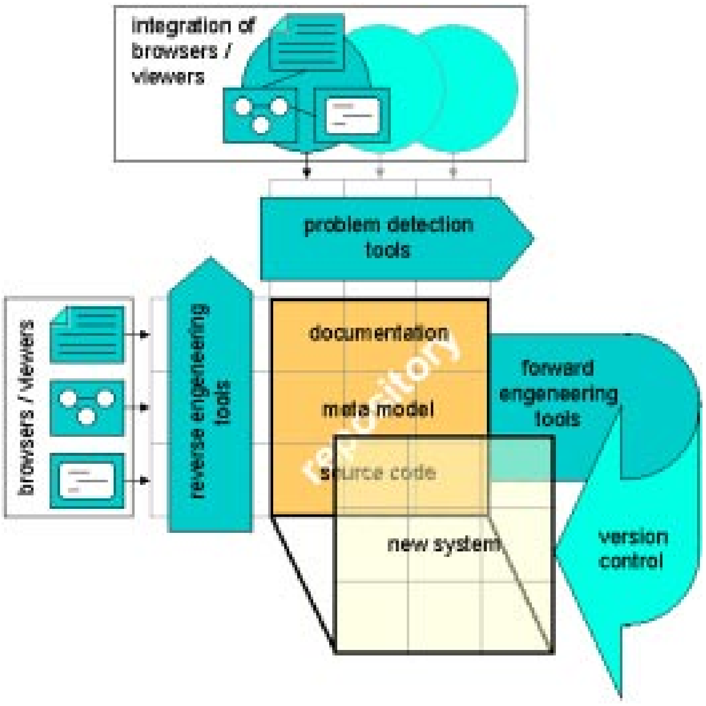


Figure 1: Reengineering Environment

ation of clusters and groups are a current research issue in the FAMOOS-project. Metrics and heuristics for object-oriented systems would be useful for this and for the problem detection in the software as well.

Browsers and viewers for the various abstraction levels are important tools for software understanding. Therefore, they are an integral part of a reengineering environment. Such tools include text editors, hypertext viewers and tools for graph visualization. They are possibly already available, which means they only have to be integrated into the environment, or they have to be developed from scratch. There should also be tool integration so that the browsers and viewers get a deeper (vertical) view into a smaller part of a software on different abstraction levels (see figure 2.1).

The tools for forward engineering include the ordinary solutions for software engineering required to do the work for analysis, design and implementation of the new system. Special restructuring tools to convert a legacy system into a "repaired" one are unique to reengineering. It is conceivable that such tools could work on different abstraction levels, on the source code as well as on higher levels. It is very likely that the restructuring can only be done semi-automatically which means using human intervention. Pattern-based restructuring operations are under research in the FAMOOS project. The methods are in an early phase of development. Tools will hopefully be developed in parallel.

Successful software maintenance is based on a good administration of the new variants of the software system. Therefore, the integration of version control tools is an important task, patricularly in multi-user environments.

3 Implemented tool prototypes

After this common description how a tool architecture could look like, we will introduce the tool-prototypes developed at Daimler-Benz for the FAMOOS-project. The tools mainly support the early phases of reengineering. There are two prototypes processing Ada-code and vizualizing package dependencies. The third tool shows class dependencies using symbolic information of C++-code. It allows browsing through a class diagram.

3.1 Tools for ADA-Systems

One case study provided by Daimler-Benz for the FAMOOS project is a large software assembly for space-travel operations, which is implemented in ADA83. Currently at Daimler-Benz there are two separate approaches to the analysis of ADA programs. The main difference between these approaches is the

At this time we use these distinct approaches to evaluate the applicability and usefulness of two different concepts. The main difference between these concepts is the usage of different parsing-components for the tools. In future we will only pursue further one of these approaches after an accurate comparision of the various advantages/disadvantages.

In the following sections these approaches (tool-prototypes) will be called ADA-Analysator and ADAlyzer.

3.1.1 ADA-Analysator

The origin of ADA-Analysator in its current shape is a diploma thesis [Nar97] done for Daimler-Benz. This program is an Analysis-Framework with the basic functionality of static source-code analysis. The tool can be customized, especially to extend the scope of the analysis as desired. The main part of the framework consists of a syntax-analyzer capable of analyzing ADA83-code. It is based on an object-oriented C++ class hierarchy, which is responsible for the construction of a temporary language.

The syntax-analyzer is hereby generated with the compiler-generator COCO-2 [DP90]. The greatest advantage of COCO-2 is the possibility of describing both the lexical structure and the syntax/semantics of a given source language commonly in one document.

This syntax-analyzer now controls the construction of a dynamic data-structure, the temporary language which is a first abstraction (in the given scope) from the source-code. This data-structure serves as a basis for the following analysis. At this time a small set of such examinations has already been implemented. For example it is possible to show the structure and the interrelationships (WITH relations) of ADA-Packages. It is also possible to group together all instances of the same generic.

The operation of the tool and the visualisation of the results is realized via a graphical user interface (GUI). A screen-shot of the running program is shown in figure 3.1.1. Hereby currently the structure and interrelationships of packages are shown. Additionally, it is possible to export these package-relations to an EXCEL-chart.

Due to the fact that the GUI is implemented with Borland OWL, the ADA-Analysator is currently only available on a Win95/NT-platform. Since the interface is completely implemented separate from the rest of the program, it could be ported to other platforms without greater problems.

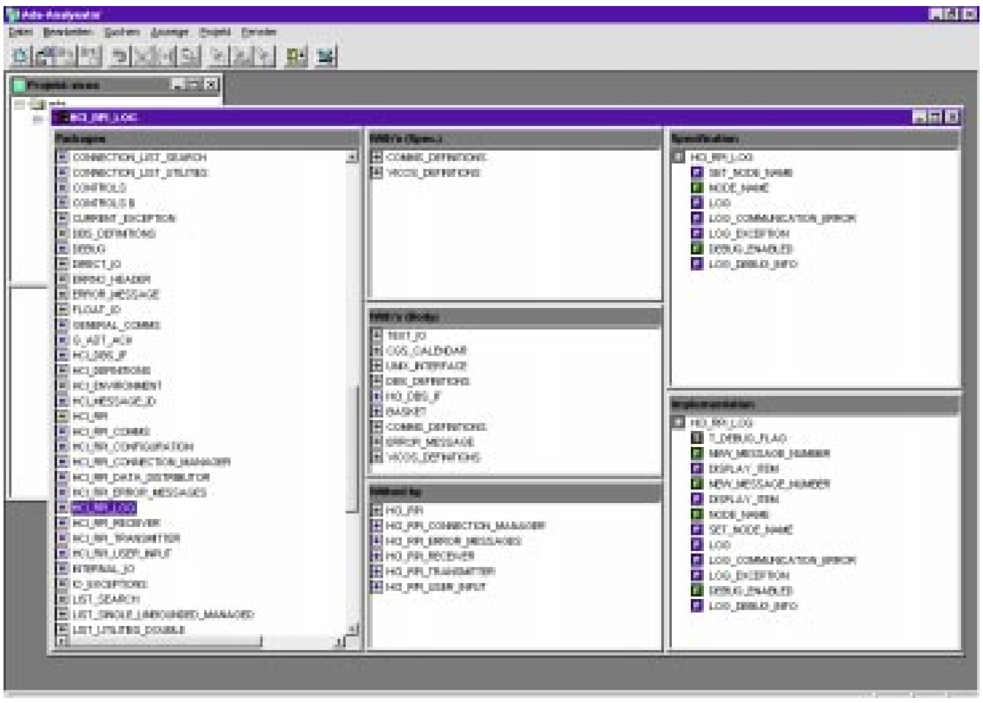
In summary, the ADA-Analysator represents a solid basis for the development of own tools. The desired scope of the analysis could easily be adapted by making slight changes in the code. Nevertheless, in parallel to the ADA-Analysator there is a second approach at Daimler-Benz which is described in the following section.

3.1.2 ADAlyzer

The working method of the ADAlyzer is quite similar to that of the ADA-Analysator which is described above. There is also a parsing-component which controls the construction of a dynamic data-structure representing the code. As before, the data-structure could be reused by subsequent analysis routines.

The ADAlyzer is also implemented in C/C++, but here the parsing-component is based on a LEX/YACC-parser which uses an ADA95-grammar. The advantage of this is that a basis for this new language standard already exists. Due to ADA95s extensive downward-compatibility to ADA83, it is possible as a rule, to analyze ADA83 programs, too. There were no problems in that respect during the analysis of the DASA case study.

The parser builds a C++ data-structure from the analyzed code. Due to the fact that it has a tree structure, it is called Abstract Syntax Tree (AST) in this case. For the construction of such an AST, we assume that the root-node of the tree



Packages

CONNECTION_LIST_SEARCH
CONNECTION_LIST_UTILITIES
CONTROLS
CONTROLS B
CURRENT_EXCEPTION
DBS_DEFINITIONS
DEBUG
DIRECT_IO
ERRNO_HEADER
ERROR_MESSAGE
FLOAT_IO
GENERAL_COMMS
G_AST_ACR
HCI_DBS_IF
HCI_DEFINITIONS
HCI_ENVIRONMENT
HCI_MESSAGE_ID
HCI_RR
HCI_RR_COMMS
HCI_RR_CONFIGURATION
HCI_RR_CONNECTION_MANAGER
HCI_RR_DATA_DISTRIBUTOR
HCI_RR_ERROR_MESSAGES
HCI_RR_LOG
HCI_RR_RECEIVER
HCI_RR_TRANSMITTER
HCI_RR_USER_INPUT
INTERNAL_IO
IO_EXCEPTIONS
LIST_SEARCH
LIST_SINGLE_UNBOUNDED_MANAGED
LIST_UTILITIES_DOUBLE

With's (Spec.)

COMMS_DEFINITIONS
VICOS_DEFINITIONS

With's (Body)

TEXT_IO
CGS_CALENDAR
UNIX_INTERFACE
DBS_DEFINITIONS
HCI_DBS_IF
BASKET
COMMS_DEFINITIONS
ERROR_MESSAGE
VICOS_DEFINITIONS

Withed by

HCI_RR
HCI_RR_CONNECTION_MANAGER
HCI_RR_ERROR_MESSAGES
HCI_RR_RECEIVER
HCI_RR_TRANSMITTER
HCI_RR_USER_INPUT

Specification

HCI_RR_LOG
SET_NODE_NAME
NODE_NAME
LOG
LOG_COMMUNICATION_ERROR
LOG_EXCEPTION
DEBUG_ENABLED
LOG_DEBUG_INFO

Implementation

HCI_RR_LOG
T_DEBUG_FLAG
NEW_MESSAGE_NUMBER
DISPLAY_ITEM
NEW_MESSAGE_NUMBER
DISPLAY_ITEM
NODE_NAME
SET_NODE_NAME
LOG
LOG_COMMUNICATION_ERROR
LOG_EXCEPTION
DEBUG_ENABLED
LOG_DEBUG_INFO

Figure 2: Ada Analysator

corresponds to the whole system being analyzed.

A system consists of at least one or more data-files. In an AST they are represented as nodes which are located one level higher than the root node. Every (ADA) file contains a set of Context-Clauses (With-Clause, Use-Clause) and/or Compilation-Units (Packages, Subprograms, ...). Again these elements represent the next level of the AST, although in such a way that they can be related to the file they are defined in (one AST-level below). The single Compilation-Units generally define, on their part, a namespace in which different ADA elements, such as packages, type-definitions, etc. can be redefined. The main difference between these elements is that they form either an entity on their own, or that they redefine a (nested) namespace which can itself contain various elements. According to their nesting-level, the single elements are assigned to their corresponding AST levels. This series continues iteratively until the maximal nesting-level of a Compilation-Unit is reached. The division of a system into different levels is a special kind of implicit Clustering (Grouping).

Together with special iterator and search algorithms, such a data-structure forms the basis for all of the subsequent higher-level analysis tasks and assessments. At this time it is possible to generate a Dependence-Graph from a system of multiple files. This graph could be visualized with the layout-toolkit VCG, which is already in use by a number of the FAMOOS-partners.

An example of the dependency relationships of approximately 300 ADA files is shown in figure 3.1.2. Every file is represented by a small square. The single filenames are displayed by clicking on the corresponding square. The arrows between the squares show the dependencies between the different files. One file depends on another, if it imports Compilation-Units from this file via With-Clauses, or if it contains an implementation and the corresponding specification is located in the other file.

In figure 3.1.2 the simple user-interface of the ADAlyzer (behind the VCG-window) is shown too. Due to the fact that this interface is implemented with the portable GUI-Framework V [Wam97], the

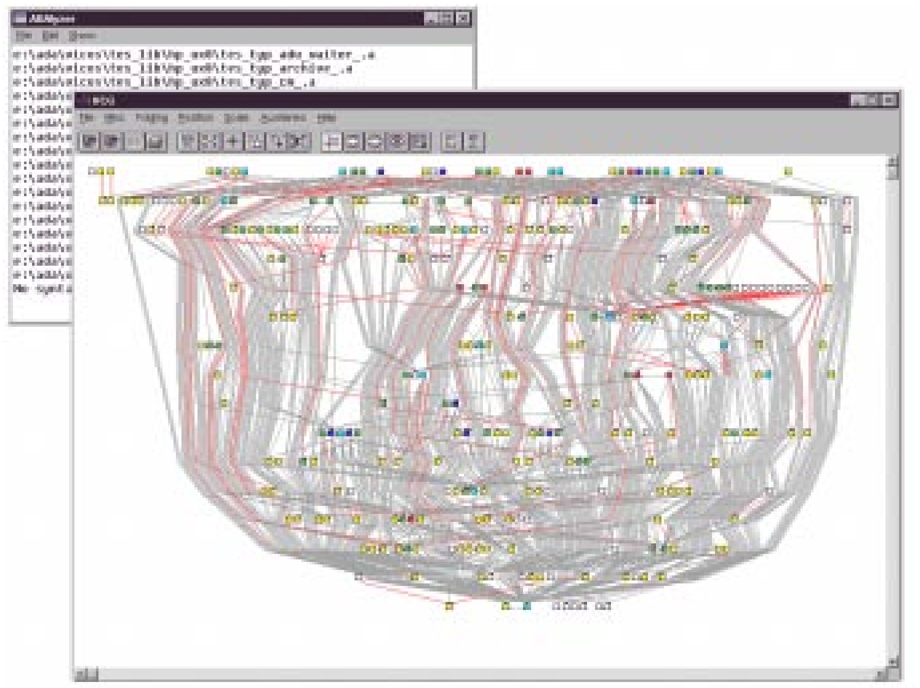


Figure 3: ADAlyzer

ADAlyzer is already available for both WIN95/NT and UNIX.

In summary it is possible to say that the ADAlyzer is a further foundation (especially for ADA) for a more extensive Analysis-, or Re-Engineering-Tool.

3.2 A Browsing Tool for class relationship

The goal of the development of this tool is to achive a focused view into an object-oriented software. Various types of relationships between classes, such as inheritance, has-relations and uses-relations, should be representable in one diagram. The tool should enable showing the whole system, but also browsing interactively through the software.

SNiFF+ was used for the realization [sni] SNiFF+ is a programming environment for C/C++ and other languages by TakeFive Software. It provides a programming interface for the extraction of symbolic information from C++ code. Unfortunately, the Sniff+ programming interface only offers information about the inheritance and the has-relations. Therefore, the capabilities of the tool are limited. As the data sructure provided by Sniff+ is very inflexible, classes for graph representation were developed. The graph is shown via the VCG tool. The graph data-structures are converted into a text file with the graph description for VCG. The VCG tool itself is controled by the browsing program. The classes which are to be shown are chosen from a list. There are two options how the classes are represented:

- the selected classes only and
- the selected and the related classes

A new graph is extracted from the whole system according to the selection and the data is forwarded to VCG. An illustration for this is given in figure 3.2.

Future work is focused on integrating another, language independent, layer (probably a UML repository) into the tool architecture. The graph classes are not sufficient to represent software structures, since a graph, as it is used here, is only one possible view into a real system. Another issue is

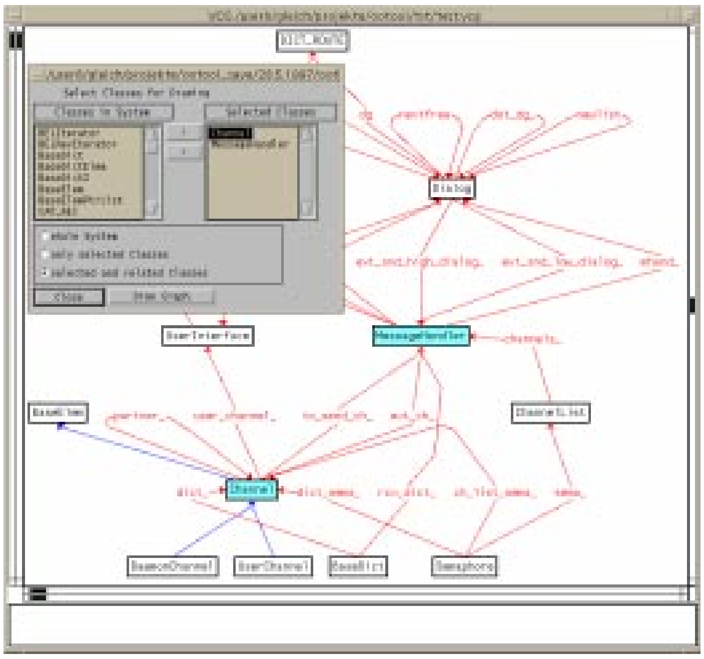


Figure 4: Browser for object-oriented systems

to include more, especially dynamic, information into the generation of views. The major problem here is not "How to get more data?", but "How to use the provided information?". Furthermore, the graph data-structures have to be extended using subgraph relations in order to support grouping of classes or other items.

4 Summary

In this document we presented our ideas for reengineering with the focus on tool support. The first steps for the implementation of prototypes have been accomplished. The developed tools mainly support the earlier stages of reverse engineering. Future work should take a step into the later phases of the reengineering life cycle. Another task within the FAMOOS project is the better integration of the methodology and the respective tools, since the acceptance of these methods in the future by the potential users depends on an extensive tool support.

References

[Doe97] Markus Doebele. Erkenntnis als Vorraussetzung erfolgreichen Wandels. *OBJEKTspektrum*, pages 41–51, April 1997.

[DP90] H. Dobler and K. Pirkelbauer. Coco-2, A New Compiler Compiler. *ACM SIGPLAN Notices*, 25(5), 1990.

[loo] Look! - C++ Visualisation and Debugging. http://www.objectivesoft.com.

[Nar97] Wolfgang Narzt. Analyse-Framework für Ada-Programme. Master's thesis, Universität Linz, May 1997.

[sni] SNiFF+ The industrial-strength programming environment for Unix C and C++ development. http://www.takefive.com.

[Wam97] Bruce E. Wampler. *V - A C++ GUI Framework*, January 1997. http://www.cs.unm.edu/ wampler.