Understanding Software – Static and Dynamic Aspects

Welf Löwe
IPD, Universität Karlsruhe
PF 6980, 76128 Karlsruhe
Germany
loewe@ira.uka.de

Andreas Ludwig
IPD, Universität Karlsruhe
PF 6980, 76128 Karlsruhe
Germany
ludwig@ipd.info.uni-karlsruhe.de

Andreas Schwind
RapidSolution Software GmbH
Haid-und-Neu Str. 7, 76131 Karlsruhe
Germany
schwind@rapidsolution.de

Abstract

We present an approach to software visualization supporting the understanding of structure and behavior of software systems. To do so, we merge information from static program analysis with dynamic information obtained during the execution of the programs. The merged information is presented graphically in different views, where users can interactively choose between more abstract or more concrete ones. This technique is implemented in an analyzing tool; its open architecture enables the easy integration of new views such as metric or profiling views.

Introduction

Understanding programs is essential for the development, maintenance and reengineering of software as well as for teaching purposes. These issues are diverse - single view on the programs is hardly sufficient. Instead, multiple views combining different aspects of the software ought to be used.

Views can be static or dynamic views: Static program information captures the program structure, but even elaborated analysis techniques obtain only little information on the behavior of the program in advance. Dynamic information visualizes the behavior of an example run of the program.

Especially in the field of software reengineering, a combination of static and dynamic views is highly desirable: the major task here is the identification of components and essential communications between them in legacy code.

Components are larger units of "coherent" modules or classes. The notion of coherency usually mixes static and dynamic system properties: it includes structural connection between the modules or classes in the call or inheritances graphs (static information). Additionally, it requires strong interactions between the modules or classes by procedure calls (dynamic information).

The essential communications between the components define the transfer of data independently of their implementations by simple calls, via shared memory accesses, by events, or callbacks. Finding these essential communications in legacy codes is a prerequisite to port them to component architectures like Corba or (D)Com or to transform them into a web service. However, analyses only picture the implementation of the communication. Static structure analysis often comes to misleading results: Assume that the communication is implemented by an event-listener pattern. The source of the communication provides a method called by the target to add itself as an event listener. Note that the direction of this call is the opposite of the direction of the essential communication. Additionally, the event source captures the listeners in a container of abstract listener objects. There is usually no static type information pointing back to the communication target. This connection is only visible via the object identifiers captured in the communication source: Such information is runtime information.

The combined analysis of static and dynamic aspects of a software system is also necessary for detecting hot spots in the program. Static views provide for an overview of the system while dynamic analysis profile the part that are extensively used. Those parts are first candidates for restructuring or optimization.

Combining static and dynamic aspects has already been established for software documentation. A typical means to illustrate programs are UML diagrams. There are a couple of different diagram types, such as static class diagrams, but also the dynamic sequence diagrams which developers have found useful for communication. UML specifications show how a software system should behave, but not how it actually does.

For understanding large software systems, we have to reduce the amount of information displayed. Filters as well as aggregations perform such a reduction:

Information filtering: users want to disregard parts of the software system or phases of the execution. They should be able to define, enable and disable filters.

Information aggregation: Even if some information is analyzed, it is not necessarily displayed in detail. A single representative entity can visualize several computed objects and relations. Observers, e.g., are often interested in the coupling of classes instead of viewing each involved reference between them. The user should enable or disable the aggregation dynamically.

Filtering reduces the computed information and, hence, the run time of analyses. However, if filtered information turns out necessary later, the static analysis must be performed again and the software must be restarted for the dynamic analysis.

Aggregated relations are always available. As aggregation is computed dynamically, it may be crucial for the analysis performance of large software systems.

We developed and implemented an architecture for the visualization of software structures and behavior. The *VizzAnalyzer* combines dynamic and static aspects of the software system using a front-end for the static analyses, a debugger for dynamic analyses and a visualization framework for the graphical presentation of the combined information. It allows for information filtering and aggregation.

Approach

Designing views of software is not a trivial task: the static view requires static analysis of the software system usually done by compilers or similar tools. The dynamic view needs the execution of the software for some relevant input data. Additionally, we need some adequate graphic tools providing figures for the static or dynamic information. Such a graphic tool is adequate if it allows abstraction and concretization of the information. Otherwise, it would fail in large applications. Finally, some infrastructure should glue together the information gathering and the imaging devices. The analysis tool architecture uses two more general frameworks:

- The *Recoder* for providing the static information on software systems,
- The *VizzEditor* for providing dynamic information and the actual visualization tools.

The *Recoder* framework performs static analyses and program transformations. The former is used in our architecture; the latter is disregarded so far. Beside the standard implementations, users may integrate their own analysis algorithms and transformations.

The *VizzEditor* framework supports the rapid design of visualizations of general program runs. It maps information from a debugger interface to graphic views. Users may use the standard views or add special views appropriate for their programs. Figure 1 displays the component diagram of the *VizzAnalyzer*.

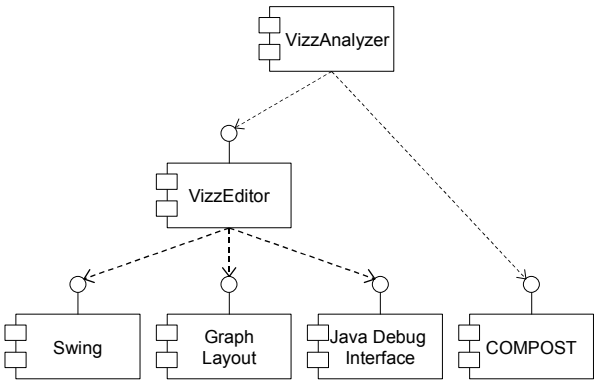

Figure 1: Components of the *VizzAnalyzer*

Entities and Relations Displayed

For most object-oriented languages, entities of interest include methods, constructors, attributes, classes, packages, and objects. Objects occur at run time only, while the other entities are static by nature. Relations between them are also either static or dynamic. Table 1 gives an overview on the relations and distinguishes static and dynamic ones.

The relations are defined as follows: A method `n` *refers to* a method or attribute `m`, iff there is a call or access operation to `m` in the body of `n`. A method `n` *calls* a method `m` or *accesses* an attribute `m` in a certain program run, iff a call or access operation to `m` in the body of `n` is executed in that program run. The *containment* relation follows the nesting of the corresponding entity definitions. An object `x` *knows* an objects `y`, iff an attribute of `x` contains a reference to `y`. The *subclass* relation is included in the static class definitions. An object `x` is an *instance of* a class `Y` iff its type attribute refers to `Y`.

Static Relations	Dynamic Relations
Refers-To(method, method)	*Calls*(method, method)
Refers-To(method, attribute)	*Accesses*(method, attribute)
Contains (class, attribute)	*Knows*(object, object)
Contains (class, method)	
Contains (class, class)	
Contains (package, class)	
Subclass-Of (class, class)	*Instance-Of* (object, class)

Table 1. Static and Dynamic Relations Computed by the *VizzAnalyzer*

The union of these entities and relations defines a graph with multiple node and edge types. Actually, there are three different graphs representing relations: the package graph, the inheritance graph, and the call graph. Despite of dynamically loaded classes, the package and inheritance graphs show static properties, whereas the call graph merges static and dynamic information.

All graphs are displayed with automatic layout algorithms. Depending on the static or dynamic nature of their elements (entities and relations), the graph is computed at compile time and updated at run time, respectively.

Except for trivial cases, it is not statically computable which method or attribute is actually called or accessed at run time and how often. Even sophisticated data flow analyses cannot predict all branches and loop iterations. Similarly, it is not statically computable which concrete method is called on a polymorph call. An additional problem is the detection of implicit method calls as they occur if a default constructor of a base class is called. As objects are created at run time, relations over objects are dynamic by nature. All these relations can only be computed dynamically for a concrete program run.

Despite of the variety of software systems, the data structures capturing the structure information and their

graphical views are the same in all applications. Therefore, a fixed set of *Recoder* analyses can be applied.

Moreover, potential program points of interest for the dynamic analyses are known in advance: we trace method calls, entries, exits, and attribute accesses. Constructors of objects are considered special methods. It is always clear how to update the graph views on the static analysis results at method calls, entry, exit, and attribute accesses events:

- Method calls and attribute accesses increase the weight of the edge corresponding to the respective *Refers-To* relation.
- A method entry and exit increases and decreases the weight of the node corresponding to the object or class it is defined in.

The weight of a graph node or edge can be related to its size or its color. Our instantiation of the *VizzEditor* framework implements the standard graphical views and the gluing infrastructure between the program and the views.

Filtering and Aggregation

The granularity of filters is the package level. Users can exclude entities and relations from contributing to the displayed graphs by adding their package names to the static or dynamic filter list.

Whenever a class, a method or another entity is found by *Recoder*, its package name is tested using the static filter list. It is disregarded if it belongs to an excluded package.

If a call occurs at run time, the caller and callee classes are tested against the filter list. If caller or callee is already statically filtered, the call is not traced at all.

Additionally, the caller and callee packages are compared with the dynamic filters. Then the corresponding methods and classes are included in the static analysis but the call does not trigger dynamic changes of the views.

This feature allows the user to concentrate the analysis on the hot spots of the program and last but not least it speeds up execution.

Aggregation follows the subclass, containment and instance-of relations. A superclass may represent subclasses, a class its class members or objects. A problem occurs with multiple inheritances where, in general, the representative superclass is not unique. An easy solution is to forbid such aggregations. Alternatively, we can resolve the ambiguity by conventions: in Java, e.g., we would prefer the super*class* as a representative to the super*interfaces*. Finally, we could simply insert a copy of edges to aggregated classes to each superclass. Aggregation allows to define arbitrary levels of abstraction over the actual relations. The users may choose more abstract or more concrete views on their software systems.

Actually, we aggregate in both directions of the inheritance hierarchy: we can hide either the children or the parents of a class, cf. Figure 2. If we collapse the children of a class X, we redirect to X all edges starting and ending in a transitive child of X. If we collapse the parents of a class X, we redirect to X all edges starting and ending in a transitive parent of X. Collapsing parents stops at the first transitive super class, which has more then one child, i.e. we do not copy edges.

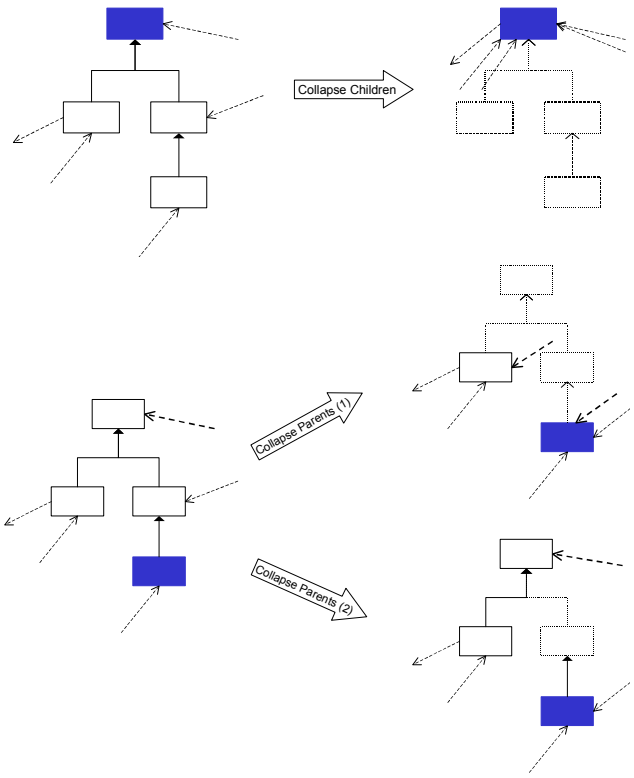

Figure 2: Aggregation of children (top) and parents (bottom - (1) with edge copying and (2) without)

In addition to the aggregations along the relations defined by the program, users can define custom relations for aggregation. For optimizing a package structure with respect to the communication frequency internal communications are maximized while communications to other packages are minimized. Then, it is desirable to define new, temporary relationships modeling possible packages. The effects of each "virtual" package structure are visualized before performing the corresponding reorganizations at hand.

Information Flow in the System

The program to analyze is first loaded and analyzed with *Recoder*. The static relations are visualized with the respective graphs. Then the program is executed. Whenever a relevant program point (method call, exit, entry, attribute access) is executed, the *VizzEditor* receives an event. If no filter applies, the corresponding graph is updated. The following sections discuss more details.

Whenever *Recoder* detects a new type, its package is added to the view model maintaining the package structure. Obviously, this primarily happens during the static analysis. However, the package view model is also updated if a class is loaded dynamically. The views for the inheritance, the

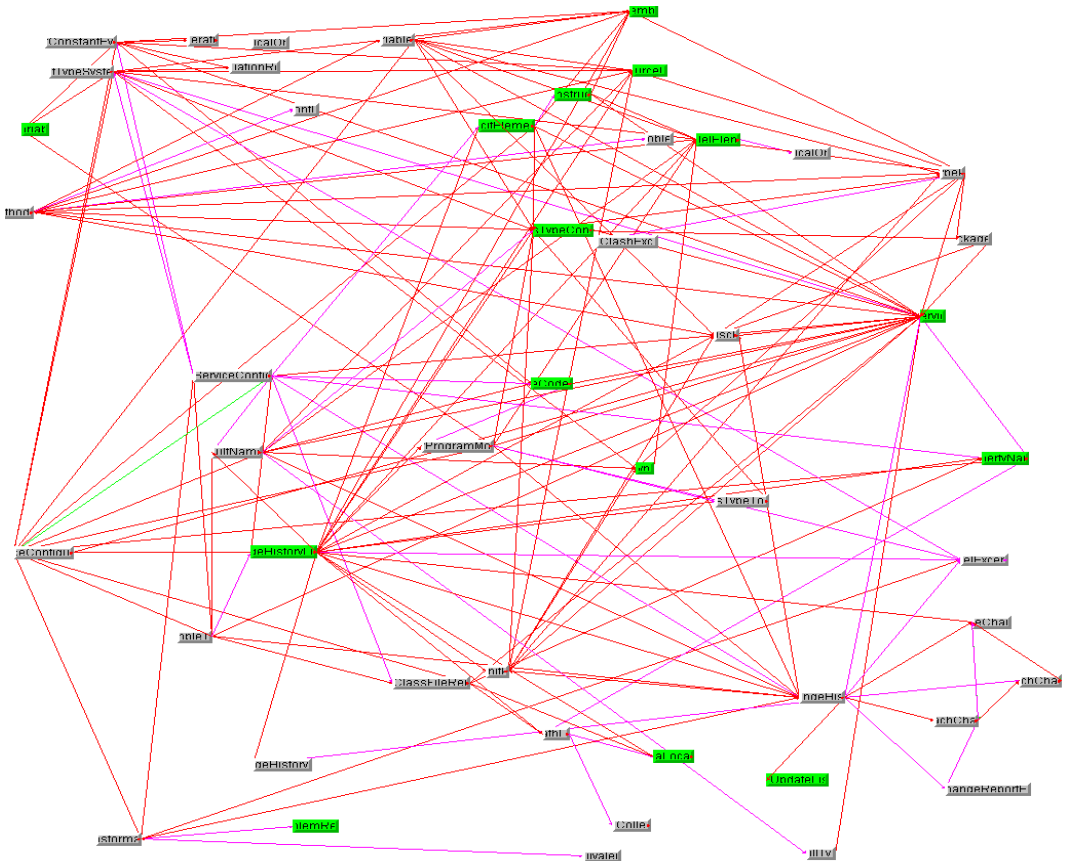

Figure 3: Static call graph of *Recoder* analyzed by *Recoder*; view filtered and aggregated along method containment and inheritance relations.

reference and the call relations work analogously to the package view. Graph drawing is done automatically using one of the predefined layout algorithms. These algorithms include upward drawings and spring embedding algorithms. The user also may stop the automatic layout and rearrange the nodes with the mouse.

After the static analysis the three graph structures are displayed. Then the program is started and the *VizzEditor* registers itself as a listener to the Debug Interface of the Java Virtual Machine. Whenever a method call of a method occurs and the call is not filtered, the graphs are modified.

The user is able to stop the execution of a program to inspect arbitrary program states in more detail. While the visualizations are frozen, the user can exploit the usual debugger functionality. The current position in the program execution is not only marked in the source code view but also in the graphical views: the node representing the current program position is marked. Additionally, the user obtains information on the nodes (representing packages, classes, objects, or methods) and edges (representing relations) displayed in the views.

The user may now resume the program execution or proceed step-by-step.

Results

To demonstrate the practical results, we analyzed the *Recoder* system itself. As *Recoder* contains more then 600 classes and 80.000 lines of code, this example shows the ability to handle with large systems. Figure 3 shows the call structure of *Recoder*. First filtering reduces the amount of classes – we filtered utility and system classes.

Further reduction is due to aggregation: we collapsed all methods and a number of sub- and superclasses. As classes and interfaces as well as normal and constructor calls are colored differently, the graph appears quite clear.

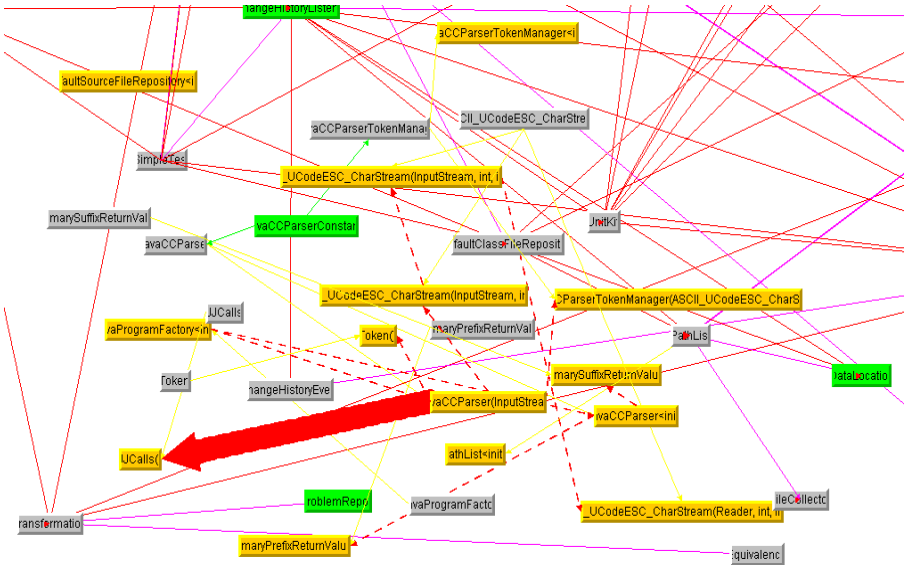

Figure 4: Zoom into the call graph. No aggregation along method containment; the fat edge indicates many run time calls.

Figure 4 shows a zoom into the call graph. Here, we displayed the methods. We observe heavy dynamic interactions in one part of the program indicated by the fat call edge (calls from the parser to the lexer – a well known hot spot for compiler front-ends), while other parts are not active so far.

Related Work

Many approaches and tools support the understanding of software structures. Some of them provide graphical views on the structure. *Together* from (TogetherSoft 2000) additionally supports round-trip engineering for Java and C++. This development suite performs a static analysis of the source code and visualizes the structural information as UML diagrams using an automatic graph layout.

Aggregations used in static visualization tools compress the displayed information similar as we do. *Goose* (O. Ciupke 1999) gives a static view on C++ program structures. In addition to a pure visualization, it detects patterns indicating design problems. Other approaches go further in their abstractions and compute software metrics on the structure. A good overview give (H. Bär et al. 1999) and (S. R. Chidamber and C. F. Kemerer 1994). (S. Demeyer, S. Ducasse, and M. Lanza 1999) define an architecture to combine software metrics and visualizations.

Online debugging and profiling techniques are state of the art. Visual debuggers support the understanding of program behavior by graph structures. *GraphTrace* (M. F. Kleyn and P. C. Gingrich 1988) computes static and dynamic views on an object-oriented LISP derivate. However, the static information is obtained by reflection. Hence, the tool visualizes only those program parts that are actually executed. Further static information relies on user annotations. Their architecture requires interpreted languages.

The approach of (W. De Pauw 1993) also graphically visualizes dynamic information. They use an object-to-class aggregation. The same authors developed (*Jinsight* 2000). It instruments a Java VM to access dynamic information about Java programs.

Scene (K. Koskimies and H. Mössenböck 1996) computes UML scenario diagrams from Oberon program executions. Their aggregation goes beyond ours: they collapse call sequences to one representative node. Their current work deals also with Java programs.

The work of (T. Ball and S. Erick 1996) combines static and dynamic information. It focuses on very large systems. It abstracts using metrics and statistics. For the visualization, it pictures scalable, colored charts instead of graphs. The tools operate post mortem; they can detect hot spots but cannot focus on them interactively or change views on the fly.

Conclusion

We presented an approach to support the understanding of software. It merges results from static program analysis with dynamic information about program execution. This combined information is presented visually with graph structures. We argued that neither static nor dynamic views by themselves provide an adequate understanding of software systems.

Large software systems require a reduction of information displayed at a time. We discussed filtering and aggregation techniques. The former reduces the data computed in the static analysis the latter reduces the information in a view while the data is still present in the view model. Both techniques are applied in our approach.

We instantiated two frameworks, one for analyzing static information from program representations (*Recoder*), the other for extracting dynamic information from program executions and for the visualization of information (*VizzEditor*). The resulting tool, the *VizzAnalyzer*, implements the approach of merging static and dynamic views on programs. It supports the analysis of design flaws and supports the analysis phase of the reengineering process in software systems.

In the future, the architecture could be used as a visual tool to also control source code modifications. *Recoder* already provides structures and methods for source code transformations (meta programming). Often, the effects of such operations are hard to understand especially if they are done at run time (dynamic meta programming). The *VizzAnalyzer* could visualize the transformations as well as the new program behavior. Additionally, user defined aggregation could be used to trigger a restructuring of packages or subclass hierarchies.

Another direction of future work is the integration of metric or other more abstract views into the architecture. The integration of statistic profiling views would open new fields of applications like program optimizations.

The *Recoder* software is developed in an open source project. It can be downloaded from:

http://recoder.sf.net

The *VizzEditor* and *VizzAnalyzer* home page is:

http://i44pc29.info.uni-karlsruhe.de/VizzWeb

A more elaborated presentation of the *VizzAnalyzer* approach and tool gives (A. Schwind 2000).

Acknowledgements

The kernel of our architecture provides the frameworks *Recoder* (allowing static analysis and automatic source code transformations) and *VizzEditor* (allowing the rapid design of visualization of arbitrary algorithms). We thank the students who contributed to the implementation: Stefan Arnold, Alex, Liebrich, Stefan Mandel, Heiko Schreiber, and Raphael Volz. We also thank the co-authors of *Compost* System (the origin of the *Recoder*) Prof. Uwe Aßmann and Dr. Rainer Neumann. The *VizzEditor* project was partially supported by the *ViKar* project, granted by the Land Baden-Württemberg, Germany.

References

O. Ciupke. Automatic Detection of Design Problems in Object-Oriented Reengineering. *Proceedings of Technology of Object-Oriented Languages and Systems - TOOLS 30*, pp. 18-32, IEEE Computer Society, August 1999.

H. Bär; O. Ciupke; S. Demeyer; S. Ducasse; R. Marinescu; R. Nebbe; T. Richner; M. Rieger; B. Schulz; S. Tichelaar and J. Weisbrod. The Famoos object-oriented reengineering handbook, *Report of the ESRPIT Program Project no 21975*, October 1999.

M. F. Kleyn and P. C. Gingrich. GraphTrace – understanding object oriented systems using concurrently animated views, *Proceedings OOPSLA '93*, pp 191-205, ACM SIGPLAN Notices, volume 23, number 11, November 1988.

S. Demeyer, S. Ducasse, and M. Lanza. A Hybrid Reverse Engineering Approach Combining Metrics and Program Visualization. *Proceedings of the Working Conference on Reverse Engineering WCRE'99*, IEEE, 1999.

Jinsight: http://www.research.ibm.com/jinsight.

T. Ball and S. Erick. Software visualization in the lagre. *IEEE Computer*, pp. 33-43, April 1996.

S. R. Chidamber and C. F. Kemerer. A Metric Suite for Object-Oriented Design. In *IEEE Transaction on Software Engineering*, 20(6), pages 476-493, 1994.

K. Koskimies and H. Mössenböck. Scene: Using scenario diagrams and active text for illustrating object-oriented programs. Proceedings of the 18th International Conference on Software Engineering, pp. 366-375. IEEE Computer Society Press, 1996.

TogetherSoft: http://oi.com.

W. De Pauw, R. Helm, D. Kimelman and J. Vlissides. Visualizing the Behavior of Object-Oriented Systems, *Proceedings OOPSLA '93*, pp 326-337, ACM SIGPLAN Notices, volume 28, number 10, October 1993.

A. Schwind 2000. Visualisierung von Strukturen in Softwaresystemen. (German) Diploma Thesis, Fakultät für Informatik, Univ. Karlsruhe.