Analyzing Feature Implementation by Visual Exploration of Architecturally-Embedded Call-Graphs

Johannes Bohnet
University of Potsdam
Hasso-Plattner-Institute
Prof.-Dr.-Helmert-Str. 2-3
14482 Potsdam, Germany
bohnet@hpi.uni-potsdam.de

Jürgen Döllner
University of Potsdam
Hasso-Plattner-Institute
Prof.-Dr.-Helmert-Str. 2-3
14482 Potsdam, Germany
doellner@hpi.uni-potsdam.de

ABSTRACT
Maintenance, reengineering, and refactoring of large and complex software systems are commonly based on modifications and enhancements related to features. Before developers can modify feature functionality they have to locate the relevant code components and understand the components' interaction. In this paper, we present a prototype tool for analyzing feature implementation of large C/C++ software systems by visual exploration of dynamically extracted call relations between code components. The component interaction can be analyzed on various abstraction levels ranging from function interaction up to interaction of the system with shared libraries of the operating system. The user visually explores the component interaction within a multiview visualization system consisting of various textual and a graphical 3D landscape view. During exploration the 3D landscape view supports the user firstly in deciding early whether a call relation is essential for understanding the feature and, secondly, in finding starting points for fine-grained feature analysis using a top-down approach.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging – *Tracing*, D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *Restructuring, reverse engineering, and reengineering*, H.5.2 [**Information Interfaces and Presentation**]: User Interfaces – *Graphical user interfaces (GUI)*, I.5.3 [**Pattern Recognition**]: Clustering – *Algorithms*

General Terms
Design, Experimentation, Human Factors, Measurement, Verification

Keywords
Dynamic Analysis, Dynamic Slicing, Feature Analysis, Program Comprehension, Reverse Engineering, Software Visualization

WODA'06, May 23, 2006, Shanghai, China.

1. INTRODUCTION
As stated in [32] "year after year the lion's share of effort goes into modifying and extending preexisting systems, about which we know very little". Requests for software changes are often expressed by end users in terms of features, i.e. an observable behavior that is triggered by user interaction. Requesting feature changes concerns bug fixes or enhancements of feature functionality and is a key concept for maintaining, reengineering, and refactoring large and complex software systems. To implement these new requirements, software developers have to translate the feature change requests to changes in code components and their interaction behavior. The term *component* is used here in a more general way than that defined by the Unified Modeling Language (UML) [31] which describes it as an autonomous, modular unit within a system or subsystem that has well-defined interfaces and is replaceable within its environment. Here the term refers to a structural unit of source code of any level of abstraction and includes functions, classes, subsystems, and systems.

Before converting feature change requests to code changes, first developers have to identify the components that implement the feature functionality. Subsequently, they need to understand how these components interact. After that, they can finally modify the code and implement new functionality. Analyzing feature implementation is particularly difficult where legacy systems are concerned, as the existing documentation often differs substantially from the as-is system design due to the long evolution period [7]. Hence, in many cases the only reliable documentation on a legacy system is the source code itself which may consist of more than 1.000.000 lines of code (LOC).

Program slicing [34] is a common approach for reducing the search space the user has to inspect in order to be able to understand a specific functionality. Static slicing reduces a software system implementation to all statements that may directly or indirectly influence a set of variables at a specific code position (backward slice) or to all statements that may be influenced by a specific code statement (forward slice) [10]. Dynamic slicing further narrows the search space by only taking into account the statements that actually do affect a specific program state based on a given input [1]. Profiling techniques can extract dynamic call relationship among functions. These light-weight slices do not cover variable accesses as static slicing techniques typically do [7]. However, dynamic call-graph extraction techniques can reveal some call relations that are difficult to identify by static

analysis as they are defined at runtime [4]. Event driven systems, for instance, often used for graphical user interfaces (GUI) or network communications, are based on dynamic callback mechanisms. Furthermore, object-oriented programming paradigms such as polymorphism and dynamic binding make the understanding of static call sequences more difficult.

Along with the technique of extracting analysis data, an important usability aspect for an analysis tool is how to communicate the results to the user. Established visualization techniques for call-graph exploration are simple 2D graph drawings or just the user guidance through the textual representation of the source code. Some approaches abstract from the function level and visualize the interaction of selected classes, e.g. by creating UML sequence diagrams [9, 18, 29]. A prerequisite for this technique is that the user has a basic understanding of the feature implementation and knows the classes of interest. Additionally, UML sequence diagrams are limited to a small number of interacting classes.

In this paper we present a prototype tool for analyzing feature implementation by abstracting a dynamically extracted function call graph to call-graphs of higher-level components of the hierarchical software architecture, i.e. class, subsystem, and system components. The user explores this *architecturally-embedded call-graph* within an interactive multiview visualization system. Prerequisites for the analysis of a software system are firstly, the availability of the source code written in C/C++ and, secondly, an executable file that is compiled with debug information and can be executed under a GNU/Linux operating system (OS). The tool has been tested with software systems consisting of over a million LOC.

The tool guides the user during the call-graph exploration task. The user can analyze the call relations on different levels of abstractions, i.e. function interaction, class interaction, subsystem interaction, and interaction of the system with shared libraries of the OS. Provided with the opportunity of switching between abstraction levels, the user receives (a) support with finding specific functions as entry points for further fine-grained function level analysis (top-down approach) and (b) is given support with fast excluding function calls that are not likely to be helpful when trying to understand the feature implementation. This assessment is based on the higher-level component of which the function is a part.

2. RELATED WORK

Wilde et al. introduced *Software Reconnaissance* [35, 36], a technique for locating feature implementation based on a comparison of traces from test cases with and without feature execution. The analyzed source code needs to be instrumented. With *TraceGraph* [17] Lukoit et al. facilitate the *Software Reconnaissance* analysis process by providing a visualization technique that displays source file, subroutine, or lines-of-code activity, color encoded per time interval. Contrary to our approach, no component relationship is shown. Other research on locating feature implementation is based on *Program Dependency Graphs* [26] which are extracted by static analysis. *RIPPLES* [4] supports the user during manual exploration of a dependency graph by 2D graph visualization. The user decides whether a component, i.e. function, basic block, or statement, is relevant for the feature and adds it to the *search graph* that finally represents the feature implementation. In contrast to our approach, no decision hint by embedding the functions into a high-level structure is given. Eisenbarth et al. [7] first compare dynamic execution traces, which depend on a set of features, by applying *concept analysis* to find out to which feature a computational unit contributes. Later the user identifies additional feature specific units by exploring the statically extracted dependency graph. Our approach, which suggests the embedding of functions into the system architecture, is orthogonal to the described techniques above and can, accordingly, be combined with them. Furthermore, our user-supporting visualization technique may be used as visualization front-end.

A variety of visualization techniques are used to facilitate reverse engineering tasks. Some are based on the *SeeSoft* technique [5] of displaying LOC metrics as color encoded miniaturized source code lines, e.g. *Tarantula* [14], *Gammatella* [25], *Bee/Hive* [28]. *sv3D* [19] extends the technique to 3 dimensions. Other techniques, such as ours, are based on graph visualization. The *Rigi* [21] reverse engineering environment displays the hierarchical software architecture as a 2D layout of nested boxes with connecting straight lines. *SHriMP* [20] enhances Rigi views by means of elaborate navigation and exploration techniques and is used for visualizing statically analyzed Java programs. *CodeCrawler* [6] calculates various software metrics and encodes them in box shapes of simple graphs. Several graph visualizing techniques use virtual reality techniques: *NV3D* [27] displays nested cubes connected by tubes and can be used to visualize execution traces with animated arrows moving along the tubes. *CrocoCosmos* [16] creates a universe-like visualization of software components by a force-directed layout technique based on static software metrics. Similarly, *JST* [12] creates a software component universe by parsing Java source code and applying metrics. In the case of both approaches, the user explores the data by applying standard navigation techniques of a VRML browser. As our approach proposes, some visualization techniques use the landscape metaphor for presenting software systems structure and behavior. Zhou et al. [37] visualizes message flows in massively parallel supercomputers. Boxes regularly positioned on a plane represent processors and are connected by arcs. Balzer et al. [2] map the architecture of Java programs as nested box and sphere shapes positioned on a plane. Relations between components are represented by connecting arcs. This differs from our approach in that no self-organizing layout for encoding information on component relations is used.

3. ARCHITECTURALLY-EMBEDDED CALL-GRAPHS

Our prototype tool enables the user to analyze a dynamically extracted function call graph on higher-level abstractions, i.e. as class-class, subsystem-subsystem, or system-shared library call relations. By abstracting from functions to higher-level components, the number of interacting components the user has to inspect is reduced significantly. Depending on the analyzed feature, a function trace log may consist of more than 10.000 interacting functions.

Starting at a high level of abstraction, the user first analyzes how the system interacts with shared libraries of the OS in order to gain a rough understanding of the system's behavior. Then the user refines the analysis and explores the interaction behavior of those subcomponents of interest until the low-level function call abstraction is reached. This top-down approach helps the user to find low-level entry points for further fine-grained analysis as it reduces the search space the user has to examine.

Vice versa, a bottom-up approach can prove useful for an early identification of function calls of little interest. During the low-level step-by-step analysis from one function to another, the user is supported in his/her efforts to determine which call relation to follow. As each function belongs to a specific higher-level component, the user can refrain from further analyzing a function call if the target function belongs to a higher-level component that he/she earlier classified as being negligible for understanding feature functionality. Such a high-level component might be a subsystem that encapsulates IO functionality while the user is trying to understand the feature of coloring text in a word processor application.

The analysis process with the prototype tool is divided into three steps (see Figure 1):

1) In a semi-automated step a model of the system architecture is created (above class abstraction). An initial model is automatically recovered from the directory structure of the source code and can be manually refined by the user.

2) The user first identifies a scenario, i.e. a sequence of user interactions that triggers the feature execution, and second applies a logging mechanism while executing the scenario. For this, the regular executable file compiled with debug information can be used. No code instrumentation is necessary.
   The tool automatically analyzes the logged function call relations and embeds them into the architecture model. Function interactions are aggregated to interactions of higher-level components (function interaction, class interaction, subsystem interaction, system/shared library interaction) creating an *architecturally-embedded call-graph*.

3) The architecturally-embedded call-graph is visualized by an interactive multiview system. A combination of synchronized textual and graphical views enables the user to efficiently explore the call-graph. Two textual views focus on communicating the hierarchical structure of the architecture and on providing comprehensive information on the components and their call relations. The graphical view focuses on visualizing both the hierarchical structure and the call relations of components within this structure. With its clustering layout technique the graphical view supports the user in visually finding strongly coupled components, i.e. components with a high number of control flow transfers from one to another.

3.1 Architecture Model

A variety of tools exist that automatically recover low-level models of the static structure from the source code [3, 11]. The term low-level refers to models up to class level abstraction. If the analyzed software system makes use of further mechanisms for structuring code units (e.g. C++/Java: namespace/package

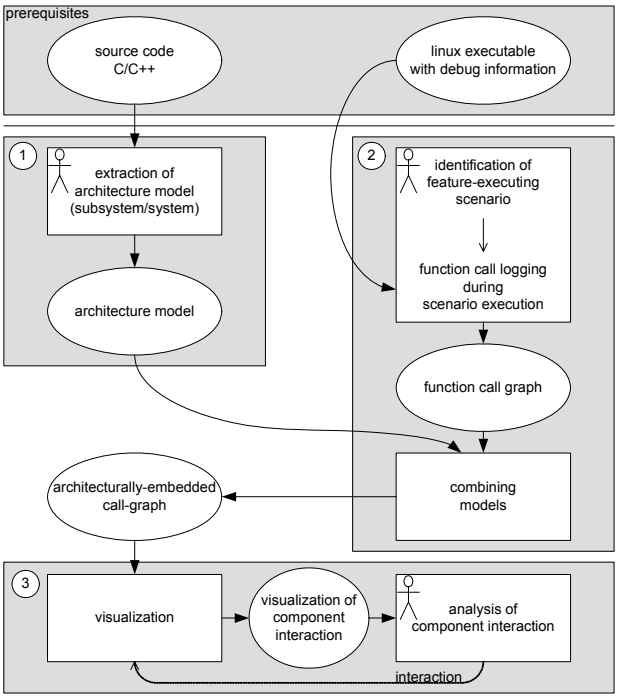

Figure 1: Overview of the semi-automated analysis process of feature analysis. The user-symbol indicates activities with user interaction.

keyword), an automated extraction of these high-level structures can be carried out automatically. The use of naming conventions as well as organizing related source files into one directory are other ways of combining classes to composite code units. The latter method, source file structuring, is commonly used in large software systems.

Our approach maps the directory structure of source files to a hierarchical model of *subsystem components*. Typically the user has additional architectural knowledge of the system and can refine the model. After this, the user integrates the subsystem components into a model that describes the highest-level components, i.e. the system itself and shared libraries of the OS. These *system components* correspond to ELF objects on Linux-OS (Executable and Linking Format).

In a similar way as proposed by Murphy et al. [22], the user finally defines how to integrate the function information from the dynamic analysis into the model by pattern matching with regular expressions (see section 3.2). So the model of system and subsystem components is brought to completion by *function and class components* later. Figure 2 shows the metamodel. All the models consist of a directed, acyclic graph of layered components. The root node, representing the runtime *environment*, is backed up

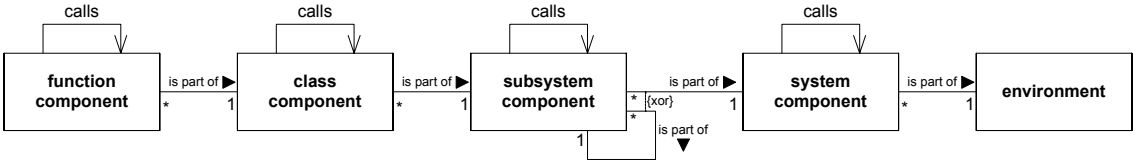

Figure 2: Metamodel. Function call relations from dynamic trace log are aggregated to call relations of higher-level components.

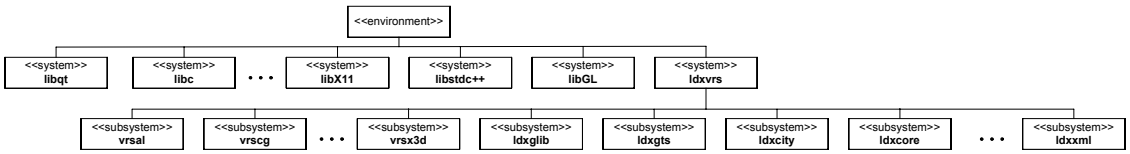

Figure 3: Architecture model of *LandXplorer Studio (LDX)*.

by various layers of system, subsystem, class, and function components.

For reasons of clarity we illustrate the analysis process of our prototype tool by means of a feature analysis of the commercial geographic information system (GIS) tool *LandXplorer Studio (LDX)* from *3D Geo GmbH* [15]. LandXplorer Studio is an interactive, dynamic 3D visualization system for geodata. The C++ source code consists of approximately 700.000 LOC. Figure 3 shows the architecture model of LDX in the OS environment. The feature we analyze is the insertion of a textual label into the geovirtual environment.

3.2 Tracing Scenario Execution

Logging function calls of a software system during runtime permits to recapitulate the internal system behavior once a specific feature has been executed. This dynamic technique overcomes some of the limitations of static source code analysis techniques as it gives a holistic picture of the analyzed system within the runtime environment. Function calls to and from shared libraries can be detected, which is important when analyzing features that are triggered by event driven GUIs. What is more, this technique solves typical comprehension problems during static analysis due to dynamic binding and polymorphism. Additionally, on the basis of dynamic analysis the user has information on how often a function call is executed. This metric allows for assessing qualitatively the coupling strength between functions and, by this, for differentiating between calls used for initializing or for implementing an algorithm core [30].

In contrast to most static approaches, function call logging as a dynamic technique reflects only one specific execution and is therefore incomplete. Another disadvantage of this technique is that it does not detect interaction based on shared data. Moreover, the system behavior is always affected, as the logging mechanism needs to add additional instructions to the system that produce the log entries. This is either done before compile time by source code instrumentation or after building the executable file by binary code instrumentation [24]. A common technique for integrating the logging mechanism in Java applications is to address the Java Platform Debugger Architecture (JPDA) [13]. With this no code instrumentation is necessary. For C/C++ applications running on Linux-OS a similar logging mechanism can be implemented by introducing a layer between the application and the kernel. This layer intercepts the Linux system calls and logs the application behavior without the need for code instrumentation. However, all dynamic analysis techniques slow the analyzed system down. Depending on the amount of logged information, the analyzed system can register a change in behavior. If the user analyzes realtime behavior, the loss in performance is crucial. However, other behavior can also be affected. At worst, the system will no longer react to user interaction properly.

We use Callgrind [33] to log function calls during runtime. Callgrind is a profiling tool based on Valgrind [23], which is a suite of tools for debugging and profiling Linux applications. Valgrind works as layer between the Linux kernel and the application as described above. Hence, no code instrumentation is necessary. Callgrind is a light-weight profiling tool that does not log the full sequence of function calls but summarizes call relations and reconstructs the call-graph from the summaries. The logging mechanism is enabled and disabled interactively by the user, allowing only those function calls to be logged that are executed during feature execution.

Figure 1 (step 2) illustrates the process of extracting the call-graph for a specific feature of the analyzed software system. To begin with, the user identifies a scenario that triggers feature execution. Then the user executes the sequence of user interactions while the logging mechanism is applied. Finally the log file is analyzed automatically and the graph of function calls reconstructed.

Next, the function call graph is embedded within the architecture model of the system. Figure 2 shows the metamodel of the resulting model. Each function is mapped to a *function component*. The full function name permits to reconstruct the function's parent class. Global or C functions are wrapped by dummy classes. Each class is mapped to a *class component*. By pattern matching with regular expressions the class components are assigned to *subsystem components*. The expressions relate to the debug information of the classes' functions. Available debug information per function is the function name, the file where it is implemented, and the ELF object it belongs to.

Function call relations are aggregated to class call relations and integrated into the model. The aggregating step is done for each layer of the model up to the *system component* level, so that the model finally contains one call-graph for each abstraction level describing the interaction behavior between the components of this level: from function interaction to interaction of the analyzed system with shared libraries of the OS. Depending on the feature executed, such an architecturally-embedded call-graph consists typically of more than 10.000 components and several million call relations when analyzing a +100kLOC software system.

4. VISUALIZATION & EXPLORATION

An important usability aspect for an analysis tool is how the tool communicates the analysis results to the user. Our prototype tool uses a multiview visualization system for presenting the architecturally-embedded call-graph. It consists of 3 views: 2 textual and 1 graphical (Figure 4). The combination of views enables the user to effectively explore both the hierarchical system architecture and the call relations embedded within this hierarchy. The views are synchronized, so that a selection of a specific component in one view updates the other views which show complementary information on the component selected.

4.1 Textual Views

The two textual views are (a) the *hierarchy view* and (b) the *detail view*. The hierarchy view ensures rapid navigating within the system architecture and gives information on architectural aspects such as a list of all subsystems, or a list of all classes of a specific

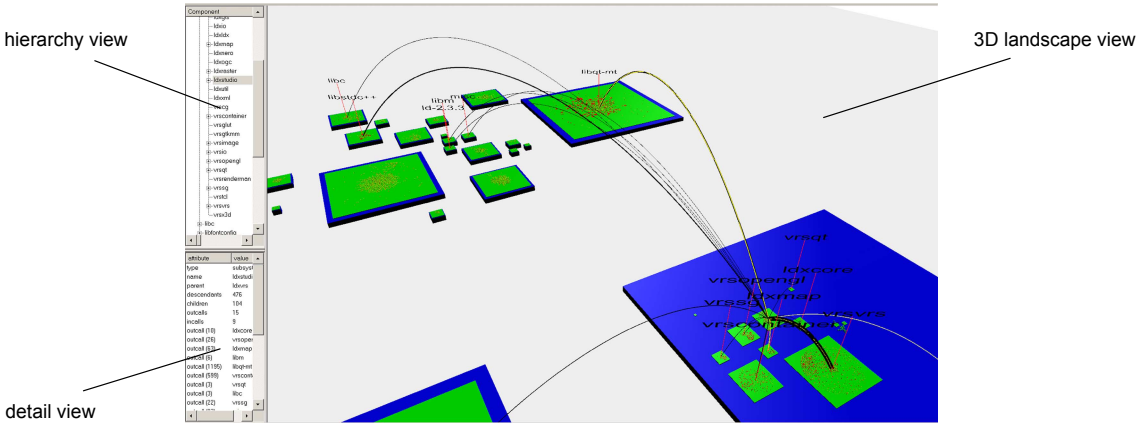

Figure 4: The multiview visualization system consists of a hierarchy tree view, a detail view, and a 3D landscape view.

subsystem. The detail view provides extensive information on a single component, e.g. the number of descendants (all components in the subtree), the number of direct subcomponents, a list of interacting components, or call relations with a specific component.

The user explores the call-graph by selecting out- or in-calls (see section 4.2) in the detail view. The result of view synchronization means that the focus of all views moves on to the next component. In this way, the user can follow a certain execution path in the call-graph.

4.2 3D Landscape View

The 3D landscape view portrays graphically both the hierarchical structure of the components and the call relations between them and enables the user to explore them using elaborate navigation techniques. It provides overview-like information that helps to decide where to step next in the call-graph without the need for an exhaustive analysis of textual views. The 3D landscape view is synchronized with the textual views. Hence, selecting a component in the 3D landscape view updates the other views, which provide detailed textual information on the component selected. The 3D landscape view gives visual cues in respect of various aspects of the architecturally-embedded call-graph:

Hierarchical Structure: Components are represented by flat, 3-dimensional, nested box shapes that are placed on a plane. Subcomponents are represented by miniaturized box shapes that are stacked on the parent component representation. All components of any specific hierarchy level are illustrated in the same color so that the user is able to appreciate the hierarchical structure and easily identify the hierarchy level of any one specific component. A discussion of such landscape-like visualizations is given in section 4.2.2.

Component Activity: The size of a box shape depends on the component's number of descendants. In this way, the user is able to easily identify the components that consist of many active functions during feature execution.

Component Coupling: Component interaction means that a function, which is part of one component, calls a function of another component, i.e. the control flow goes beyond one component to another (transgression). A high number of incidents of transgression means that two components are collaborating intensively. This coupling strength is mapped in the physical distance of the component representations. So the user can intuitively assess component collaboration. A detailed description of the layout algorithm is given in section 4.2.1.

Interaction Partners: The textual detail view provides a list of partner components that are connected to a specific component by an *out-call* (control flow transgression from the specific component to another) or by an *in-call* (vice versa). In the 3D landscape view all out- and in-calls are visualized as being component connecting arcs. Because the components are depicted as nested box shapes, the user can easily determine whether a call relation is taken place within the same parent component. The line strength of the arcs relate to the number of calls between two components.

Starting Point for Analysis: In-call arcs are marked in yellow and out-call arcs in black. This distinction is used at the beginning of the analysis process to find a starting point for the call-graph analysis. Commonly, a feature is triggered by a callback based GUI. The user can visually detect the relevant out-calls from the GUI library by its color.

4.2.1 Component Coupling Layout Algorithm

We use an energy-based layout technique to visualize the amount of coupling strength between components, i.e. the number of instances of control flow transgression. The position of each component is incrementally updated based on the force that the component experiences in its energy field. This force $F \in R^2$ is the negative gradient of the 2-dimensional, scalar energy field $E : R^2 \to R$ [8]: $F = -\nabla E$.

The energy field of each component is composed of three energy terms (see Figure 5):

The **Attraction Energy** describes the component coupling and is proportional to the number of incidents of transgression between two components.
The **Repulsion Energy** describes a general repulsion between components. It ensures the optimal usage of layout space and avoids collisions between components.
The **Layout Space Constraint Energy** guarantees that subcomponent positions are restricted to the dimension of the parent component.

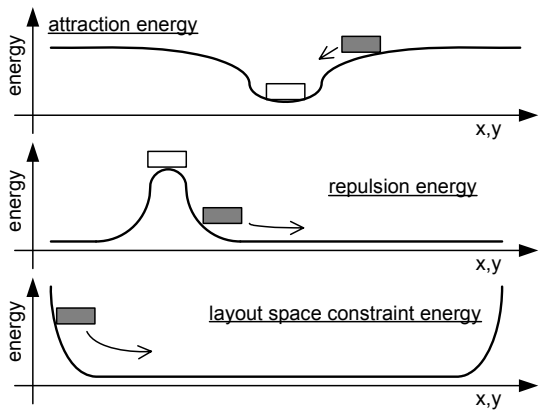

Figure 5: The three energy terms for energy-based component layout calculation.

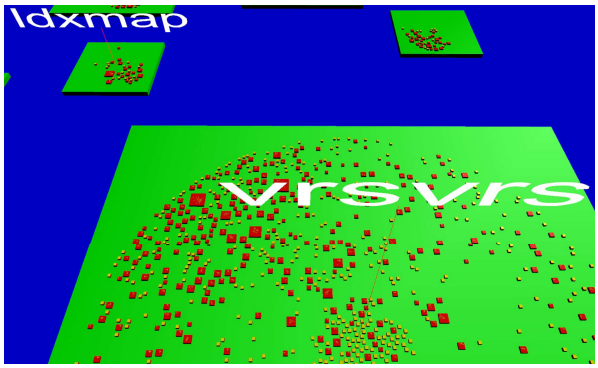

Figure 6: The layout provides visual cues for identifying subcomponent interaction that transgresses the parent component. Some classes of the *ldxmap* subsystem are likely interacting with classes of the *vrsvrs* subsystem.

For each component a layout of all its subcomponents is calculated by minimizing their energy values. The layout reflects clusters of strongly coupled groups of subcomponents. However, these clusters can only reflect interaction between subcomponents of the same parent component, because the nested structure forbids a physically close layout of subcomponents with different parent components.

An extension of the algorithm gives the user a visual cue for identifying subcomponent interaction that transgresses the parent component limits. During the layout calculation of a parent component's subcomponents, all subcomponents of different parent components receive temporary positions on the same parent component as well. They are assigned a fixed position at the parent component's border in the direction of their proper parent component. During the layout calculation, these *virtual subcomponents* may attract real subcomponents. Figure 6 illustrates the resulting effect. Some subcomponents are grouped at the border of the parent component. From a global perspective the user can visually identify them as subcomponents that interact with subcomponents of a certain other parent component.

4.2.2 3D Landscape Visualization

The algorithm we present produces 2-dimensional layouts. By embedding the 2D layouts within a 3D visualization we create a landscape-like 2½D visualization, i.e. a 3D visualization with the majority of information encoded in 2 dimensions; however, the user makes use of all 3 dimensions for exploring the information space. A lot of research has been done on the advantages and drawbacks of 2D and 3D visualizations, respectively [19]. 2½D visualizations seem to combine some of the advantages of both 2D and 3D visualizations, however, avoiding the drawbacks. When drawing complex graphs, two major drawbacks of 2D visualizations are the limited display space and the problem of edge crossings. 3D visualizations can overcome these drawbacks by enlarging the display space by the third dimension. However, the opportunity to build dense visualizations may lead to a high spatial complexity and to obscuring effects. A further drawback of 3D visualizations is that the user becomes easily disoriented while exploring the information space. In other words, the user loses his/her sense of the current position in the information space and fails to recognize regions already explored.

3D landscape visualizations permit the use of spatially simple 2D layouts that can be enhanced by additional geometries, thereby exploiting the third dimension. We use this technique to visualize component connecting arcs. Furthermore, the density of 3D landscape visualizations can continuously be adjusted by the user by tilting the viewing angle onto the landscape. Orientation and navigation tasks are facilitated when compared with fully 3-dimensional visualizations. Thanks to the landscape metaphor, users can rely on their spatial interpretation and navigation skills gained from real world experiences.

5. CONCLUSIONS

When large and complex software systems are concerned, features represent core elements in maintenance processes and, therefore, developers need to identify the code components that implement a specific feature and to understand how the components interact. The dynamic extraction of function call-graphs during feature execution is of great assistance to developers when working on these processes. Typically, a call-graph is explored by stepping from function to function deciding each time whether a function contributes to the feature functionality or not. Our approach facilitates this decision-making by abstracting the function interaction to interaction of higher-level components, i.e. class-class, subsystem-subsystem, and system-shared library. Furthermore, the technique presented provides an intuitive, interactive multi-view visualization that allows developers to explore these high-level call-graphs directly. Thus, analyzing architecturally-embedded call-graphs appear to be an effective approach to gaining a better understanding of essential aspects of complex software systems and to speeding up related reverse engineering tasks.

We plan to perform user studies to evaluate our approach and to identify improvements such as an more elaborate use of color and shape geometry in the visualization, e.g. for encoding additional profiling information like performance data. Furthermore, we plan to consider the ideas proposed in [7, 36] such as analyzing multiple execution traces to identify those code components that dedicate mainly to only one specific feature. Additionally, our future research aims at integrating the prototype tool into an IDE to provide a seamless transition to source code representation.

REFERENCES

[1] Agrawal, H., Horgan, J. R. Dynamic program slicing. In *Proc. of the ACM Conf. on Programming Language Design and Implementation*, 1990, 246-256.

[2] Balzer, M., Noack, A., Deussen, O., Lewerentz, C. Software Landscapes: Visualizing the Structure of Large Software Systems. In *Proc. of the IEEE TCVG Symposium on Visualization*, 2004, 261-266.

[3] Borland Together, *www.borland.com/together*.

[4] Chen, K., Rajich, V. RIPPLES: tool for change in legacy software. In *Proc. of the IEEE Int'l Conf. on Software Maintenance*, 2001, 230-239.

[5] Eick, S. C., Steffen, J. L., Sumner Jr., E. E. Seesoft - A Tool for Visualizing Line Oriented Software Statistics. In *IEEE Trans. on Software Engineering*, 18, 11, 1992, 957-968.

[6] Demeyer, S., Ducasse, S., Lanza, M. A Hybrid Reverse Engineering Approach Combining Metrics and Program Visualization. In *IEEE Working Conf. on Reverse Engineering*, 1999, 175-186.

[7] Eisenbarth, T., Koschke, R., Simon, D. Locating Features in Source Code. In *IEEE Trans. on Software Engineering*, 29, 3, 2003, 210-224.

[8] Feynman, R. P., Leighton, R. B., Sands, S. *The Feynman Lectures on Physics*. Addison-Wesley, 2005.

[9] Gestwicki, P., Jayaraman, B. Methodology and architecture of JIVE. In *Proc. of the ACM Symposium on Software visualization*, 2005, 95-104.

[10] Horwitz, S., Reps, T., Binkley, D. Interprocedural slicing using dependence graphs. In *Proc. of the ACM Conf. on Programming Language Design and Implementation*, 1988, 35-46.

[11] IBM Rational Software, *www.ibm.com/software/rational*.

[12] Irwin, W., Churcher, N. Object oriented metrics: Precision tools and configurable visualisations. In *Proc. of the IEEE Symposium on Software Metrics*, 2003, 112-123.

[13] Java Platform Debugger Architecture, *java.sun.com*.

[14] Jones, J. A., Harrold, M. J., Stasko, J. Visualization of Test Information to Assist Fault Localization. In *Proc. of the IEEE Int'l Conf. on Software Engineering*, 2002, 467-477.

[15] LandXplorer Studio, 3D Geo GmbH, *www.3dgeo.de*.

[16] Lewerentz, C., Simon, F. Metrics-Based 3D Visualization of Large Object-Oriented Programs. In *Proc. of the IEEE Int'l Workshop on Visualizing Software for Understanding and Analysis*, 2002, 70-80.

[17] Lukoit, K., Wilde, N., Stowell, S., Hennessey, T. TraceGraph: Immediate Visual Location of Software Features. In *Proc. of the IEEE Int'l Conf. on Software Maintenance*, 2000, 33-39.

[18] Malloy, B. A., Power, J. F. Exploiting UML dynamic object modeling for the visualization of C++ programs. In *Proc. of the ACM Symposium on Software Visualization*, 2005, 105-114.

[19] Marcus, A., Feng, L., Maletic, J. I. Comprehension of Software Analysis Data Using 3D Visualization. In *Proc. of the IEEE Int'l Workshop on Program Comprehension*, 2003, 105-114.

[20] Michaud, J., Storey, M. A., Müller, H. Integrating Information Sources for Visualizing Java Programs. In *Proc. of the IEEE Int'l Conf. on Software Maintenance*, 2001, 250-259.

[21] Müller, H. A., Tilley, S. R., Wong, K. Understanding software systems using reverse engineering technology perspectives from the Rigi project. In *Proc. of the Centre for Advanced Studies on Collaborative research: software engineering*, 1, 1993, 217-226.

[22] Murphy, G. C., Notkin, D., Sullivan, K. Software Reflexion Models: Bridging the Gap between Source and High-Level Models. In *Proc. of the ACM Symposium on the Foundations of Software* Engineering, 1995, 18-28.

[23] Nethercote, N., Seward, J. Valgrind: A Program Supervision Framework. In *Electronic Notes in Theoretical Computer Science*, 89, 2, Elsevier Science Publishers, 2003.

[24] Nethercote, N. *Dynamic Binary Analysis and Instrumentation*. Ph.D. Thesis, University of Cambridge, 2004.

[25] Orso, A., Jones, J., Harrold, M. J. Visualization of Program-Execution Data for Deployed Software. In *Proc. of the ACM Symposium on Software Visualization*, 2003, 67-76.

[26] Ottenstein, K. J., Ottenstein, L. M. The program dependence graph in a software development environment. In *ACM SIGPLAN Notices*, 19, 5, 1984, 177-184.

[27] Parker, G., Franck, G., Ware, C. Visualization of Large Nested Graphs in 3D: Navigation and Interaction. In *Journal of Visual Languages and Computing*, 9, 3, 1998, 299-317.

[28] Reiss, S. P. Bee/Hive: A Software Visualization Back End. In *Proc. of ICSE Workshop on Software Visualization*, 2001, 44-48.

[29] Souder, T., Mancoridis, S., Salah, M. Form: A Framework for Creating Views of Program Executions. In *Proc. of IEEE Int'l Conf. on Software Maintenance*, 2001, 612-621.

[30] Stroulia, E, Systä, T. Dynamic analysis for reverse engineering and program understanding. In *ACM SIGAPP Applied Computing Review*, ACM Press, 2002, 8-17.

[31] Unified Modeling Language, *www.uml.org*.

[32] Waters, R. G., Chikofsky, E. Reverse engineering: progress along many dimensions. In *Communications of the ACM*, 37, 5, 1994, 22-25.

[33] Weidendorfer, J., Kowarschik, M., Trinitis, C. A Tool Suite for Simulation Based Analysis of Memory Access Behavior. In *Int'l Conf. on Computational Science ICCS*, 2004.

[34] Weiser, M. Program Slicing. In *Proc. of the IEEE Int'l Conf. on Software Engineering*, 1981.

[35] Wilde, N., Gomez, J., Gust, T., Strasburg, D. Locating user functionality in old code. In *Int'l Conf. on Software Maintenance*, 1992, 200-205.

[36] Wilde, N., Scully, M. C. Software reconnaissance: mapping program features to code. In *Journal of Software Maintenance: Research and Practice*, 7, 1, 1995, 49-62.

[37] Zhou, C., Summers, K. L., Caudell, T. P. Graph visualization for the analysis of the structure and dynamics of extreme-scale supercomputers. In *Proc. of the ACM Symposium on Software visualization*, 2003, 143-149.