Software Comprehension – Integrating Program Analysis and Software Visualization

Welf Löwe Morgan Ericsson Jonas Lundberg Thomas Panas

Software Technology Group
MSI/University of Växjö
Vejdes Plats 7, SE 351 95 Växjö, Sweden
{Welf.Lowe|Morgan.Ericsson|Jonas.Lundberg|Thomas.Panas}@msi.vxu.se
http://www.msi.vxu.se/~rics

Abstract— **We advocate that successful software comprehension methods (and tools) need the synergy of low-level code analyses known from the field of compiler construction, high-level analyses from the field of re-engineering and software visualization techniques. We argue that each individual technique would be either not goal directed or too shallow (or both). After a thorough state-of-the-art analysis and a problem discussion, we propose an approach to integration.**

I. Introduction

Understanding the architecture of a legacy system is essential for the further development, the maintenance and the re-engineering of the system. Unfortunately, architecture is hardly documented in such systems. The only trustworthy source of information is the system implementation. Hence, the architecture has to be retrieved from this source. As real world legacy systems tend to be large, the source code cannot be read directly. Instead, one uses (semi-)automatic *program analyses* to extract the information.

As these analyses are not unique by nature, system engineers have to be involved to accept or reject certain results proposed by the automatic analyses. Hence, the result of such analyses ought to be presented in a form that is intuitive to the system engineer. Therefore, the program analysis must go hand in hand with interactive *software visualizations*.

The combination of program analyses and software visualization techniques is crucial to succeed in comprehending legacy systems' architectures: plain program analysis results are hard to capture by the software engineers. Moreover, the analyses have to be controlled by the software engineers interactively, making it necessary to assess intermediate results. Such an assessment is preferably based on software visualizations. However, plain visualization of the system's structure (e.g. the abstract syntax tree) or behavior (e.g. program traces) cannot provide the right information to gain a system's architecture. Software engineers would drown in the flood of information. In order to achieve an architectural understanding we must therefore reduce the complexity of the original system by program analysis. Implicitly this means that we reduce the amount of information in order to achieve comprehension at the cost of accuracy.

A. Taxonomy

The major task in software architecture comprehension is the identification of components and the essential communications between them. Given the source of a legacy system, program analyses ought to be able to identify components and communications. However, the achievement of this goal is a non-trivial task involving various analyses techniques and representations as shown in Figure 1.

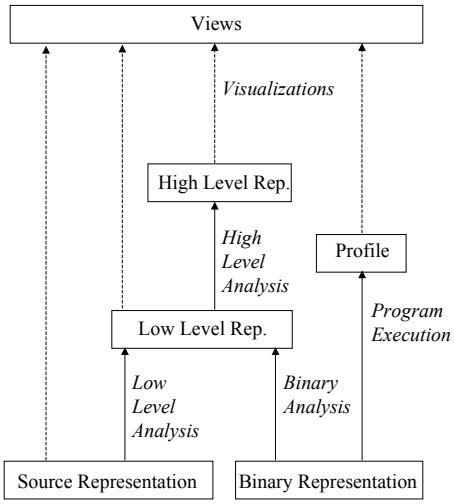

Fig. 1
Taxonomy of analyses techniques and representations.

We denote analyses techniques as *high-level analysis* if designed to retrieve a view on the systems that is more abstract than the source code. Examples of such techniques are dominance, coherence, concept, and metric analysis, as well as pattern recognition. Each one of these techniques provides a *high level representation* of the system at hand (e.g. call graphs, control flow graphs, class interaction graphs, lists of possible component or pattern candidates, and architectural description languages – ADLs). The visualization on that level include dominance tree drawings, dendograms, UML diagrams, graph browsers with "collapsing" capabilities, and component/connector graph drawings.

High level analysis is usually not applied directly on source level representations like for example Java, Java

byte. Most papers dealing with high level analysis assume that some *low level analysis*, like lexical and syntactical analyses as well as typing and data-flow analyses, already have been applied, and that the system is given in some kind of *low level representation*. Examples of low level representations are abstract syntax trees (AST), basic block graphs, and single static assignment (SSA) representations. The most significant difference between low and high level representations is that low level representations usually contain the same amount of information as the source representation, and that the low level analysis therefore often is reversible. However, the low level representation is still much to complex to provide any architectural insight for any non-trivial program. *Low level visualization* tools like AST explorers or graph browsers are therefore most useful for people that develop high level analysis techniques.

A somewhat separate technique to achieve program comprehension is based on execution of the actual system at hand. We refer to this approach as *run-time analysis*. Examples of run-time analysis are profile or event traces from an program execution visualized, e.g. in debugging or profiling tools, or call graph drawings based on a single program execution.

Furthermore, we distinguish between *structural*, *behavioral*, and run-time information. With structural information we mean the basic static information that can be retrieved by compiler front ends. That is, syntactic and rough type information. Behavioral information is the result of data flow analysis that allows us to derive more accurate approximations of the actual types, call graphs (with resolved polymorphic calls), and control flow graphs associated with the system. Finally, we have run-time information that we gather during single, or multiple, executions of the program at hand. The run-time information can be considered as exact although not complete behavioral information.

Finally, legacy systems are not always completely available in source code but partially only in binary, e.g. libraries, run-time systems, foreign components of the shelf. This requires *binary analyses* of those parts before structural or behavioral information can be retrieved.

B. Article Focus and Overview

The focus of this paper is the necessary synergy of different levels of analysis. We argue that the final result of any effort within program comprehension not only depends on the high level analyses and their visualizations, but also on the accuracy and the performance of the underlying low level analyses and their representations. Furthermore, the performance (memory usage and speed) of the underlying low level analyses are in many cases the limiting factor for what can, and cannot, be done in practice.

The remainder of the article is organized as follows. Section II gives a review of certain software architecture topics relevant for program comprehension. We also point out that both structural and behavioral information is needed in order to recover the architectural structure of a software system. After an introduction to the notion of software architecture – the goal of software comprehension – in Section II, the following sections (III to VI) give a thorough state-of-the-art analysis of various approaches to binary code, low level and high level analysis, as well as software visualization. Section VII points out various problems within each of these areas and in Section VIII we discuss how the program comprehension process can benefit from an integration of different levels of analysis.

II. Software Architecture Recovery

The goal of program comprehension is to recover the abstract architectural representation of a given system. A system's architecture is defined by its components, communications among them and the containment relation of components in larger components [1] [1], [2], [3], [4], [5], [6]. We define components to be software artifacts with typed input and output ports. This definition focuses on computational components, but is sufficiently general to cover all other variants. Components are identified as larger units of "coherent" modules or classes. The notion of coherency usually mixes static and dynamic system properties: it includes structural connection among the modules or classes in the call or inheritance graphs (static information). Additionally, it requires strong interactions between the modules or classes by actual calls (behavioral information).

In a software architecture, the essential communication among components is given in terms of ports and connectors. A port defines points in a component that provide data to its environment and require data from its environment, respectively. A connector defines out- and in-ports to be connected and specifies whether data is transported synchronously or asynchronously [7]. Some connectors may be as complex as most components, and thus require the same amount of consideration in design, but they are all based on simple point-to-point data paths. Figure 2 sketches this basic component model.

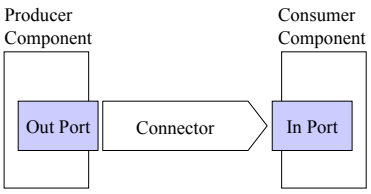

Fig. 2
Basic Component Model.

In practice, ports and connectors are implemented by patterns using basic communication constructs like calls, RPCs, RMIs, input/output routines etc. provided by the implementation language or the component system. The Observer Pattern, e.g., is such a port and connector implementation as it connects an event generator with some listener objects. Hence, we must be able to identify these communication patterns in order to recover the architectural view of a system's inter-component communication (i.e. the connectors). Again, static structure analysis alone

[1] The notion of components is recursive.

is insufficient. It often comes up with misleading results regarding source and target of the communication: Assume, e. g., 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. Moreover, 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 behavioral information.

Let us summarize our conclusions. Software comprehension means in practice to identify individual components and communication patterns. As sketched above, both components and essential communication among them, are defined in terms of (static) structure and (dynamic) behavior. Hence, software comprehension require both static and behavioral information in order to be successful.

III. Binary Code Analyses

The goal of the binary analyses described here is to recreate as much of the information contained in the original program as possible. The process to recreate this information is very much like the compilation process. This means that the first analysis is a syntax analysis that identifies the machine dependent opcodes and parameters that are stored in the binary object. Once this is done the result is subject to a semantical analysis. The role of the this semantical analysis is among other things to identify basic type information and figure out what instructions or groups of such actually do.

While the syntactical and semantical analyses may be complicated, this is an area that is well researched since every tool that operates on binaries, for instance re-compilers, needs to do these analyses. [8] gives a good introduction and overview of the area and what has been done. Since these analyses are always needed, some work has been done to automate, or at least simplify them, for instance SLED, Specification Language for Encoding and Decoding, [9], and SSL, Semantic Specification Language [10]. Some work has also been done to simplify the processing of different binary object file formats, for instance BFD, Binary File Descriptor, and BFF, Binary File Format [11].

IV. Low Level Analyses

We denote the classic analysis techniques known from the field of compiler construction as low level. Their origin is the translation and optimization of programs. However, they can support the understanding of systems on higher levels of abstraction than the source code. We distinguish analyses for checking the well-definedness of programs from behavioral analyses of program runs. The former is done by compile front-ends the latter by data flow analyses.

A. Syntax and Static Semantic Analyses

Classic compilers do name and type analyses and operator identification in order to check the well-definedness of the programs, to bind operations or names to their definitions and, if applicable, to insert type adaptations [12]. In meta-programming systems, e.g. *Together* [13], *Compost* [14], *Recoder* [15], this information is accessible by an API. The above analyses are usually done on Abstract Syntax Tree (AST) representation of the program.

B. Data Flow Analyses

Classic approaches of static program analysis rely on the theory of monotonous data flow frameworks [16]. In each node of the basic block graph, they compute a analysis value of a lattice $L = (M, sup)$ of possible values. Therefore, they assign each node i a monotonous transfer function $f_i : L \rightarrow L$ representing a abstraction of the code in that basic block wrt. to an analysis problem. Initially, one assigns an appropriate value from L for the start node and $\perp$ to the other nodes. The analysis is then performed iteratively by actualizing the values $v_i \in L$ of nodes i where $v_i = f_i(sup_{j \in pred(i)} v_j)$, with $pred(i)$ as the set of predecessor nodes of i in the graph. Provided the transfer functions are monotonous, this approach is guaranteed to terminate with a meaningful fix point. Depending on the definition of $\top$ and $\perp$ in L and on the definition of *sup*, we can distinguish optimistic and pessimistic analyses ("may" and "must" analysis). Depending on the orientation of edges in the basic block graph, we can distinguish forward and backward problems.

Anyway, to improve convergence, the updates of values should be computed along the dependencies induced by basic block edges. As the graph is cyclic in general, interval analyses [17] or the computation of connected components [18] precede the data flow analyses in order to determine an appropriate update order.

A general approach to determine and to prove correct transfer functions is called abstract interpretation, [19]. Therefore, the abstraction α represented by the transfer function is formalized and a concretization γ is defined . Both must build a Galois Connection. Together with the program semantics $[\cdot]_i$ of node i, the transfer function is: $v_i = \alpha[\gamma(sup_{j \in pred(i)} v_j)]_i$.

Although the classic basic block graph is defined procedure by procedure, data flow analyses can also be performed in an inter-procedural way. Therefore, the basic block graph is extended to an inter-procedural control flow graph containing additional edges between call node and callee basic block graph and back. A problem with this extension is that some paths are never executed at runtime (e.g. paths from a call x to the callee and back to another caller y) but they contribute to the results of the analysis.

Context sensitive analyses [20] avoid the negative influences of non-executable paths on the analysis results. Therefore, one distinguishes analysis results reaching a node via different paths, approximated by the path' last k nodes. Contexts may be derived from previous analysis information: [21] and [22], e.g., define an analysis context as the result of a context insensitive analysis.

PAG — Program Analysis Generator [23], [24] and *Optimix* [25] are two frameworks for generating data flow anal-

yses. While *PAG* works on the basic block graph and actually implements monotonous data flow frameworks, *Optimix* works with arbitrary graphs and actually implements a general class of graph transformations. Many analyses may be formalized by graph transformations that add edges to the basic structure (so called edge addition transformations). This is still a subclass of the *Optimix* transformations.

Control flow analyses are specific data flow analyses. They try to determine the target of jumps and calls. In object oriented languages, it is necessary to determine the dynamic type of callees, i.e. the type of the qualifying expression of a call. In the best case, the static type of these expression is a sufficient approximation. In general, however, these static types are polymorphic. To reduce the set of dynamic types of objects bound to the qualifying expressions, we have to prove that some objects are never bound to these expression at runtime, cf. [26], [27].

Inter-procedural constant analysis [28], [29] attempts to determine the value of the type-tag of objects. In the presence of dynamic object generation and object references, this approach early reaches its limits. [30] abstract objects by types of the source program. Using context sensitive and inter-procedural analyses (that distinguish all possible types of actual call parameters with a separate context), the set of dynamic types of a callee is restricted as far as possible. The high level of this abstraction prevents more exact heap object analyses. However, it allows to analyze programs with approx. 30.000 lines of code.

[22] distinguishes between different contexts if and only if it produces different analysis results avoiding unnecessary redundancies. Still the memory consumption is high for the distinguished contexts; the approach breaks for programs with more then 1.500 lines of code.

[31] define type analyses for functional languages using constraints. Their flow-sensitive analyses do not generalize for imperative languages. However, the most appealing idea in this approach is that they start with fairly abstract analyses. After the fix point is reached, the level of abstraction is lowered for the hot spots and fix point iteration starts again. Programs with approximately 2.000 lines of code are admissible for this approach.

Pointer analyses (actually Points-To analyses), cf. [32], [33] strive to analyze the value of pointer expression. *Heap analyses* are restricted to analyze the pointer to heap objects. [34] analyses context-insensitive but in linear time. Programs with more then 75.000 lines of source code are controllable. However, the analyses distinguish not more objects than the static object generation statements in the program. The current record for context- and flow-insensitive points-to analyses is held by [35] with approximately 1.000.000 lines of C-code per second.

The analysis of [20] is context-sensitive. It distinguishes call path (contexts) with fixed size k; for $k = 2$, however, programs with approx. 8.000 lines, for $k = 3$ maximum 4.000 line are admissible. As discussed already, systems with fine grained components contain rather lengthy indirections and call path. Therefore, $k = 3$ hardly contributes to an improvement in the analysis results.

[36] analyze the structure of the heap. They pay attention on the efficient storage of the analysis information: the heap shape is stored by deltas at program points where it actually changes. [37] extends this approach to a context sensitive analysis. Their implementation, however, couldn't go beyond 500 lines of code.

[38] propose a name schema denoting anonymous objects by a set of local variables uniquely pointing to them. It is restricted to procedures. The practical use is limited as programs with more than 100 lines have not been controlled yet.

The current state of the art in points-to analyses is also discussed by [39]; this work also identifies scalability, precision of the analyses and their applicability in open systems as open problems.

Data flow analyses could also be done on the Abstract Syntax Tree (AST). For effective and efficient dataflow analyses, it is necessary to understand the essential data dependencies and distinguish them from those induced by program construction. Modern intermediate representations in *Static Single Assignment (SSA)* [40] form compute essential data dependencies once and store them as edges in the a code graph. Its extension Memory-SSA [41] allows to represent and analyze also object oriented programs.

V. High level Analyses

The low level analysis described above usually results in some kind of a basic block graph where the individual blocks are interconnected with different dependencies. These block graphs can be more or less precise depending on the accuracy of the low level analyses. However, for any non-trivial real-world program this structure is still much to complex to provide the developer with any architectural insight. In order to achieve an architectural understanding we must therefore reduce the complexity of the low level representation by means of high level analysis.

We denote analyses techniques as high-level if designed to retrieve a view on the systems that is more abstract than the source code. Implicitly this means that we reduce the amount of information in order to achieve comprehension at the cost of accuracy. The problem of high level analysis is therefore to determine what information that is actually needed in order to gain system understanding, and how to extract it.

The high level analysis can roughly be divided into two steps: The first step uses the result of the low level analysis to derive abstractions involving classes (or objects) interconnected by various dependencies (e.g. inheritance, method calls, and aggregation). We refer to this abstraction level simply as the class level since it usually involves classes interacting with each other.[2] On this level, classes are sometimes cloned to distinguish different subsets of in-

[2]We could have referred to this abstraction level as the UML level since UML diagrams are a well-known method to visualize such relations. However, we think it is important to separate the representation (e.g. interacting classes) from methods of visualization, (e.g. UML class diagrams). Hence, we stick with the term class level.

stances (objects) of classes and their interaction. The latter type of class level representation is today rarely used since it requires extensive (and very memory consuming) data flow analysis in order to derive a low level representation that can separate different instances of a class. Exceptions are the object graph presented in [42] and the object model presented in [43].

The second step of high level analysis pushes the level of abstraction even further. It usually starts at the class level and uses different techniques to identify individual components, connectors, or well-known design patterns - or at least possible candidates. Established techniques to identify component candidates are different clustering techniques based on metrics, dominance analysis or concept analysis, as well as pattern detection used to identify well-known, and often used, communication patterns:

Pattern Detection In forward engineering, design patterns [44] are used as good standard solutions for the implementation connectors with certain properties. Inversely, one can try to detect the patterns in the code in order to understand the intended high level connector. This is the approach most relevant for program comprehension. The same technique is used to find design flaws. Here one looks for so called anti-patterns [45].
[46] employs code instrumentation to extract dynamic information to analyze and transform architectures. The presented approach only identifies communication primitives, but no complex protocols. The Goose system [47] is a re-engineering tool that gives a graphic visualization of C++ program structures using a similar approach for their detection. It uses aggregations for its static visualizations to compress the displayed information. Additionally, it detects patterns indicating design problems. [48] present structured analyses to discover design patterns (Template Method, Factory Method and Bridge) from C++ systems. They identify the necessity for human insight into the problem domain of the software at hand, at least for detecting the Bridge pattern due to the large number of false positives. The Pat system [49] detects structural design patterns by extracting design information from C++ header files and storing them as Prolog facts. Patterns are expressed as rules and searching is done by executing Prolog queries. In [50] we detect Observer and other connector patterns in legacy code. The static analysis computes a set of connector candidates while the dynamic information is used to support or reject the static decision. This analysis is implemented as an extension of our VizzAnalyzer framework [51][3].
Metrics The basic strategy to identify components here is to look for sets of classes with high internal cohesion and low external coupling. The used set of metrics to measure coupling and cohesion, as well as the algorithm to find candidate component sets will effect the result. A common approach is to identify a "central class" recognized by high internal complexity and strong external coupling and to iteratively merge it with its neighbors if they together form a group with lesser external coupling.
This approach is for example taken in [52]. A good overview of metric based methods is given in [53] and [54]. [55] defines an architecture to combine software metrics and visualizations. [52] uses object oriented metrics to reduce the complexity of UML diagrams. In the research project presented in [56], reverse engineered Java systems are represented as graphs using the Chidamber-Kemerer suite of metrics [54].
Dominance Analysis is a graph theoretic method that often have been applied to call graphs in imperative languages to identify so called dominant nodes (i.e. nodes that are the only entry point to a subsection of the graph). The dominant node and the subsection they dominate are then suitable module candidates since they represents a set of procedures with a low external coupling. This idea can easily be transferred to an object oriented setting by applying it to a graph of classes interacting with methods. In that case should the presence of a dominating node indicate a component interface candidate.
[57] perform dominance analysis on the call graph of C programs to organize systems as a hierarchy of modules, where modules are architectural component candidates. The proposed algorithm is based on static information only. Hence, they have problems especially with the function pointers. [58] extends the work presented in [57] by further heuristics.
Concept Analysis This general technique is often applied to software re-engineering. The concept lattice is computed on a relation of (abstract) objects and attributes. In the context of software understanding, this could be algorithm classes and data structure classes resp., in the feature analysis, e.g., these are use cases and classes, respectively. The analysis computes maximum sets of objects using the same set of attributes. There is an order relation defined on the maximum sets forming the concept lattice.
[59], e.g., combines static and dynamic feature analysis to aid program understanding. The basic technique is concept analysis to derive correspondences between features and components (sets of subprograms). First dynamic analysis, performed by a profiling tool, identifies subprograms dealing with a feature. Concept analysis then yields relationships between features and executed subprograms. These subprograms serve as starting point for a more detailed static analysis along the static dependency graph (transitive closure on the call graph, detection of strongly connected components and dominance analysis) to narrow the set of executed subprograms to those that form self-contained and understandable feature-specific components.

The basic approach to use the above techniques are similar in spite of their separate appearances. It is a bottom-up approach starting from some class level representation (usually a graph) and then, using the above mentioned techniques to identify components (or patterns), "collapses" those entities into a single node, and continues the search.

It is important to realize that the final result of the high level analysis very much depend on the accuracy of the un-

[3]As the VizzAnalyzer is actually a software visualization framework, it is described in Section VI.

derlying low level analysis. For example, pattern detection often requires the capability to identify individual objects (i.e. behavioral information). Pattern detection based on structural information usually end up with a rather large set of possible candidates and then uses manual inspection, or dynamic information based on executions, to examine each one of the candidates. This shows that some high level analysis requires behavioral information in order to be automated. There are other areas where behavioral information probably would increases the accuracy of the outcome. For example, dominance analysis ought to give a better result if the low level analysis could resolve polymorphic method calls.

VI. Software Visualization

The discipline in computer science dealing with the construction of static and dynamic views on software systems is called software visualization. The visualization of static pictures or dynamic movies in general is vital for the process of understanding compressed information. This observation continues to hold in computer science, where graphical metaphors are used in the design and documentation of software systems. Graph drawing is therefore a major part in software visualization.

A. Graph Drawing

A graph drawing algorithm takes as input a graph and computes a layout, i.e. a drawing assigning coordinates to the vertices and mapping each edge to a simple curve. The layouts should be "aesthetically nice" and "easy-to-understand". Some important criteria for readable diagrams are a small number of edge crossings and bends in edges, uniformly distributed vertices and edges, short edges, a small layout area or volume, symmetry, and a good angular resolution. These aesthetics are such that optimality of one may prevent optimality in others. Additionally, graph layout algorithms in general can be viewed as optimization problems and are typically NP-complete or NP-hard [60]. These two observations suggest a heuristic approach to general graph drawing for many applications.

There is a wide variety of graph drawing methods that take different aesthetic criteria into account. Drawing methods can be classified according to the kind of drawings they produce (e.g., hierarchical drawings, orthogonal drawings, straight-line drawings, circular drawings), the model they use (e.g., force-directed, planarization), and the class of graphs they can be applied to (e.g., planar graphs, directed acyclic graphs, trees).

Dynamic graph drawing addresses some interesting problems in information visualization: navigation in large graphs, display of dynamic data represented by graphs, and interactive graph editing. Current graph drawing systems usually employ static (batch) layout algorithms, although stable dynamic layout seems to be essential in the context of software comprehension.

Directed graphs are usually drawn with Sugiyama's algorithm [61] or one of its numerous variations or improvements. Given the wide application of graphs structures in display, there are relatively few algorithms for drawing general undirected graphs [62]. This, in part, is due to the inability to specify the aesthetic criteria individuals use in understanding graphs [63]. Nonetheless, for certain restricted classes of graphs in which graph-theoretic expressions of aesthetic criteria can be specified, satisfactory algorithms have been developed [64], [65], [66]. Comprehensive reviews can be found in [60] and [62].

Quite a few academic products implement these drawing algorithms. E.g. cvg and vcg with a simple textual interface to describe the graphs producing static drawings or, more advanced, Graphlet and AGD, a library of Algorithms for Graph Drawing, (based on the LEDA C++ library and Tcl/Tk) with a programmable interface that can also create dynamic drawings.

B. Software Visualization

An overview on software visualization in general is given in [67]. The present paper, however, focus on those approaches important for the field of software comprehension.

In the previous sections we discusses analyses of structural and behavioral aspects of a software system. We argued that it could be crucial for the understanding of a system to have both kind of views. Hence, software visualization techniques should be able to display both structural and behavioral information.

For understanding large software systems, we have to reduce the amount of information displayed. Filters as well as aggregations perform such a reduction. Filtering disregards parts of the software system and reduces the computed information and, hence, the run-time of analyses. Aggregation on the other hand combines some analyzed information, so that it is not necessary to display all in detail.

On source code level, current program editors, aspect editors and source browsers provide a views that are able to aggregate and filter information. These tools are often integrated with online debuggers and profilers provide behavioral information. Visual debuggers support the understanding of program behavior by providing additional graphical structures.

On class level, many approaches and tools support the visualization of software structures. These approaches fail to merge in behavioral information. We discuss some exceptions below.

GraphTrace [68] computes static and dynamic views on an object-oriented LISP derivative. 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.

[69] uses dynamic information, analyzing the flow of messages. His approach is restricted to detecting design patterns in Smalltalk, since he only regards flows in VisualWorks for Smalltalk.

The VizzAnalyzer [51], our own tool, combines structural and behavioral information of Java software system on source and class level. It uses a compiler front-end for

the static analyses, accesses the Java debugger interface for dynamic analyses and visualize the combined information with various graphical views. Also, it allows for information filtering and aggregation.

The approach of [70] also graphically visualizes dynamic information. They use an object-to-class aggregation. The same authors developed JInsight [71]. In contrast to the VizzAnalyzer, JInsight instruments a Java VM to access dynamic information about the programs.

For the design and documentation of new large systems, software visualization has been established as the tool of choice to control their complexity. A typical means to illustrate programs are UML diagrams combining graphical views of static and dynamic aspects of software. Of particular interest are UML class diagrams whose purpose is to display class hierarchies (generalizations), associations, aggregations, and compositions in one picture. The combination of hierarchical and non-hierarchical relations poses a special challenge to a graph layout tool. Commercial software typically uses Sugiyama-style, e.g. Rational Rose [72]. Together [13] additionally supports round-trip engineering for Java and C++. Both development suite perform low-level analyses of the source code to visualizes the structural information as UML diagrams. Unfortunately, these methods that cannot automatically distinguish between hierarchical and non-hierarchical relations.

Besides static class diagrams there are a couple of different other diagram types, such as the dynamic sequence and activity diagrams which developers have found useful for communication. Scene [73] computes UML scenario diagrams from Oberon program executions. They aggregate call sequences to one representative node. Their current work also deals with Java programs. However, these views show how a software system behaves in some runs, but not how it actually does in general. It is unclear how to derive such views with static analyses.

Above class level, we have quite a few graphical metaphors capturing the information. Dominance tree, program trace or concept lattice as well as architecture description languages have natural views.

However, surprisingly few tools actually support the automatic visualization, especially when it should include behavioral information. The work of [74] is an rare exception. It combines static and dynamic information and focuses on very large systems. Their aggregation use 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.

VII. Problem Discussion

In this section, we discuss the problems of each individual technique when applied to the problem of software comprehension.

A. Binary Code Analyses

The recovery of data- and control flow information from binary object is a relatively well understood process. A major problem is to recover procedures and procedure calls. Efforts to automate this process is described in [75], [76], but require that the compiler adhere to the ABI (Application Binary Interface) of the platform for all procedures, while the ABI only require that exported procedures adhere to the specification. If the compiler uses a internal format for non-exported procedures or inline procedures then recovery is extremely hard or even impossible, at least in the general case.

Another problem is the recovery of composite types, which is a very hard problem to solve in the general case. [77] and [78] describes to different efforts to recover types. While neither is good enough for out purposes a combination of the two that starts with the types we know from interface definitions, procedure signatures or APIs, similarly to Ohori and Katsumatas approach and then uses Mycrofts methods to calculate any types that the first attempt fails to recover.

B. Low Level Analyses

Syntactic and static semantic analyses are very easy to perform. However they do provide only very limited insight on the dynamic behavior of the system (one could understand the static types as a very rough approximation of the system behavior). Even the structural information could be improved with more advanced analysis techniques.

The current state of the art in data flow analysis is not satisfactory regarding software comprehension. There are open problems in both program analysis and representation of analysis results. Representations below AST level are hardly assessable to human designers. They are therefore not appropriate for iterative development cycles. The memory consumption of analysis results is still the main problem for handling large systems. However, we observe redundancies, especially in the representation of results of context-sensitive analyses as often distinguished contexts do not provide distinguished analysis results. The exceptions [37], [41] only control small example programs. The analyses themselves are either too inefficient or too ineffective (sometimes even both). The reason for that is a statically predefined precision of the analyses with the exceptions of [31], [41]. Interleaved different analyses improves the overall precision. This is disregarded by many approaches, attempts [79] and more elaborated [41], [80]. As it is unclear how to update the analyses results after such transformations, expensive analyses have to be repeated, hindering an iterative approach with stepwise refined analyses and transformation in-between. Finally, analyses on binary code level are necessary to assess the behavior of libraries and runtime systems as well as legacy components that are not available in source code. Here, we cannot base suitable analyses at all.

In some cases, however, dataflow analyses are too precise and detailed for the actual problems in software comprehension [42]. They provide a lot more information than actually needed on higher level.

C. High Level Analyses

High-level analyses usually deal with structural information only as they are targeted to large systems. Because of their complexity, precise data flow analysis simply cannot be applied, [20].

To compensate for this lack of dynamic information, some approaches (see above) use profile information from system runs. However, this only provides information that is strongly depending on the profiled use cases and, even worse, on the used data sets in the use cases. It is therefore unclear, how the profile information generalize.

High-level analyses are not unique in the sense that they provide the originally intended architecture of a system. Instead, they propose reasonable options and the final decision is up to a systems engineer. The information, however, is not directly assessable to a human user. Although many papers suggest a visual representation of the information, usually graphs, an automatic drawing of large graphs is not a trivial task, see below.

Finally, quite a few high level analyses are published and comparative studies between different techniques do not favor a specific one [58]. Instead, each individual analysis outperforms the others in certain systems. Hence, the best approach is unknown as long as the system is. The lack of a general method triggers the question for a reasonable combination of the single analyses, which is not answered yet.

D. Software Visualization

The information computed by high level analyses must be displayed, however. The question is, how to draw this information automatically. This remains an open question despite the fact that there are quite a few graph layout algorithms appropriate for different kinds of single substructure: Graph drawing use heuristics to approximate the optimal layout of the graphs as the problem in general as well as most relevant subproblems translate to NP-hard optimizations. Hence, they compromise in drawing the graph structures. However, the trade-offs are not goal directed, as the algorithms have no notion of the semantics of the graphs drawn. Specific drawing algorithms making trade-off decisions regarding the information displayed are not known so far.

Moreover, if we were to draw a combination of graphs, consisting of substructures, each representing an analysis result from a different high- or low-level analysis method, we would face new problems: each drawing algorithm applied to the whole structure would deliver only sub-optimal results for each substructure.

VIII. Integration

We discuss the advantages of an integration of techniques in a demand driven way. We start with the interface to system engineers and go down to the binary code analyses.

A. Visualizations benefit from high level analyses

Filtering and aggregation allow to view only partial information about the system. However, if filtered information turns out necessary later on, analyses must be performed again. Aggregated information is always available. As aggregation is computed dynamically, it may be crucial for the visualization performance of large software systems. It is therefore necessary to have a determined notion of the understanding process in order to know what information to filter and to provide at a certain stage in that process. Hence, information reduction for software visualization is not possible without an understanding of the information displayed.

The information neither filtered nor aggregated must be displayed, however. This problem remains open despite the fact that there are quite a few graph layout algorithms appropriate for graphs of different structures: Tree layout algorithms draw the hierarchical package structures; rearrangement is no problem, as we know from file system explorer views. Upward drawings could layout the directed acyclic inheritance structures. Redrawing after aggregation and unfolding, respectively, is an open problem. For the directed and cyclic graphs depicting the call structure, spring embedding and simulated annealing algorithms appear appropriate. Redrawing could be traced by displaying also intermediate states in the force directed rearrangements. Graph drawing use heuristics to approximate the optimal layout of the graphs as the problem in general as well as most relevant sub-problems are NP-hard optimizations. When choosing the heuristics only depending on graph properties, the trade-offs are not goal directed, as the algorithms have no notion of the semantics of the graphs drawn. One could do better, if trade-off decisions would regard the information displayed.

Moreover, a drawing of the combination of graphs of different kinds poses new problems: each algorithm applied to the whole structure delivered only sub-optimal results for each substructure. We assume that at a certain stage in the process of understanding one structure (or even a substructure thereof) could dominate the others implying that the layout algorithm for this structure dominates the others in an appropriate view. Again, such improvements are only possible with an understanding of the information displayed.

B. High level analyses benefit from visualizations

High level analyses would benefit from an integration of software visualizations besides the obvious benefit of getting this information displayed.

Researcher in the field of re-engineering are aware of the situation that the high level analyses can only provide optional structures that are likely to represent the actual architecture. These options are to be assessed by a system engineer. Currently, there is little support for the comparative assessment of the single options. Software visualizations could bridge this gap. Dominance trees and the concept lattices, e.g., used in component recognition define partitioning of the systems classes implying their possible

grouping into components. So does the package structure. Each horizontal cut of the dominance tree, the concept lattice, and the package graph defines such a partitioning. To compare the results of the analyses, one should consider a simultaneous view of the partitionings induced by analyses. Although the actual results of analyses are incomparable data structures, they could be easily assessed by comparing the induced component groupings.

Moreover, system engineers understand the legacy system architecture and behavior only in terms of notions that are common also in forward engineering, e.g. in terms of components, connectors, classes, objects etc. Auxiliary data structures like dominance trees and the concept lattices, even if drawn nicely, do not directly trigger an understanding of the system. They have to be presented together with their induced system structures. System engineers should be able, e.g., to move the cut of the dominance tree, the concept lattice, up- and downwards in the hierarchies and see simultaneously the implications in terms of a system partitioning into different components.

C. High level analyses benefit from low level analyses

We have noted in section V that the accuracy of the high level analysis in general depends on the accuracy of the underlying low level analysis. Obviously, the program structure has to be retrieved with syntax and static semantic analyses in all methods of high level analysis. However, data flow analyses could sharpen the structural information. Call graph and type analyses, e.g., restrict the declared type of a call target to a subtype that defines the actually possible targets of that call. Another example is the distinction of objects of the same class leading to a cloning of that class (and a context dependent adaptation of each individual clone).

For high analyses requiring behavioral information one would expect low level data flow analyses to be the standard. Instead quite a few high analyses avoid the complex and complicated flow sensitive low level analyses. They rather try to work around, either by using flow insensitive analyses or by using dynamic information for sample executions. The former is conservative, the latter a optimistic approximation to the information that is actually needed. Integrated data flow analyses – if affordable in time and memory consumption – could improve the quality of the high level analyses.

D. Low level analyses benefit from high level analyses

The main limitation of the low level analyses are memory and time consumption. However, simple analyses and optimizations could cut down a system first and enable more advanced analyses. One define a refinement from simple flow- and context-insensitive to flow- and context-sensitive analyses where analysis results of one round should be propagated to the next. Refinement of analyses should only be performed in hot-spots of the system. What a hot-spot is and how far one should go with the refinement, however, depends on the analysis goal, which is defined by the analyses on the higher level.

Orthogonal to the stepwise refinement of the analysis, is a stepwise widening of the system parts analyzed. Instead of inspecting the whole system on the first try, one could pick up parts thereof. Each subsystem is assumed to run in an unknown environment. Step by step, we can relax these restrictions and provide more analysis information in these environments. Again, a suitable initial partitioning and the subsystems one would like to analyze (and merge later) is goal directed and, hence, depends on the high level analyses questions.

E. Low level analyses benefit from binary code analyses

In order to enable a closed world assumption in the low level analyses, binary code could be represented with a conservative approximation in terms of SSA code. This uses the fact that each foreign component behaves equivalent to a sequence of assignments interacting with the rest of the system via parameter and result objects. It is to find out by the binary analysis, which objects are read, which are written. In case of multi threaded components, the access could be even asynchronous to the call, which should be found out be the analyses, as well.

However, if control is passed to a binary component, usually all assembled analysis information is spoiled because one conservatively assumes everything could happen then. Hence, one strives to integrate the binary components in a less conservative way and to represent them by approximations that are conservative wrt. certain specific low level analysis goals. This in turn requires the integration of low level and binary analysis.

F. Binary code analyses benefit from high level analyses

Inversely, one need only to retrieve information from binary components that is actually relevant making the approximations less conservative. This is much more goal directed than just analyzing everything possible.

Moreover, binary code is always embedded in a system environment. Hence, interface, call protocol, and transaction behavior are defined within the source code of the caller. With low level analyses at hand, this information could be made available for the binary analyses.

IX. Conclusion

We analyzed the current state of the art in software comprehension. Therefore, we proposed a taxonomy. We classified the *techniques* to retrieve information – binary, low- and high level analyses and software visualization – and the information itself – structural and behavioral, cf. Section I. With this taxonomy at hand, we gave a structured overview about the numerous approaches in software comprehension, cf. Sections III – VI.

It followed a discussion of problems that remained unsolved in the single approaches, cf. Section VII. We identified the missing integration of analysis techniques with each other and with software visualizations as the major one.

Finally, we showed how software comprehension would benefit from such an integration. We argued, that the

interaction with system engineers would improve, results could be more precise and analyses more efficient. Despite the state of the art analyses, this is the main contribution of our paper.

However, such an integrated approach to software comprehension has not been achieved yet. It is the current and future work of our group. With these efforts, we contribute to *CATE* – the Component Analysis and Transformation Engine – a joint project between the Växjö university, the Technical University of Karlsruhe and the Research Center Computer Science in Karlsruhe, see `www.msi.vxu.se/~cate`.

References

[1] David Garlan and Mary Shaw, "An introduction to software architecture," in *Advances in Software Engineering and Knowledge Engineering*, V. Ambriola and G. Tortora, Eds., Singapore, 1993, vol. 1, pp. 1–40, World Scientific Publishing Company.

[2] M. Shaw and D. Garlan, *Software Architecture in Practice – Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[3] Len Bass, Paul Clements, and Rick Kazman, *Software Architecture in Practice*, Addison Wesley, 1998.

[4] Ralph Melton, *The Aesop System: A Tutorial*, School of Computer Science Carnegie Mellon University, `http://www-2.cs.cmu.edu/~able/aesop/aesop_home.html`, 2002.

[5] Shaw, DeLine, and Zelesnik, "Abstraction and Implementations for Architectural Connections," in *3rd International Conference on Configurable Distributed Systems (ICCDS '96)*, 1996, concerns Unicon.

[6] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann, "Specification and Analysis of System Architecture Using Rapide," *IEEE Transactions on Software Engineering*, vol. 21, no. 4, Apr. 1995.

[7] Dirk Heuzeroth, Welf Löwe, Andreas Ludwig, and Uwe Aßmann, "Aspect-oriented configuration and adaptation of component communication," in *Third International Conference on Generative and Component-Based Software Engineering, GCSE*, Jan Bosch, Ed. 2001, p. 58 ff., Springer, LNCS 2186.

[8] C. Cifuentes, *Reverse Compilation Techniques*, Ph.D. thesis, Queensland University of Technology, School of Computing Science, July 1994.

[9] Norman Ramsey and Mary F. Fernández, "Specifying representations of machine instructions," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 3, pp. 492–524, May 1997.

[10] C. Cifuentes and S. Sendall, "Specifying the semantics of machine instructions," in *6th International Workshop on Program Comprehension - IWPC'98, Ischia, Italy, June 24-26 1998.* 1998, pp. 126–133, IEEE Computer Society.

[11] D. Ung and C. Cifuentes, "SRL – a simple retargetable loader," in *Proceedings of the Australia Software Engineering Conference*. September 1997, pp. 60–69, IEEE CS Press.

[12] W. M. Waite and G. Goos, *Compiler Construction*, Springer, New York, 1984.

[13] *Together*, TogetherSoft, `http://http://oi.com`, 2001.

[14] Uwe Aßmann, *Invasive Software Construction*, to appear, 2001.

[15] Andreas Ludwig, *RECODER Homepage*, `http://recoder.sf.net`, 2001.

[16] Marlowe and Ryder, "Properties of data flow frameworks. A unified model," *Acta Informatica*, vol. 28, pp. 121–163, 1990, An overview of data flow frameworks and their characterizing properties is given. Contains many references to the field of data flow analysis.

[17] Steven S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997.

[18] Keith Cooper and Taylor Simpson, "SCC-based value numbering," Tech. Rep. CRPC-TR95636-S, Rice University, Oct. 1995.

[19] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points," in *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, New York, NY, 1977, pp. 238–252, ACM.

[20] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers, "Call graph construction in object-oriented languages," in *OOPSLA*, 1997.

[21] William Landi, Barbara G. Ryder, and Sean Zhang, "Interprocedural side effect analysis with pointer aliasing," *SIGPLAN Notices*, vol. 28, no. 6, pp. 56–67, June 1993, *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[22] W. Golubski, "Typanalyse fr objektorientierte programme," 1998, Habilitationsschrift an der Universit"at Siegen.

[23] Martin Alt and Florian Martin, "Generation of Efficient Interprocedural Analyzers with PAG," in *SAS'95, Static Analysis Symposium*, Alan Mycroft, Ed. September 1995, vol. 983 of *Lecture Notes in Computer Science*, pp. 33–50, Springer.

[24] *PAG — Experiencing Program Analysis*, Universitt des Saarlands, `http://pag.cs.uni-sb.de`, 2001.

[25] Uwe Aßmann, "OPTIMIX, A Tool for Rewriting and Optimizing Programs," in *Graph Grammar Handbook, Vol. II.* 1999, Chapman-Hall.

[26] Rainer Mauch and Martin Trapp, "Ein bersetzer und laufzeitsystem fr die objektorientierte programmiersprache sather," M.S. thesis, Universitt Karlsruhe, 1992.

[27] G. DeFouw, D. Grove, and C. Chambers, "Fast interprocedural class analysis," Technical Report TR-97-07-02, University of Washington, Department of Computer Science and Engineering, July 1997.

[28] Paul R. Carini and Michael Hind, "Flow-sensitive interprocedural constant propagation," *ACM SIGPLAN Notices*, vol. 30, no. 6, pp. 23–31, June 1995.

[29] Cooper, Hall, Kennedy, and Torczon, "Interprocedural analysis and optimization," *CPAM: Communications on Pure and Applied Mathematics*, vol. 48, 1995.

[30] Ole Agesen, "The Cartesian product algorithm: Simple and precise type inference of parametric polymorphism," in *ECOOP'95—Object-Oriented Programming, 9th European Conference*, Walter G. Olthoff, Ed., Åarhus, Denmark, 7–11 Aug. 1995, vol. 952 of *Lecture Notes in Computer Science*, pp. 2–26, Springer.

[31] John Plevyak and Andrew A. Chien, "Type directed cloning for object-oriented programs," in *Proceedings OOPSLA '94*, Oct. 1994, pp. 324–340.

[32] A. Deutsch, "Interprozedural may alias analysis for pointers," in *PLDI — ACM Sigplan Conference on Programming Languages, Design and Implementation*, 1994.

[33] M. Emami, R. Ghiya, and L. Hendren, "Context-sensitive interprocedural points-to analysis in the presece of function pointers," in *PLDI — ACM Sigplan Conference on Programming Languages, Design and Implementation*, 1994.

[34] Bjarne Steensgaard, "Points-to analysis in almost linear time," in *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages (POPL '96)*, 1996, pp. 32–41.

[35] N. Heintze and O. Tradieu, "Ultra-fast aliasing analysis using cla: A million lines of c code in a second," in *PLDI — ACM Sigplan Conference on Programming Languages, Design and Implementation*, 2001.

[36] David Chase and Mark Wegman F. Kenneth Zadeck, "Analysis of pointers and structures," in *SIGPLAN 90*, 1990, pp. 296–310.

[37] Uwe Amann and Markus Weinhardt, "Interprocedural heap analysis for parallelizing imperative programs," in *Proceedings of the Conference on Programming Models for Massively Parallel Computers*, S. Giloi, W. K., Jähnichen and B. D. Shriver, Eds., Los Alamitos, CA, USA, Sept. 1993, pp. 74–82, IEEE Computer Society Press.

[38] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm, "Solving shape-analysis problems in languages with destructive updating," in *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg Beach, FL, 21–24 Jan. 1996, pp. 16–31.

[39] M Hind, "Pointer analysis: Haven't we solved this problem," in *PASTE — ACM Sigplan/Sigsoft Workshop on Program Analysis for Software Tools and Engineering*, 2001.

[40] Ron Cytron, Jeanne Ferrante, and Vivek Sarkar, "Compact representations for control dependence," June 1990, vol. 25(6) of *ACM SIGPLAN Notices*, pp. 337–351.

[41] Martin Trapp, *Optimierung Objektorientierter Programme*, Xpert.press. Springer, 2001.

[42] Andr Spiegel, "Object graph analysis," Tech. Rep., 1999.

[43] Robert O'Callahan, *Generalized Aliasing as a Basis for Program Analysis Tools*, Ph.D. thesis, Scholl of Computer Science, Carnegie Mellon University, Pittsburg, USA, 2000.

[44] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Software Components*, Addison-Wesley, 1995.

[45] William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick III, and Thomas J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, John Wiley, New York, NY, 1998.

[46] S. J. Carriere, S. G. Woods, and R. Kazman, "Software Architectural Transformation," in *Proceedings of WCRE 99*, October 1999.

[47] Oliver Ciupke, "Automatic detection of design problems in object-oriented reengineering," in *Technology of Object-Oriented Languages and Systems - TOOLS 30*. 1999, pp. 18–32, IEEE Computer Society.

[48] Rudolf K. Keller, Reinhard Schauer, Sebastien Robitaille, and Patrick Page, "Pattern-based reverse-engineering of design components," in *International Conference on Software Engineering*, 1999, pp. 226–235.

[49] L. Prechelt and Ch. Krämer, "Functionality versus practicality: Employing existing tools for recovering structural design patterns," *J.UCS: Journal of Universal Computer Science*, vol. 4, no. 12, pp. 866ff, 1998.

[50] Dirk Heuzeroth, Thomas Holl, and Welf Löwe, "Combining Static and Dynamic Analyses to Detect Interaction Patterns," in *Proceedings of the Sixth International Conference on Integrated Design and Process Technology (IDPT)*, Jun 2002.

[51] "VizzAnalyzer," `http://www.msi.vxu.se/~vizz`, 2002.

[52] Ralf Kollmann and Martin Gogolla, "Metric-based selective representation of uml diagrams," in *Proc. 6th European Conference on Software Maintenance and Reengineering. IEEE, Los Alamitos, 2001*, 2001.

[53] H. Bär, M. Bauer, O. Ciupke, S. Demeyer, S. Ducasse, M. Lanza, R. Marinescu, R. Nebbe, O. Nierstrasz, T. Richner, M. Rieger, C. Riva, A.-M. Sassen, B. Schulz, P. Steyaert, S. Tichelaar, and J. Weisbrod, "The FAMOOS Object-Oriented Reengineering Handbook," Tech. Rep., Forschungszentrum Informatik, Karlsruhe, Software Composition Group, University of Berne, ESPRIT Program Project 21975, 1999.

[54] S. R. Chidamber and C. F. Kemerer, "A Metric Suite for Object-Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, June 1994.

[55] Serge Demeyer, Stéphane Ducasse, and Michele Lanza, "A hybrid reverse engineering approach combining metrics and program visualization," in *Proceedings: Sixth Working Conference on Reverse Engineering*. 1999, pp. 175–186, IEEE Computer Society Press.

[56] T. Systa, P. Yu, and H. Muller, "Analyzing java software by combining metrics and program visualization," 2000.

[57] Jean-Francois Girard and Rainer Koschke, "Finding components in a hierarchy of modules: a step towards architectural understanding," in *Proc. of the International Conference on Software Maintenance - ICSM '97.*, 1997.

[58] Rainer Koschke, *Atomic Architectural Component Recovery for Program Understanding and Evolution*, Ph.D. thesis, Institute for Computer Science, University of Stuttgart, 2000.

[59] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon, "Aiding Program Comprehension by Static and Dynamic Feature Analysis," in *Proceedings of the International Conference on Software Maintenance*. Nov. 2001, IEEE Computer Society Press.

[60] P. Eades and R. Tamassia, "Algorithms for automatic graph drawing: An annotated bibliography," Tech. Rep. CS-89-09, Dept. of Comp. Science, Brown Univ., 1989.

[61] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for visual understanding of hierarchical systems," *IEEE Trans. on Systems, Man and Cybernetics*, vol. SMC-11, no. 2, pp. 109–125, 1981.

[62] R. Tamassia, G. Di Battista, and C. Battini, "Automatic graph drawing and readability of diagrams," *IEEE Transactions on Systems*, vol. 18, no. 1, pp. 61–79, 1988.

[63] Peter Eades and Xuemin Lin, "Spring algorithms and symmetry," *Theoretical Computer Science*, vol. 240, no. 2, pp. 379–405, 2000.

[64] C. Batini, E. Nardelli, and R. Tamassia, "A layout algorithm for data-flow diagrams," *IEEE Trans. on Software Engineering*, vol. 12, no. 4, pp. 538–546, 1986.

[65] M. J. Carpano, "Automatic display of hierarchized graphs for computer aided decision analysis," *IEEE Transactions on Systems*, vol. 10, no. 11, pp. 705–715, 1980.

[66] L.A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis, and A. Tuan, "A browser for directed graphs," *Software - Practice & Experience*, vol. 17, no. 1, January 1987.

[67] John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, Eds., *Software Visualization: Programming as a Multimedia Experience*, MIT Press, 1998.

[68] Michael F. Kleyn and Paul C. Gingrich, "GraphTrace — understanding object-oriented systems using concurrently animated views," in *OOPSLA'88: Object-Oriented Programming Systems, Languages and Applications: Conference Proceedings*, Norman Meyrowitz, Ed., 1988, pp. 191–205.

[69] K. Brown, "Design reverse-engineering and automated design pattern detection in smalltalk," 1997.

[70] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides, "Visualizing the behavior of object-oriented systems," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1993, pp. 326–337.

[71] "Jinsight," `http://www.research.ibm.com/jinsight`, 2000.

[72] *Rational Rose*, Rational, `http://www.rational.com/products/rose/index.jsp`.

[73] Kai Koskimies and Hanspeter Mössenböck, "Scene: Using scenario diagrams and active text for illustrating object-oriented programs," in *Proceedings of the 18th International Conference on Software Engineering*. 1996, pp. 366–375, IEEE Computer Society Press / ACM Press.

[74] Thomas Ball and Stephen G. Erick, "Software visualization in the large," *IEEE Computer*, vol. 29, no. 4, pp. 33–43, 1996.

[75] Cristina Cifuentes and Doug Simon, "Procedural abstraction recovery from binary code," in *CSMR*, 2000, pp. 55–64.

[76] Mark W. Bailey and Jack W. Davidson, "A formal model and specification language for procedure calling conventions," in *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.*, New York, NY, 1995, pp. 298–310.

[77] Alan Mycroft, "Type-based decompilation," in *European Symposium on Programming*. 1999, vol. 1576 of *Lecture Notes in Computer Science*, pp. 208–223, Springer.

[78] Shin ya Katsumata and Atsushi Ohori, "Proof-directed decompilation of low-level code," in *European Symposium on Programming*. 2001, vol. 2028 of *Lecture Notes in Computer Science*, pp. 352–366, Springer.

[79] Cliff Click and Keith D. Cooper, "Combining analyses, combining optimizations," *ACM Transactions on Programming Languages and Systems*, vol. 17, no. 2, pp. 181–196, Mar. 1995.

[80] Sorin Lerner, David Grove, and Craig Chambers, "Combining dataflow analyses and transformations," in *Proceedings of the 29th Symposium on Principles of Programming Languages*. ACM SIGACT-SIGPLAN, 2002, vol. 37(1) of *SIGPLAN Notices*, pp. 270 – 282.