Visualization of Execution Traces and its Application to Software Maintenance

Dissertation
zur Erlangung des akademischen Grades
„doctor rerum naturalium"
(Dr. rer. nat.)
in der Wissenschaftsdisziplin Praktische Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

von
Dipl.-Phys. Johannes Bohnet
geboren am 19.5.1976 in München

Potsdam, den 13.10.2010

Danksagung

Ich möchte mich bei allen Personen bedanken, die mich während meiner Promotionszeit unterstützt haben. So danke ich besonders Prof. Dr. Jürgen Döllner. Durch ihn wurde es mir möglich, auf dem Gebiet der Softwarevisualisierung zu forschen und die vorliegende Arbeit zu schreiben. Ich konnte mich dabei immer auf seine Unterstützung verlassen. Zudem habe ich durch ihn vieles aus den Bereichen wissenschaftliches Schreiben und Didaktik gelernt.

Bedanken möchte ich mich weiterhin bei Prof. Dr. Guido Wirtz, Prof. Dr. Robert Hirschfeld und Prof. Dr. Konrad Polthier dafür, dass sie sich bereit erkärten, diese Arbeit zu begutachten.

Auch möchte ich meinen Kollegen am Fachgebiet *Computergrafische Systeme* des HPIs danken. Sie sind ein wichtiger Grund dafür, dass mir diese Arbeit so viel Spaß gemacht hat. Namentlich möchte ich besonders Dr. Marc Nienhaus, Dr. Stefan Maaß, Stefan Voigt und Jonas Trümper erwähnen.

Meinen Eltern danke ich sehr dafür, dass sie mich vielfältig gefördert und mich immer bedingungslos unterstützt haben. Ein ganz großer Dank geht zudem an meine beiden Söhne Leonel und Hannes. Durch Euch beide lerne ich täglich neue Seiten des Lebens kennen, die mir –auch für diese Arbeit– viel Energie spenden und gespendet haben.

Kurzfassung

Die Wartung und Weiterentwicklung komplexer Softwaresysteme ist im Allgemeinen kostenintensiv, da die Beteiligten einen signifikanten Teil ihrer Zeit mit dem Verstehen der Struktur und des Verhaltens dieser Systeme verbringen müssen. Programmverstehen ist unter anderem deswegen zeitaufwendig, weil die Struktur komplexer Softwaresysteme und deren inneres Verhalten kaum intuitiv wahrnehmbar und damit nur beschränkt einsehbar sind. Die Visualisierung von Programmabläufen (Traces) stellt einen Ansatz dar, der hilft, die Struktur und das Verhalten eines komplexen Softwaresystems zu verstehen. Die Programmablaufvisualisierung protokolliert während der Systemausführung Sequenzen von Funktionsaufrufen, analysiert und abstrahiert diese Daten und generiert visuelle Darstellungen, die Einblicke in die Struktur und das Verhalten von Softwaresystemen erlauben.

Das Ziel dieser Arbeit ist es, ein Konzept und ein Werkzeug zur Visualisierung von Programmabläufen zu entwickeln, welche die berechnungs- und kognitionsbedingten Herausforderungen überwinden, die bei Techniken zur Programmablaufvisualisierung wegen der typischerweise sehr großen Menge an protokollierten Programmablaufdaten vorhanden sind. Das in dieser Arbeit entwickelte Konzept enthält die folgenden Teilkonzepte: (1) Ein Konzept zur skalierbaren Generierung von Programmablaufdaten aus C/C++ Softwaresystemen; (2) ein Konzept zur Reduktion von Programmablaufdaten, das automatisch rekursive, zeitliche Grenzlinien in den Daten identifiziert und es damit ermöglicht, Programmablaufdaten in einem "top-down" Ansatz zu explorieren; (3) ein Rahmenwerk für Techniken zur Programmablaufvisualisierung, das Lösungen auf die Frage aufzeigt, wie Kerntechniken zur Darstellung von Programmablaufdaten implementiert werden können, sodass "top-down" und "bottom-up" Verstehensstrategien unterstützt werden und (4) ein Konzept zur Kombination von Programmablaufvisualisierung mit weiteren Werkzeugen und Systemen zum Erzeugen von höherwertigen Sichten auf Implementierungsartefakte eines Softwaresystems.

Ein weiterer Beitrag dieser Arbeit ist die Validierung des Gesamtkonzepts durch eine Implementierung, Performanzmessungen und Fallstudien. Die Implementierung des vorgestellten Konzepts liegt als Rahmenwerk zur Erzeugung von Werkzeugen zur Programmablaufvisualisierung vor. Um Skalierbarkeit zu gewährleisten, wurden Performanzmessungen während der Anwendung des Werkzeugs auf große C/C++ Softwaresysteme durchgeführt. Weiterhin wurde das Werkzeug auf industriell erstellte Softwaresysteme angewendet, um Softwareentwickler bei dem Lösen von Wartungsproblemen zu unterstützen.

Abstract

Maintaining complex software systems tends to be costly because developers spend a significant part of their time with trying to understand the system's structure and behavior. Among many reasons, program understanding is time consuming because the system's structure and its internal behavior are not intuitively realizable and can only be partially inspected. The visualization of execution traces represents an approach to help developers to understand complex systems. Practically, execution trace visualization captures the sequence of function calls over time during system execution, analyzes and abstracts that data, and derives visual representations that permit developers to analyze the system's structure and behavior.

The goal of this thesis is to develop a trace visualization concept and tool that can handle the computational and cognitive scalability issues that trace visualization encounters due to the large amount of data that is typically produced when logging runtime processes. The thesis' concept includes the following building blocks: (1) A concept for generating traces of C/C++ software systems in a scalable way; (2) a concept for trace reduction that automatically identifies recursive boundaries within the trace and, by this, supports developers in exploring a trace using a top-down approach; (3) a framework for trace visualization techniques that provides solutions to the question as to how core techniques for viewing trace data can be implemented such that developers are supported in performing top-down and bottom-up comprehension strategies; and (4) a concept for combining trace visualization with 3rd party tools and systems for reverse engineering.

As a further contribution, this thesis validates the proposed concept by means of an implementation, performance measurements, and case studies. The implementation of the concept is provided as a framework for creating trace visualization tools. To ensure scalability of the concept, performance measurements were taken while applying the tool to large C/C++ software systems. Furthermore, the concept and tool has been experimentally applied to industrially developed software systems to solve particular maintenance problems in real-world scenarios.

Contents

CHAPTER 1

Introduction

1.1 Problem Statement

A large fraction of the costs in a software system's life cycle is spent on its maintenance [40]. Erlikh [57] reports an estimate of 85-90%. One important reason why software maintenance tends to be costly is that long-living software systems such as legacy software systems can only be understood in part, and an up-to-date documentation describing the system's structure and behavior is rarely available. Hence, developers, including any person participating in software development and maintenance that operates on source code, devote up to 50% of their time to trying to understand the system's implementation [40]. Likewise, more recent studies emphasize the importance of program understanding during maintenance [5, 106]. One of the many reasons why program understanding is time consuming is that the system's internal behavior can only be inspected in parts. Developers need to re-establish the links between the external, visible behavior and the system's implementation. State-of-the-art developer tools support developers by "showing" the inner processes either by providing information on the system's state at a single point in time (e.g., symbolic debuggers) or by providing time aggregated overviews (e.g., profilers). While these tools help developers to acquire an understanding of the system's execution, reconstructing the execution history needs to be done mentally—a cognitively demanding task.

The visualization of execution traces, i.e., sequences of function calls, represents an approach to help developers to understand a complex system's structure and behavior [41, 45, 179]. Trace visualization reveals the participating functions, their relationships, and their call order while the system runs and exhibits a specific externally visible behavior. In practice, execution trace visualization captures the sequence of function calls over time, analyzes and abstracts that data, and derives visual representations that permit developers to analyze the system's structure and behavior. Throughout the thesis, the term *trace* is used synonymously with the term *execution trace.*

There are a number of commercial and academic program program comprehension tools available that have been adopted by developers for daily use in an industrial software maintenance setting. However, these do not focus on trace visualization.

The main reason for this is that building trace visualization tools encounters major scalability [6, 232] issues: First, it is computationally difficult to process the large amount of data that is typically produced when logging system behavior. Second, it is difficult to explore the vast amount of runtime data—a cognitive scalability issue. For example, capturing the behavior of the *Google Chrome* web browser for 5 seconds while it is downloading and displaying a web page involves more than 10 million function calls.

The aim of this thesis is to develop a concept and a tool that tackles the scalability challenges, pushing the technique of trace visualization one step forward in an effort to get it more widely accepted by developers as a technique for understanding the structure and behavior of software systems during software maintenance.

The research questions of this thesis can be summarized as follows: *How can information on control flows of complex software systems be captured at runtime, analyzed and abstracted, and presented? In particular, how can the computational and cognitive scalability problems be reduced that are encountered by trace visualization?*

1.2 The Approach in a Nutshell

This thesis presents a concept and implementation for the visualization of traces that aims to reduce the computational and cognitive scalability problems that trace visualization encounters. The concept comprises the following building blocks (Figure 1.1):

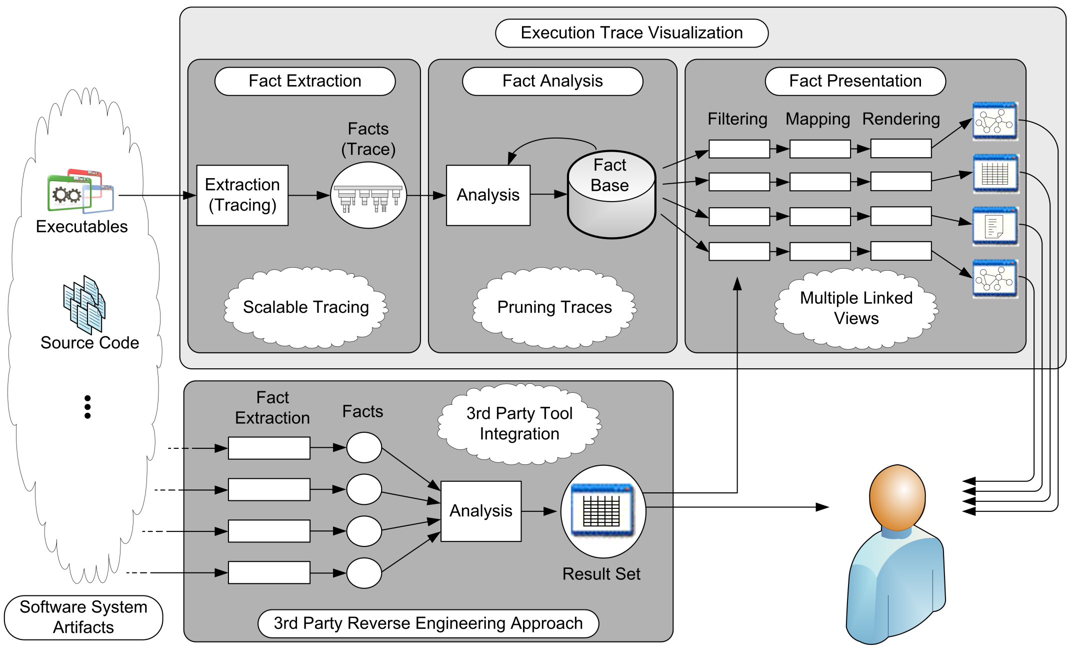


Figure 1.1: This thesis presents a concept that tackles the computational and cognitive scalability problems that the visualization of traces encounters.

1. Scalable technique for recording traces of C/C++ software systems
2. Technique for pruning traces
3. Technique for providing multiple linked views on traces
4. Technique for combining trace visualization with 3rd party tools and systems for reverse engineering

The overall concept—except for parts of the C/C++ tracing technique—is platform-independent and can be applied to software systems written in any procedural programming language. With minor adaptions, even the tracing technique can be applied for other languages than C/C++.

1.2.1 Scalable Technique for Recording Traces of C/C++ Software Systems

A robust and scalable tracing technique for C/C++ software systems is presented in Chapter 4 providing the basis for a scalable trace visualization tool, which can be seamlessly integrated into existing maintenance processes. It is robust in a sense that it does not introduce faulty system behavior such as crashes because it handles all compiler optimizations and other special binary code situations that need to be considered when runtime instrumentation is concerned. Furthermore, it can be easily integrated even into complex build processes because it relies only on common compiler features for code instrumentation, which can be activated by global build options. Moreover, it can be applied during development and maintenance with only a short waiting time (less than a few seconds), as tracing can be quickly activated and deactivated at runtime by standard debugger features. It is possible to apply the tracing technique and, hence, trace visualization to a running software system even in the midst of a debugging session. Finally, the tracing technique offers a means of coping with the scalability issues of trace visualization: The technique detects functions at runtime that are massively called and excludes them from being traced. Thus, the size of the resulting trace is reduced by several orders of magnitude. Additionally, the runtime overhead introduced by the tracing technique is reduced significantly.

1.2.2 Technique for Pruning Traces

Traces typically consist of several hundred thousand function calls even if only a part of the software system's implementation is traced. The *trace pruning algorithm* described in Chapter 5 identifies boundaries within the trace, thus creating a coarse-grained hierarchical trace representation that supports developers in exploring a trace using a top-down approach and in identifying those parts of the trace that are relevant to the maintenance task at hand. The key idea is to take advantage of the fact that the implementation of software systems commonly follows the paradigms of reuse and modularization, which force programmers to implement higher-level system functionality as executions of lower-level functionality. Automatically identifying the calls that "coordinate" executions of lower-level functionality makes it possible to recursively describe a trace as sequences of *phases*, which is a reduced and coarser-grained description of the trace. Based on these trace descriptions, highly compact trace visualizations can be provided.

1.2.3 Technique for Providing Multiple Linked Views on Traces

Chapter 6 explains various techniques to present traces. These trace views are designed to show different characteristics of a trace and to support different program comprehension strategies. For each view, the transformation rules from trace data to the resulting image are given. Additionally, it is demonstrated how the views are linked to support developers in cross-referencing multiple mental models on system behavior that have been built up by analyzing the different views.

1.2.4 Technique for Combining Trace Visualization with 3rd Party Tools and Systems for Reverse Engineering

For many maintenance tasks, analysis tools and systems already exist. They allow developers to identify specific artifacts of the software system (e.g., functions, classes, files) that are relevant to the given maintenance task. Typically, the resulting artifacts contains false positives, i.e., artifacts not relevant to the task at hand. Identifying the true positives tends to be time consuming, especially if one needs to "dig into code" to distinguish between false and true positives. Applying trace visualization in these situations has the following benefits:

- Trace visualization can often be applied as an intermediate step *after* having received the set of artifacts and *before* analyzing the code manually to verify that an artifact is relevant to the given maintenance task. Trace visualization facilitates comprehension of the execution context of the artifacts and helps to eliminate false positives.
- Combining trace visualization with other analysis tools and systems helps to master the scalability issue. The resulting artifacts provide developers with precise entry points for trace exploration. They may perform detailed trace analysis without having to search for maintenance task relevant parts in the trace via a top-down exploration.

Chapter 7 explains how result sets originating from existing analysis tools and systems are incorporated within the trace visualization process. Furthermore, a novel fault localization technique is proposed that supports developers in identifying recently introduced code changes that, contrary to expectations, cause modified system behavior.

1.2.5 Further Contributions

The thesis includes as a further contribution a validation of the proposed concept. For this, the concept was implemented as framework for creating trace visualization tools, performance measurements were taken, and case studies on industrially developed software systems were carried out:

- Chapter 8 introduces `CGA`, a framework for creating trace visualization tools. Performance measurements on `CGA` applied to complex, industrial C/C++ software systems show that the thesis' concept helps to overcome problems pertaining to major scalability issues.

- Chapter 9 presents case studies where `CGA` was applied to real-world maintenance problems in cooperation with industrial partners. The case studies provide evidence that trace visualization enhances the performance of industrial developers during maintenance activities.

CHAPTER 2

The Context of Trace Visualization

Trace visualization can facilitate software maintenance tasks, especially those tasks that require an understanding of the software system's structure and behavior. This chapter discusses fundamental concepts of software maintenance, reverse engineering, program comprehension, and data mining, which form the context of trace visualization. Furthermore, fundamental concepts of information visualization are outlined, in particular the visualization pipeline. The chapter gives definitions of key terminology such as *traces* and *trace visualization.*

2.1 Software Maintenance

Software maintenance has to cope with implementations of software systems that can be considered to be vast, consisting of hundreds of thousands or even millions of lines of code. As the system evolves over time, the implemented concepts tend to drift away from the initially documented ones. Hence, it is likely that no reliable up-to-date documentation is available explaining the system's structure and behavior on a higher-level of abstraction, i.e., conceptually higher than the implementation itself.

One reason why software systems are complex is that their implementation is highly likely to contain design anomalies in the long-run. Commonly, the software system is developed and maintained over a considerable period of time by a constantly changing developer team. Except for the initial development phase, when developers change code, they do not have the original developers' concepts in mind, causing the system's implementation to degrade. As Parnas [166] describes: *"Changes made by people who do not understand the original design concept almost always cause the structure of the program to degrade. Under those circumstances, changes will be inconsistent with the original concept; in fact they will invalidate the original concept. [...] After those changes, one must know both the original design rules, and the newly introduced exceptions to the rules, to understand the product. After many such changes, the original designers no longer understand the product. Those who made the changes, never did. [...] Software that has been repeatedly modified (maintained) in this way becomes very expensive to update."*

Software Maintenance Tasks Tasks that are performed on pre-existing software systems are called *maintenance tasks*. The IEEE standard for software maintenance [130] defines the term as follows.

> **Definition 1 (Software Maintenance Task)** Any change made to a software system after it is operational is a software maintenance task (short: maintenance task).

This definition includes tasks performed to eliminate bugs (corrective maintenance), tasks to adapt the system to changes in the environment (adaptive maintenance), and tasks to improve performance or maintainability (perfective maintenance). Efforts to add new functionality that arise from changed requirements fall within the category of adaptive maintenance. A large fraction of the time that developers expend on performing maintenance tasks is dedicated to program comprehension. Corbi [40] reports that this fraction exceeds 50%.

Developer Throughout the thesis, the term *developer* is a placeholder for a variety of terms.

> **Definition 2 (Developer)** The term developer stands for all participants in software development processes that operate on the system's source code and further artifacts. Hence, the term incorporates roles such as *programmer*, *implementer*, *maintainer* and, to some degree, *tester*.

Artifact The term artifact is used in this work according to the definition given by Kruchten in the context of the *Rational Unified Process* [114].

> **Definition 3 (Artifact)** An artifact is a piece of information that is produced, modified, or used by a (software development) process. Artifacts are the tangible products of the project: the things the project produces or uses while working toward the final product. [...] Artifacts may take various shapes or forms: a model, such as the use-case model or the design model; a model element, such as a class, a use case, or a subsystem; a document, such as a business case or software architecture document; source code; executables.

2.1.1 Program Comprehension

Research into program comprehension is concerned with the cognitive processes that developers adopt when trying to understand how a software system is structured and how it behaves [200]. A key question in program comprehension theory is how mental models are constructed [135]. In cognitive psychology mental models are used to describe a given situation in the world and to generate conclusions from it. Johnson-Laird et al. [97] state that "*each mental model represents a possibility, and its structure and content capture what is common to the different ways in which* *the possibility might occur*". According to Eysenck and Keane [59] "*a mental model represents a possible state-of-affairs in the world*".

The program comprehension community has developed various models to describe the cognitive processes and information structures used to form a mental model. While all these cognitive models differ to some degree, they share many concepts so that key activities in program comprehension can be identified [135]. Developers choose from a set of different comprehension strategies including *top-down* and *bottom-up* comprehension strategies [135, 200]. Furthermore, developers frequently switch between the various strategies.

Top-Down Comprehension Strategy

In the case of the top-down comprehension strategy, the developer reconstructs domain knowledge and tries to map this knowledge to the system implementation [21], the software system's externally visible functionality thereby serving as a starting point. The process begins by constructing a hypothesis about the global nature of the software system. After that, the hypothesis is successively refined in a hierarchical fashion by forming subsidiary hypotheses. Soloway and Ehrlich [195] have observed that this strategy is used when the software system or the type of software system is familiar.

Bottom-Up Comprehension Strategy

Developers following a bottom-up comprehension strategy start by analyzing low-level implementation artifacts, e.g., by reading source code. This information is then aggregated and *chunked* into higher-level abstractions. The term *chunk* is defined in cognitive psychology as an *in memory stored unit formed from integrating smaller pieces of information* [59]. Shneiderman and Mayer further distinguish between language-dependent knowledge (syntactic knowledge) and language-independent knowledge (semantic knowledge) that is used for aggregation and chunking to build a final mental model [190]. Pennington points out that developers typically start by building a mental model of the control flow [169]. She states that abstract knowledge of control flow plays the initial role in organizing memory representation and that control flow or procedural relations dominate in the memory representation. Storey et al. [200] remark that "*reading code belonging to a delocalized plan can be cumbersome as it may involve frequent switching between files which will rapidly lead to a feeling of disorientation*". A *delocalized plan* is conceptually related code that is implemented in non-contiguous parts of the implementation. The term refers to the concept of *programming plans.* Expert developers are assumed to have acquired a repertoire of such plans, which represent stereotypic code fragments, allowing them to generate code and recognize its structure more easily [66].

Switching between Comprehension Strategies

Letovsky [119] concluded that developers form a *knowledge base* to encode domain, application, and programming expertise. Depending on the knowledge base contents, developers opportunistically follow either top-down or bottom-up comprehension strategies. A mental model describing the software system's implementation in terms of data structures and algorithms is built up iteratively during *inquiry episodes* in instances where a developer asks a question, conjectures an answer, and then searches through the software system's artifacts to verify or reject the conjecture.

Experiments carried out by Van Mayrhauser and Vans revealed that developers frequently switch between different comprehension strategies [134]. Developers create and maintain a set of mental models—depending on the used strategies, frequently switching between the strategies and cross-reference the models [200].

Program Comprehension as Interactive Task

An outstanding characteristic of program comprehension is that it is a cyclic and interactive process. Klint [105] explains: "*Initial findings on initial questions trigger new questions that lead to new findings, and so on. In addition to this, the results of understanding may be viewed and browsed in various manners.*"

2.1.2 Reverse Engineering

In forward engineering, higher-level concepts are transformed into lower-level artifacts such as source code [114]. Reverse engineering is the opposite process where developers need to recover and understand the higher-level concepts by analyzing the lower-level artifacts. For reverse engineering tasks, tools and systems are available that support developers during program comprehension. These tools should be designed with regard to comprehension strategies. Storey et al. [200, 202] specify a set of important cognitive design elements which a comprehension supporting tool should provide. A subset with regard to trace visualization consists of the following requirements:

- Support for Top-Down Strategies [202]: "*A tool may provide a layered view of the program (previously prepared during system evolution or through reverse engineering) to entice a maintainer to explore the program in a top-down fashion. [...] To explore programs top-down, access to the software architecture should be provided at various levels of abstraction. [...] Programmers can make use of top-down views to gain a high-level understanding of the entire program, and then focus only on the parts that need to be understood in order to complete some task.*"
- Support for Bottom-Up Strategies [202]: "*Bottom-up comprehension involves 3 main activities: 1) identifying software objects and the relations between them; 2) browsing code in delocalized plans; and 3) building abstractions from lower-level units. [...] In order to support bottom-up understanding, a software visualization tool needs to provide immediate access to the atomic units in the program. [...] Reading code belonging to a delocalized plan can be cumbersome as it may involve frequent switching between files resulting in a feeling of disorientation. Multiple views can be used to reduce the effects of delocalized plans, as multiple views of source code, as well as other views such as call*

graphs. [...] If instances of the same object are similarly highlighted in several views, the negative effects of delocalized plans can be reduced."

- Support for Switching between Strategies [202]: "*Tools need to be designed to allow frequent switching between top-down and bottom-up comprehension strategies. [...] Programmers create various mental models and frequently switch between them during the course of comprehension. A tool [...] should support the construction of several linked views representing a variety of cross-referenced mental models."*

Chikofsky and Cross [33] define reverse engineering as the "*process of analyzing a subject system to (a) identify the system's components and their interrelationships and (b) create representations of the system in another form or at a higher-level of abstraction*". A higher-level of abstraction thereby means that these representations of the software system are less implementation-dependent and more dependent on the application domain.

The first step in a reverse engineering process is to extract facts. The term *fact* refers to data that is extracted from artifacts.

Definition 4 (Fact) A fact is data about a software system obtained by parsing, querying, and analyzing the system's artifacts. Additionally, a fact may be derived from other facts.

Reverse engineering techniques are based on processing facts about the software system obtained from various artifacts [47] (Figure 2.1):

- Source code
- Executables
- Configuration management systems
- Documentation
- Test cases
- Interviews with users and developers

The steps of the reverse engineering process are:

1. **Fact Extraction**: Facts are extracted from the source code, running executables, configuration management systems, documentation, test cases, or from interviews. In the reverse engineering community the fact extraction step is also referred to as *extraction step* [48, 103, 105, 115, 144, 171] or *data gathering* [212].

2. **Fact Analysis**: Facts originating from different sources are integrated into a fact base. Additionally, some form of abstraction is derived from the facts

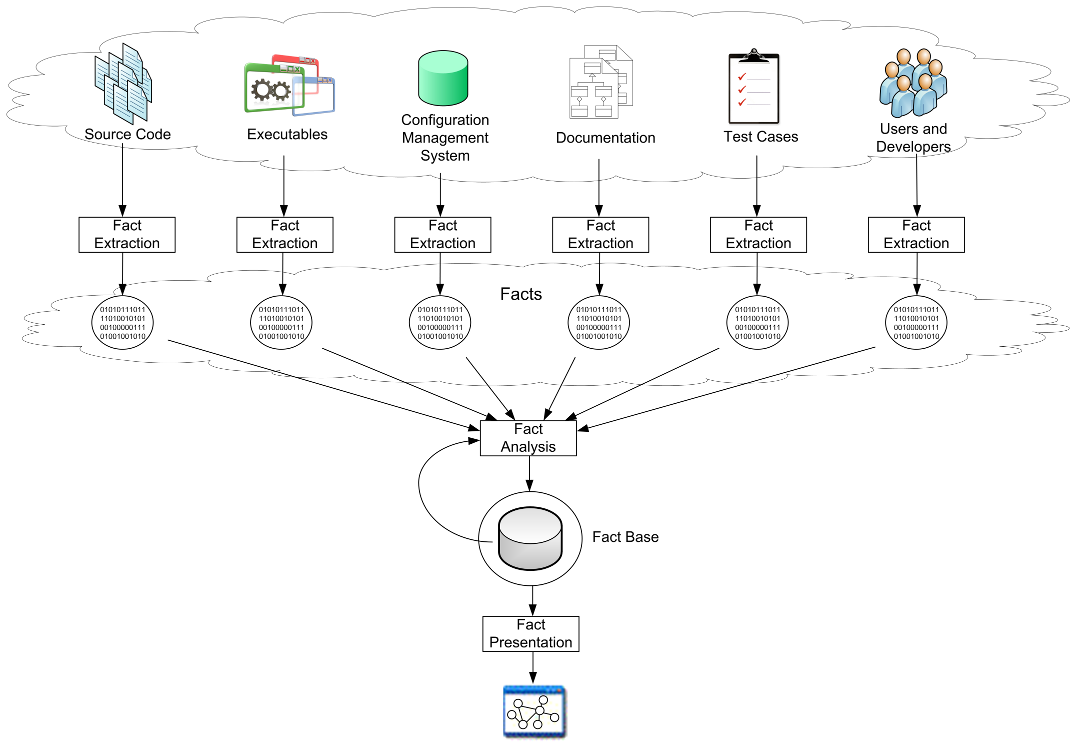


Figure 2.1: Illustration of important aspects of the reverse engineering process.

resulting in additional facts. One example involving object-oriented software systems is lifting up a method call graph to call relations between packages. Synonyms for the fact analysis step commonly used in the reverse engineering community are *analysis* [49, 103], *knowledge inference* [212], and *abstraction* [105, 115, 144, 171].

3. **Fact Presentation**: Developers explore the fact base to comprehend aspects of the software system that are related to a particular maintenance task. Exploration means querying the fact base and analyzing the query results in a highly interactive way. In each query, some facts are selected from the fact base and presented to the developer, giving a specific view of the fact base. In a typical situation, developers should be provided with multiple views simultaneously. Synonymously used terms for the fact presentation step are *visualization* [49, 103], *knowledge presentation* [212], *presentation* [144], and *view* [48, 105, 115, 171].

2.1.3 Dynamic versus Static Analysis Techniques

Reverse engineering techniques are classified as *dynamic*, *static*, or combined analysis techniques. *Static analysis* techniques examine source code and build an abstraction of the runtime state, which permits developers to reason over all possible behaviors [58]. Diehl [49] states that "*static analysis computes properties of a program which* *hold for all executions of the program.*" These techniques are typically sound, i.e., they guarantee—due to their conservatism—specific properties to be true. *Dynamic analysis* techniques, on the other hand, execute the software system over certain inputs and observe the execution [58]. For this, these techniques require an instrumentation infrastructure. Additionally, one must decide what to measure and which scenario to run. Hence, dynamic analysis techniques are precise but unsound by nature—their results may not allow generalization as to future executions.

For program comprehension, applying static analysis alone is not sufficient. Cornelissen et al. [42] emphasize the need for dynamic analysis: "*Among the benefits over static analysis are the availability of runtime information and, in the context of object-oriented software, the exposure of object identities and the actual resolution of late binding.*" The same holds for function pointers in procedural languages.

Heuzeroth and Löwe [81] explain likewise that "*static program information captures the program structure, but even elaborated static analysis techniques obtain only little information on the runtime behavior of the program in advance. Hence, we additionally need dynamic information to understand the behavior of the system in example runs for specific use cases. We also need to assess dynamic information to gain structural information, e.g., the target of polymorphic calls or the class(es) of objects contained in heterogeneous containers. Static analysis cannot provide this information in the general case.*"

Gamma et al. [63] illustrate the problem as follows: "*An object-oriented program's run-time structure often bears little resemblance to its code structure. The code structure is frozen at compile-time; it consists of classes in fixed inheritance relationships. A program's run-time structure consist of rapidly changing networks of communicating objects. In fact, the two structures are largely independent. Trying to understand one from the other is like trying to understand the dynamism of living ecosystems from the static taxonomy of plants and animals, and vice versa.*"

2.2 Data Mining

The reverse engineering process represents a form of *data mining*. According to Han and Kamber [75] data mining is defined as follows:

> **Definition 5 (Data Mining [75])** Data mining is the process of discovering interesting knowledge from large amounts of data stored in databases, data warehouses, or other information repositories.

Witten and Frank [230] give a similar definition.

> **Definition 6 (Data Mining [230])** Data mining is defined as the process of discovering patterns in data. The process must be automatic or (more usually) semiautomatic. The patterns discovered must be meaningful in that they lead to some advantage, usually an economic advantage.

Maimon and Rokach [128] define data mining in a more restrictive way describing it as *the* essential step in the process of *knowledge discovery in databases*.

Definition 7 (Knowledge Discovery in Databases) Knowledge discovery in databases (KDD) is an automatic, exploratory analysis and modeling of large data repositories. KDD is the organized process of identifying valid, novel, useful, and understandable patterns from large and complex data sets.

Definition 8 (Data Mining [128]) Data mining is the core of the KDD process, involving the inferring of algorithms that explore the data, develop the model and discover previously unknown patterns. The model is used for understanding phenomena from the data, analysis and prediction.

Figure 2.2 illustrates the process of knowledge discovery in databases [128]. It consists of an iterative sequence of the following steps:

- Data Preparation: This step includes data cleaning, data integration, data selection, and data transformation.
- Data Mining: With this essential step, intelligent methods are applied in order to extract data patterns.
- Pattern Evaluation and Knowledge Presentation: The truly interesting patterns representing knowledge can be identified based on some measures of pattern interestingness. Finally, visualization techniques are used to present the mined knowledge to the user.

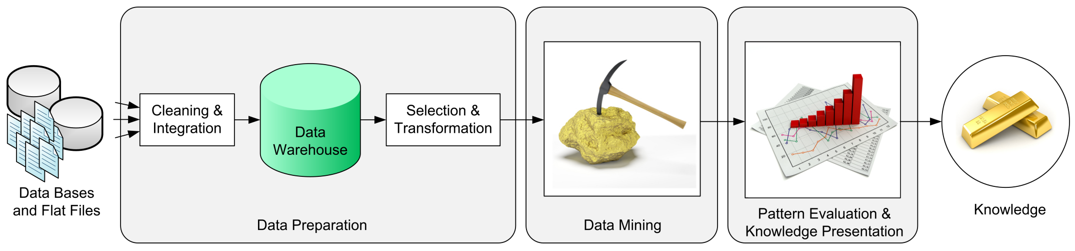

Data Bases and Flat Files
Cleaning & Integration
Data Warehouse
Selection & Transformation
Data Preparation
Data Mining
Pattern Evaluation & Knowledge Presentation
Knowledge

Figure 2.2: Illustration of the process of knowledge discovery in databases.

Different kinds of of patterns can be extracted by data mining techniques [75]. They include:

- *Class descriptions* describe classes of data in summarized, concise, and precise terms. Such decriptions can be derived from *data characterization*, a technique for summarizing the general characteristics of a class of data, or by way of *data discrimination*, a technique for comparing general characteristics of a given class with contrasting classes.

- *Frequent patterns* are frequently occuring patterns in data, such as itemsets, subsequences, and substructures. Mining frequent patterns leads to the discovery of interesting associations between data sets in the form of *association rules.* An association rule may describe certain probabilities, for example, the likelihood that a customer who buys bread also buys milk.
- In a set of class-labeled data objects, a *model* is used to describe and distinguish classes of which the labels are unknown and to predict future data trends. The derived model is based on the analysis of *training data*, i.e., data objects with known class labels. *Classification* uses models to predict categorical (discrete) labels in data. *Prediction* uses models to describe continuous-valued functions. Presentations of models include *classification rules*, *decision trees*, and *neural networks.*
- *Clusters* of data are obtained following the principle of *maximizing the intraclass similarity and minimizing the interclass similarity* in a training data set. As *clustering analysis* works without consulting class labels, it can be used to generate such labels.
- *Outliers* are data objects that do not comply with general characteristics of the objects in the data set.

2.2.1 Mining Software Engineering Data

Applying data mining techniques to software engineering data, i.e., fact bases (cf Section 2.1.2), poses several challenges. Xie et al. [231] state that "*there might be no existing mining algorithms that produce desired pattern representations, and developing new algorithms for such representations can be difficult. Overall, ensuring a scalable yet expressive mining solution is difficult. [...] Further, execution traces collected from an even average-sized program can be very long, and dynamically or statically extracted call graphs can be enormous. Analyzing such large-scale data poses a challenge to existing mining algorithms*".

Furthermore, they emphasize the challenges related to the developers' work habits: "*In modern integrated software engineering environments, especially collaborative environments, software engineers must be able to collect and mine software engineering data on the fly to provide rapid just-in-time feedback. Stream data mining algorithms and tools could be adapted or developed to satisfy such challenging mining requirements*".

2.3 Visualization

The thesis develops and applies concepts and techniques of *information visualization* within the context of reverse engineering and program comprehension. Fundamentals of the field of information visualization are outlined in this section.

Card et al. [29] define *visualization* using the term *cognition*, i.e., the gathering or use of knowledge.

Definition 9 (Visualization) Visualization is the use of computer-based, interactive, visual representations of data to amplify cognition.

Ware [226] emphasizes that "*the power of a visualization comes from the fact that it is possible to have a far more complex concept structure represented externally in a visual display than can be held in visual or verbal working memories. People with cognitive tools are far more effective thinkers than people without cognitive tools and computer-based tools with visual interfaces may be the most powerful and flexible cognitive systems*". Furthermore, Ware [225] notes that "*the human visual system is a pattern seeker of enormous power and subtlety. The eye and the visual cortex of the brain form a massively parallel processor*".

Tufte [216] points out that such graphical representations can convey a complex subject matter with clarity, precision, and efficiency. Butler et al. [27] note that visualization may support users in a variety of distinctive cognitive processes. Driven by these processes, three types of usage contexts can be distinguished for visualization techniques:

- Discovery: This represents an exploratory approach. The user does not know what to look for.
- Decision making: This represents an analytical approach. The user has a hypothesis about the data and tries to verify it.
- Explanation: This represents a descriptive approach. The user knows the phenomenon captured within the data. However, a visual representation of the phenomenon is used as a means of communication.

A further distinction is made in respect of the type of data to be visualized. One distinguishes between *scientific visualization* and *information visualization* [225]. In scientific visualization, the data is concerned with objects and concepts associated with phenomena from the physical world, typically with an inherent spatial component. *Information visualization* focuses on abstract and non-physically based data. Chi [32] tries to make this distinction more clear; however, there always remains an overlap of the two fields.

2.3.1 Information Visualization

A definition of information visualization is given by Card et al. [29].

Definition 10 (Information Visualization) Information visualization is the use of abstract, non-physically based data to amplify cognition.

According to Carr [30], information visualization is particularly useful "*if there are large amounts of data, the user goals are not easily quantifiable, and there are no simple algorithms to accomplish the goals*". Similarly, Spence [196] states that information visualization is demanded in "*situations in which data is available, sometimes in very large quantities, and where some human insight into that data is required*". Heer et al. [77] note that "*information visualization technologies have proven indispensable tools for making sense of complex data*" and that "*visual representations of abstract information have been used to demystify data and reveal otherwise hidden patterns*".

Kosara et al. [109] emphasize that visualization techniques are not efficient or inefficient per se. Efficiency always needs to be evaluated ith regard to a clearly defined task.

2.3.2 Software Visualization

Applying information visualization techniques in the domain of software engineering is referred to as *software visualization.* Diehl [49] states: "*It has often been noted that software is inherently intangible and invisible. The goal of software visualization is not to produce neat computer images, but computer images which evoke mental images for comprehending software better.*" Storey [202] regards software visualization as "*a useful and powerful technique for helping programmers understand large and complex programs*". Likewise, Kranzlmüller [113] emphasizes the need for software visualization when having to cope with challenging development processes: "*A basic necessity in this context are methods and tools to improve program understanding. An accepted and powerful technique to manage the various stages of the software lifecylce—especially during specification, design, programming, and program analysis—is visualization.*" Mili and Steiner [141] state that "*in software engineering, there is ample evidence that a clear and visual representation of a software product can significantly enhance its understanding and reduce the life cycle cost.*"

A broad definition of software visualization is given by Zhang [234]:

> **Definition 11 (Software Visualization [234])** Software visualization refers to the use of various visual means in addition to text in software development. The various forms of development means include graphics, sound, color, gesture, animation, etc. Software development life cycle involves the activities of project management, requirement analysis and specification, architectural and system design, algorithm design, coding, testing, quality assurance, maintenance, and, if necessary, performance tuning.

Price et al. [173] define the term in a similar way.

> **Definition 12 (Software Visualization [173])** Software visualization is the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both human understanding and effective use of computer software.

Mili and Steiner [141] propose the following definition:

Definition 13 (Software Visualization [141]) Software visualization is a representation of computer programs, associated documentation and data, that enhances, simplifies and clarifies the mental representation the software engineer has of the operation of a computer system. A mental representation corresponds to any artifact produced by the software engineer that organizes his or her concept of the operation of a computer system.

A more restrictive definition of software visualization is given by Diehl [49] who focuses on visualization for software reverse engineering and maintenance tasks:

Definition 14 (Software Visualization [49]) Software visualization is the visualization of artifacts related to software and its development process. In addition to program code, these artifacts include requirements and design documentation, changes to the source code, and bug reports, for example. Researchers in software visualization are concerned with visualizing the structure, behavior, and evolution of software.

Diehl [49] adds that "*in contrast to visual programming and diagramming for software design, software visualization is not so much concerned with the construction, but with the analysis of programs and their development process*". Zhang [234] emphasizes that "*the act of software analysis can be a most challenging task, which is determined by the complexity of the software itself and the actual scale of the program and its data structures during execution. In this context, software visualization tools have provided valuable assistance by enclosing the program's complexity within graphical displays to simplify the analysis task.*"

2.3.3 The Visualization Pipeline

The visualization process is modeled conceptually as a sequence of successive steps that derive visual representations from raw data [186]; this concept represented by the *visualization pipeline* is illustrated in Figure 2.3. The visualization process is conceptually divided into three steps:

1. **Filtering**: The filtering step operates on raw data. Operations on the data include interpolating missing parts of the data, computing data characteristics

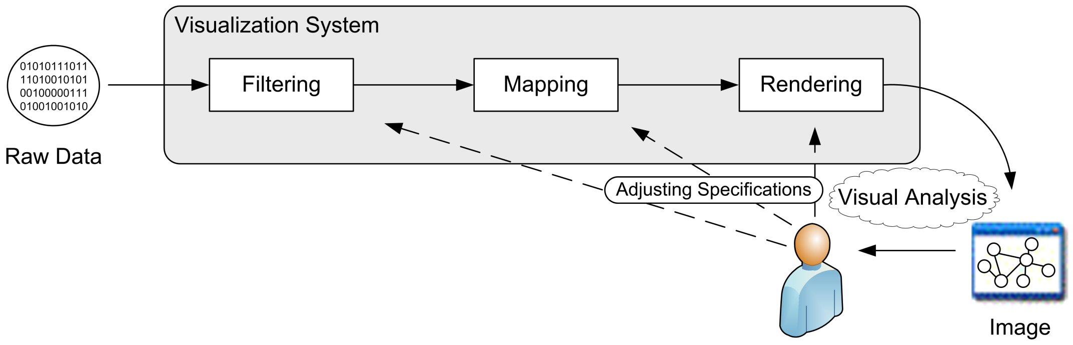


Figure 2.3: The visualization process modeled as pipeline.

such as extreme values or gradients, cleaning data from noise, and selecting a subset of the data.

2. **Mapping**: The mapping step transforms the filtered data into a geometry model. That is, data values are mapped onto geometric primitives, the primitives' attributes (e.g., color), and their layout (i.e., relative positions). For instance, scene graph representations may be used.
3. **Rendering**: The rendering step transforms mapped data into visual representations. For 3D geometry models a *virtual camera* defines the 3D view that can be represented by a projective transformation on the view frustum.

For interactive visualization, the user triggers the visualization process in a cyclic way: The image that is created by the rendering step is shown, and the user can gather insights into the underlying data. Typically, the user can adjust the specifications that control the filtering, mapping, and rendering step.

2.4 Trace Visualization

Traces present the core subjects of this work. In this section, we first give definitions for *traces*. Next, a model for the trace visualization process is elaborated on. Finally, it is discussed how information on the structural decomposition of the system is combined with trace data.

2.4.1 Definitions of Traces

A broad definition of the term *trace* is given by Clements et al. [35]: "*Traces are sequences of activities or interactions that describe the system's response to a specific stimulus when the system is in a specific state. These sequences document the trace of activities through a system described in terms of its structural elements and their interactions*".

In this thesis, we follow a more restrictive definition that is concerned with control flow on function[1] granularity. Traces result both from recording the execution of single-threaded and multi-threaded software systems. In the case of multi-threaded systems, however, multi traces are created—one trace per thread.

Definition 15 (Trace) A trace is a sequence of events that stand for a software system's control flow entering or leaving a function. As entry and exit events occur in pairs forming a function call, a trace can be interpreted as sequence of nested function calls.

1 In different programming paradigms different terms for *function* are used. Throughout the thesis, the term *function* stands as placeholder for all of these terms, e.g., procedure, subroutine, method, etc.

Function level abstraction plays an important role in reverse engineering and program comprehension: Functions are the smallest *named* and semantics conveying execution units. Control flow captured on function level abstraction helps developers to understand delocalized plans (cf Section 2.1.1). Kazman et al. point out that function-based representations of a system are essential for understanding a system on an architectural level of abstraction [101]. They represent the system in a more coarse-grained way than abstract syntax trees (AST), control flow graphs (CFG), or data flow graphs (DFG).

Specific reverse engineering techniques and tasks may require a representation of system behavior which goes beyond function calls. Examples include variable states, object identifiers in object-oriented software systems, or assembler instructions. In this thesis, the proposed concepts are based on the function granularity only.

Mathematically, traces can be defined as graphs. Let $\mathbb{T}$ be the set of all possible traces, i.e., sequences of function calls that conform to the constraints defined in the following. Each trace $T \in \mathbb{T}$ is defined as a graph $T = (F,C)$ where $F \subset \mathbb{F}$ is a set of nodes representing functions and $C \subset \mathbb{C} = \mathbb{F} \times \mathbb{D} \times \mathbb{F}$ is a set of edges representing calls from a caller function to a callee function (Figure 2.4). $\mathbb{D} = \mathbb{N} \times \mathbb{N} \times \mathbb{L}$ are edge labels that carry the information on the start time $t_s \in \mathbb{N}$, the end time $t_e \in \mathbb{N}$ of a call ($t_s < t_e$), and of the location of the call site $l \in \mathbb{L}$, e.g., the source code line of the statement that triggers the call. Table 2.1 defines convenience operations for accessing subelements of a call $c = (f, (t_s, t_e, l), g) \in C$.

For each trace $T = (F,C) \in \mathbb{T}$, C forms a tree with regard to time containment (Figure 2.4)[1]. The following constraints are given:

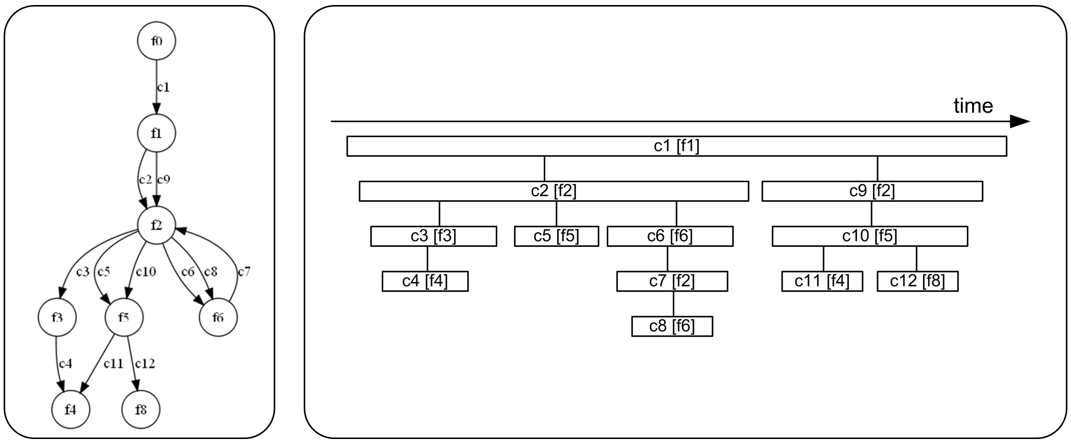


Figure 2.4: A *trace* is a graph structure of functions (nodes) connected via directed edges (calls) [left]. The time information attached to the calls ensures that the calls form a tree with respect to time containment [right]. Parent-child relations in the call tree correspond to caller-callee relations between functions in the graph structure.

1 This constraint implies a relation between sibling nodes in the call tree that arranges siblings according to their sequential execution times. Additionally, parent-child relations in the tree correspond to caller-callee relations between functions in the graph T.

Operation	Domain and Range	Result
$time$	$\mathbb{C} \to \mathbb{N} \times \mathbb{N}$	$c \mapsto (t_s, t_e)$
$start$	$\mathbb{C} \to \mathbb{N}$	$c \mapsto t_s$
end	$\mathbb{C} \to \mathbb{N}$	$c \mapsto t_e$
$callsite$	$\mathbb{C} \to \mathbb{L}$	$c \mapsto l$
$caller$	$\mathbb{C} \to \mathbb{F}$	$c \mapsto f$
$callee$	$\mathbb{C} \to \mathbb{F}$	$c \mapsto g$

Table 2.1: Convenience operations for a given trace $T = (F,C)$ to access subelements of a call $c = (f, (t_s, t_e, l), g) \in C$.

- Time values are unique.
- There is exactly one root call, $root \in C$ that encloses all calls in $C \setminus \{root\}$ with regard to time.
- Any two calls in C are either enclosed in time or do not overlap in time.
- For any $c \in C \setminus \{root\}$ it holds that:
  (1) there are one or more calls that enclose the call c in time;
  (2) for the shortest call c_{parent} of all enclosing calls it holds that $callee(c_{parent}) = caller(c)$.

Table 2.2 defines further convenience operations to facilitate the description of algorithms for traces.

For reasons of clarity, Figure 2.5 depicts the data model of traces using the UML notation.

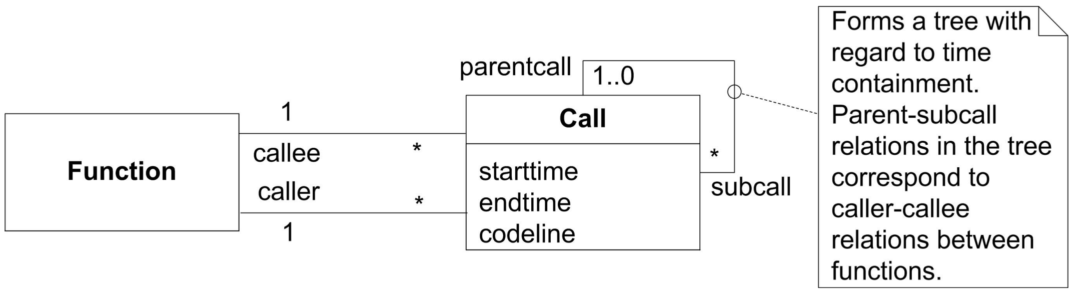

Forms a tree with regard to time containment. Parent-subcall relations in the tree correspond to caller-callee relations between functions.
parentcall
1..0
Call
starttime
endtime
codeline
*
subcall
Function
1
callee
caller
*
*
1

Figure 2.5: Data model of traces using the UML notation.

2.4.2 The Trace Visualization Process

Essentially, the model of the trace visualization process unifies the reverse engineering and the visualization process models; this is illustrated in Figure 2.6. The model consists of the following process steps:

1. **Fact Extraction**: The starting point is an executable of the software system, which is modified in such a way that system execution generates a trace.

Operation	Domain and Range	Result
$parentcall$	$\mathbb{C} \to \mathbb{C}$	Returns the shortest call of all calls that enclose c in time.
$subcalls$	$\mathbb{C} \to 2^{\mathbb{C}}$	Returns the set of calls whose $parentcall$ is c.
$enclosingcalls$	$\mathbb{C} \to 2^{\mathbb{C}}$	Returns the set of calls obtained from recursively collecting $parentcalls$ starting with c.
$triggeredcalls$	$\mathbb{C} \to 2^{\mathbb{C}}$	Returns set of calls obtained from recursively collecting $subcalls$ starting with c.
$costs$	$\mathbb{C} \to \mathbb{N}$	Returns the difference between end and start time of c.
$selfcosts$	$\mathbb{C} \to \mathbb{N}$	Returns the costs of c minus the costs of c's $subcalls$.
$funcset$	$2^{\mathbb{C}} \to \mathbb{F}$	Returns the set of functions that are either caller or callee of a call in the input call set.

Table 2.2: Convenience operations on a trace $T = (C,F)$ and a call $c \in C$ or on a set of calls $C' \subseteq C$, respectively.

The developer, who performs the trace visualization process with the aim of understanding system behavior, needs to execute the software system to capture a trace.

2. **Fact Analysis**: In the fact analysis step, the trace is combined with additional facts on the system, e.g., facts describing the static structure of the system.

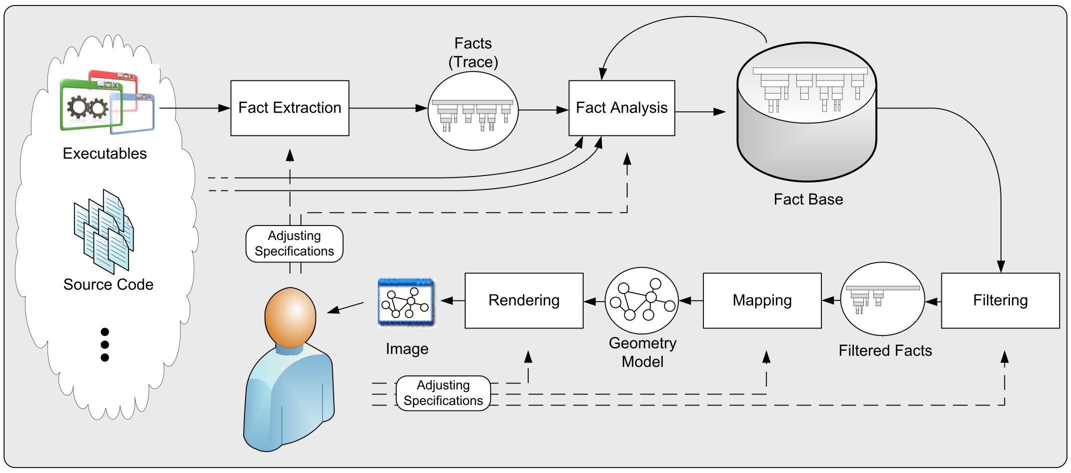


Figure 2.6: The trace visualization process.

Additionally, further facts are derived, e.g., by way of abstracting or grouping facts. The analysis step stores the results in a *fact base*.

3. **Filtering**: In the filtering step, a subset of the facts is taken from the fact base according to a developer-defined query. Defining the filtering this way differs from any definition evolved in the context of visualization. According to this definition, data preparation aspects are not included in the visualization related definition of *filtering* but are located in the fact analysis step.

4. **Mapping**: The fact subset is transformed into a geometry model. The facts are represented by geometric primitives and their associated attribute values such as form, size, orientation, position, color, brightness, and texture.

5. **Rendering**: The geometry model is rendered; the resulting images are presented to the developer.

The three steps *filtering* (3), *mapping* (4), and *rendering* (5) are combined within the term *fact presentation*. Fact presentations define *views* on the fact base. The trace visualization process is an interactive process. In other words, the developers interactively adapt the views to facilitate the building of mental models on the system's structure and behavior.

2.5 Extending Traces with Module Hierarchy Information

Functions as smallest named implementation units are grouped into more coarse-grained implementation units, forming a hierarchical structure – the *module hierarchy* of the software system.

The term *module* is used in this work as defined by Clements et al. [35].

Definition 16 (Module) A module is an an implementation unit of software that provides a coherent unit of functionality. [...] It is a hierarchical element, i.e., an element that can consist of like-kind elements. A module can consist of submodules that are themselves modules.

The module hierarchy of a software system can be represented as tree $M = (V,H) \in \mathbb{M}$ where $V \subset \mathbb{V}$ is a set of nodes representing modules and $H \subset \mathbb{H} = \mathbb{V} \times \mathbb{V}$ is a set of tree edges representing the hierarchical containment relation between modules.

Table 2.3 defines operations on a module hierarchy $M = (V,H)$ that simplify the description of algorithms.

Figure 2.7 shows the data model for traces combined with module information using the UML notation. Functions are leaf nodes in the module hierarchy.

A trace $T = (F,C)$, whose functions $f \in F$ are leaf nodes in a module hierarchy $M = (V,H)$ are referred to as *compound graph*. A compound graph [204] is a graph with two sets of edges: adjacency edges and hierarchy edges. The hierarchy edges are regarded as an inclusion relationship. Depending on the notion, hierarchy edges are required to be acyclic or to form a tree.

Operation	Domain and Range	Result
parent	$\mathbb{V} \rightarrow \mathbb{V}$	Returns the module that contains v.
ancestors	$\mathbb{V} \rightarrow 2^{\mathbb{V}}$	Returns the set of modules obtained from recursively collecting *parent*s starting with v.
descendants	$\mathbb{V} \rightarrow 2^{\mathbb{V}}$	Returns the set of modules that have v as an ancestor.

Table 2.3: Operations on a module $v \in V$ of a given module hierarchy $M = (V,H)$.

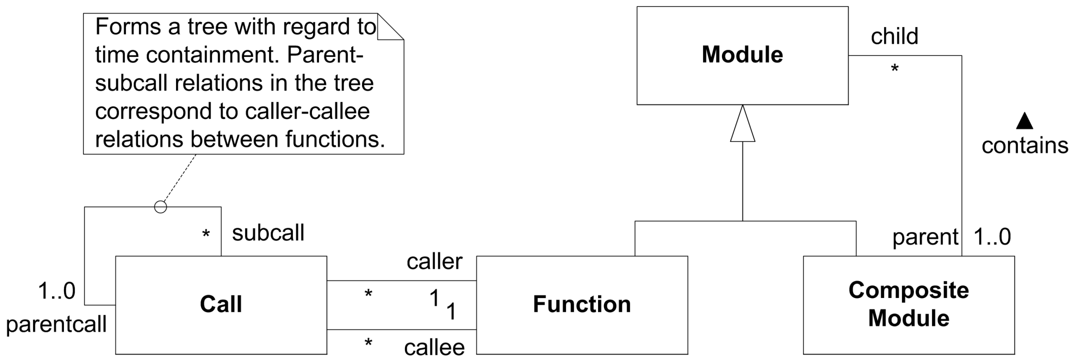

Forms a tree with regard to time containment. Parent-subcall relations in the tree correspond to caller-callee relations between functions.
Module
child
*
▲ contains
*
subcall
caller
parent
1..0
1..0
parentcall
Call
*
1
1
Function
Composite Module
*
callee

Figure 2.7: Data model of traces extended with module hierarchy information in the UML notation.

2.5.1 Module Names and Function Semantics

When developers explore traces, they need to understand the functions' responsibilities for the exhibited system behavior. The comprehension process is accompanied by interpreting function names and function call relations. Further semantics conveying information on functions can be obtained from taking the module hierarchy into account. Developers may derive additional information on a function's semantics from the names of the modules of which the function forms a part. The purpose of a function called `callOnStartRequest()`, for example, can be assessed more easily, if the developer is aware of the fact that the function is contained in the module hierarchy of `Channel`, `Http`, `Protocol`, and `Network`. To facilitate the exploration of traces, module names can be evaluated [14].

2.5.2 Reconstruction of Module Hierarchies

In the field of *architecture recovery* [76, 100, 146], various techniques have been proposed to reconstruct the system's module hierarchy from its implementation. Such techniques are required, as in the case of many long-living software systems an explicit and up-to-date documentation of its modules and their relationships is rarely available. The techniques include the analysis of syntactical elements in the source code (e.g., class, package keywords), the analysis of how code is organized (e.g., in nested directory structures), the analysis of naming conventions, the application of clustering algorithms to group code artifacts according to their dependencies [131].

Naming conventions and code partitioning within directory structures can be exploited to reconstruct the module hierarchy of C/C++ software systems [8, 14]. The approach follows the hierarchical reflexion models proposed by Murphy et al. [146]. A developer provides a mapping table consisting of regular expressions and target modules. Each function's name and its implementing file is checked against the regular expressions and is then assigned to a module. A basic mapping table is automatically derived from the directory structure of the source code or from the package structure. Afterwards, the mapping can be fine tuned manually to incorporate existing knowledge on the module hierarchy of the system.

Depending on the reconstruction technique, various source code metrics can be determined per module. Examples of such metrics include lines of code (LOC), lines of comments, fan-in, fan-out, number of methods or functions in a class or file, and number of global variables. For a detailed discussion see Lorenz and Kidd [122] or Lanza and Marinescu [118].

2.6 Maintenance Tasks Facilitated by Trace Visualization

A large set of maintenance tasks are related to system *features.* Eisenbarth et al. [54] define a feature as follows.

> **Definition 17 (Feature)** A feature is a system behavior that can be triggered by the user of the system and produces some kind of output that is visible to the user.

Many feature-related maintenance tasks can be supported by trace visualization. Moreover, the fact that the user knows when the feature starts and when it ends is a real advantage here. Such tasks include:

Feature Location Requests for modifying and extending existing software systems are in most cases submitted and expressed by end-users in terms of features [70, 137]. In order to meet such a feature change request, developers need to translate it into terms of source code. First, they have to locate the source code components responsible for feature functionality. Second, they need to understand how these components interact. In the case of complex software systems, this is a cost intensive and time consuming task. Trace visualization can prove useful to developers performing this task because it reduces the vast amount of source code enabling developers to inspect the executed parts and as it provides higher-level views on the executed code.

Fault Localization If a failure occurs while executing a software system, i.e., if the system fails to do what it should do, it is often time consuming to locate the failure-causing fault (bug, defect) within the code. In complex software systems the executed functionality is usually scattered throughout the system implementation. Subtle and often undocumented couplings exist between different parts of the implementation.

Furthermore, code corresponding to functionality that exhibits the failing behavior is frequently not the source of faulty system behavior. Hence, a developer first needs to identify the code that corresponds to the functionality that is exhibiting the failing behavior. Having identified the starting point, the developer needs to go back in time and analyze the control flow until the fault, i.e., the *origin* of the failure, is found. With trace visualization, however, developers record the control flow before and while the system exhibits the failure. Later, they are able to analyze the system behavior back and forth in time as if they had a time machine. For instance, they can analyze what happend before a crash occured.

CHAPTER 3

Related Work

This chapter starts by outlining general *software visualization* approaches, including those for *trace visualization*. Explanations are given on where and how tracing, trace reduction, and trace presentation concepts proposed in this thesis differ from existing ones. In addition, an overview is provided on related work to techniques for software maintenance tasks of *fault localization* and *feature location*.

3.1 Software Visualization

Software visualization has its roots both in software engineering and information visualization. Various taxonomies have been introduced that structure software visualization research [129, 148, 149, 173, 181]. Myers [149] distinguishes between software visualization techniques that present *data*, *code*, and *algorithms*. In an additional dimension, the techniques are partitioned into *static* and *dynamic* approaches. Price [173] classifies software visualization techniques according to the following aspects:

- Scope: Range of software systems used as input for the visualization.
- Content: Kind of information visualized.
- Form: Characteristics of the output of the visualization system.
- Method: Specification of the visualization.
- Interaction: Means of controling the visualization system.
- Effectiveness: Effectiveness of conveying information to the user.

As to the *form*, one distinguishes between 2D versus 3D visualizations [132, 199]. Teyseyre and Campo [210] give an extensive overview of 3D software visualization tools and techniques.

Zhang [234] distinguishes between aspects of software visualization (visual modeling, visual database query, visual programming, algorithm animation, program visualization, data visualization, and document visualization) according to their applicability during the stages of a software system's life cycle. In contrast, Mili and Steiner [141] focus on software engineering and group software visualization approaches into those for *construction* and those for *analysis*. Diehl [49] further restricts software visualization approaches to program understanding and reverse engineering activities and partitions software visualization techniques according to aspects of the software system that need to be understood: the structure of a software system, its evolution, and its behavior.

We structure related work on software visualization according to Diehl's approach.

3.1.1 Visualization of the Software System's Structure

UML Based Approaches Many tools and systems generate *UML* diagrams [183] from source code. These diagrams have been established as a standard for visually documenting and planning forward engineering activities. Examples of UML based tools and systems include *Borland Together* [18], *IBM Rational* [90], *ESS-Model* [56], *Fujaba* [79], *GoVisual* [160], and *BlueJ* [112]. The major challenge when using UML diagrams for reverse engineering is scalability and the applied layout algorithm that should keep edge crossings to a minimum [53].

Generic Graph Based Approaches A variety of tools and systems provide generic graph visualizations on the software system's structure. With *Rigi* [142, 143] the source code is parsed and an initial graph is presented. The developer explores the graph and aggregates nodes and, in this way, creates higher-level abstractions of the system structure. Various approaches follow the same process, e.g., *PBS* [61], *Sotograph* [78], and *Bauhaus* [110, 174].

Source navigator [156] is a graph based tool for visualizing call graphs and variable usage dependencies for C, C++, Java, Tcl, FORTRAN, and COBOL systems. *Understand* [187] is a system understanding tool that provides a variety of graphs visualizations. Similar visualization techniques are provided by *Imagix 4D* [88].

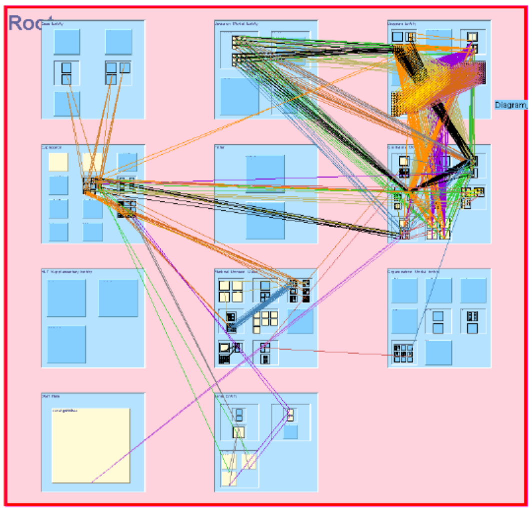


Figure 3.1: The *SHriMP* visualization system [203].

SHriMP [201, 203]—and its *Eclipse* plugin derivate *Creole* [138] for Java software systems—visualizes structural system dependencies as interactive node-link graphs (Figure 3.1). In multiple linked views, developers navigate through the system structure. A similar approach is adopted by *VizzAnalyzer* [81, 163]. Favre [60] proposes *GSEE*, a generic graph based software exploration framework that focuses on visualization of object-oriented and on component based software systems. Telea et al. [209] have introduced a graph based toolkit for prototyping reverse engineering visualizations.

Lanza and Ducasse [117] have presented *class blueprints*, a visualization technique that depicts dependency relations between classes. The classes' methods are, thereby, grouped into layers of *initialization*, *interface*, *implementation*, and *access*.

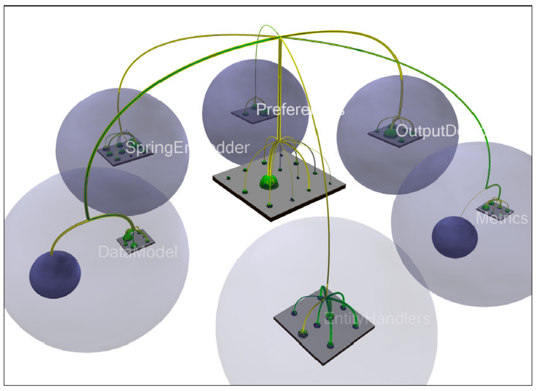


Figure 3.2: Balzer et al. use a landscape metaphor to visualize the structure of a software system [4].

Balzer et al. [4] present the hierarchical structure of a software system using a 3D landscape metaphor (Figure 3.2). Using a similar metaphor, Panas et al. [123] use 3D and visualize with *Vizz3D* a software system's structure as 3D city. Likewise, Wettel and Lanza [227] use a city metaphor to visualize the structure of software systems together with metrics on the structural elements.

An inherent problem of graph based software visualization tools and systems is the implementation of the graph layouting algorithm [147]. Most approaches use generic graph drawing tools and systems for this. Among numerous tools, we mention: *Graphviz* [64], *Graphlet* [19], *OGDF* [159], and *Prefuse* [77].

Mili and Steiner [141] point out that generic graph drawing tools may not always be the appropriate choice because "*these tools do not take into account the drawing conventions of accepted software engineering notations. Additionally, these layout tools do not scale. In re-engineering, software artifacts may necessitate the visualization of graphs consisting of tens of thousands of nodes.*"

Metrics Based Approaches Metrics based software visualization approaches do not primarily focus on dependency or containment relations between elements of the system structure. On the contrary, they organize the visual representations according to metrics values on the elements. *CodeCrawler* [46, 50] uses the concept of *polymetric*

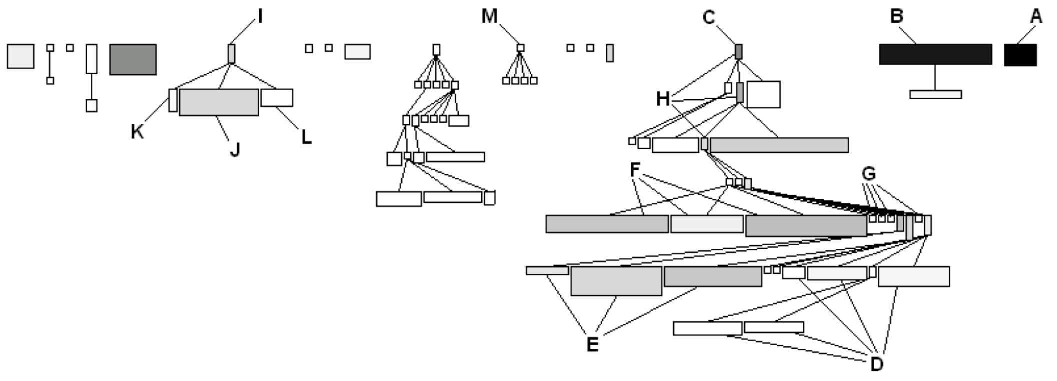
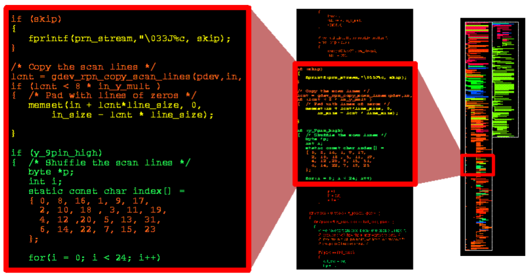


Figure 3.3: CodeCrawler visualizes metrics values about structural elements of a software system [50].

views that maps values of object-oriented metrics on the attributes of box-shaped representations of the elements, e.g., width and height (Figure 3.3). *sv3D* [133] in addition exploits the third dimension to visualize metrics values.

3.1.2 Visualization of the Software System's Evolution



```
if (skip)
{
   fprintf(prn_stream,"\033J%c, skip);
}

/* Copy the scan lines */
lcnt = gdev_rpn_copy_scan_lines(pdev,in,
if (lcnt < 8 * in_y_mult )
{  /* Pad with lines of zeros */
   memset(in + lcnt*line_size, 0,
       in_size - lcnt * line_size);

}

if (y_9pin_high)
{  /* Shuffle the scan lines */
   byte *p;
   int i;
   static const char index[] =
   { 0, 8, 16, 1, 9, 17,
     2, 10, 18 , 3, 11, 19,
     4, 12 ,20, 5, 13, 31,
     6, 14, 22, 7, 15, 23
   };

   for(i = 0; i < 24; i++)
```

Figure 3.4: The Seesoft technique visualizes metrics values on miniaturized source code [3].

The *SeeSoft* technique [3, 52] introduced by Eick et al. [52] uses colored, miniaturized source code views to visualize metrics on code changes (Figure 3.4). Various tools and systems adopt this technique to visualize metrics data [62, 72, 99, 133, 161]. Griswold et al. [72] propose the *Aspect Browser*, a tool for visualizing evolution metrics. For the same purpose, Froehlich and Dourish [62] introduce *Augur*.

Other tools and systems depict evolution using graph representations. *Gevol* [38] converts for different development snapshots the structure of a Java software system to animated sequences of graph representations. With *YARN* [82], modules of the software systems are located on a circle. Edges between the modules denote their dependency relations. The system animates through the different versions of the system and reveals the change in dependencies.

Voinea et al. [223] propose *CVSscan* and *CVSgrab* that present each version of the system as a colored line and show the system evolution as a series of these lines. The color encoding can be configured so that various characteristics of system evolution can be analyzed, e.g., coding activity of individual developers.

3.1.3 Visualization of the Software System's Behavior

We distinguish between approaches for educational purposes (section algorithm animation), for visualizing system behavior on a microscopic scale (section micro scale behavior visualization), for time-aggregated control flow visualization (section time-aggregated visualization), and for visualizing sequences of function calls (section trace visualization). Trace visualization approaches are discussed in a separate section.

Algorithm Animation The tools and systems for algorithm visualization are designed to visualize single algorithms and are usually applied in educational contexts. Brown and Sedgewick [23] propose the algorithm animation system *BALSA*. *TANGO* [198] is introduced by Stasko. *BALSA* and *TANGO* both have a variety of extended successor versions. An extensive overview of algorithm animation tools and systems is given by Kerren and Stasko [102].

Micro Scale Behavior Visualization Examples of diagrams that are visual representations of the source code include *control-flow graphs* [67], *Jackson diagrams* [93], and *structograms* [151]. Due to the detailed information that these diagrams depict, they are limited to display a small fraction of the code, e.g., the implementation of a single function.

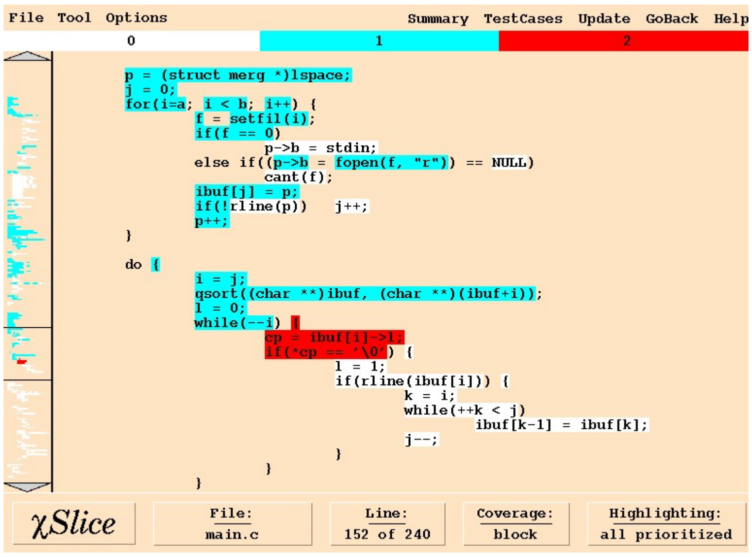
xSlice [34] computes a dynamic slice for a specific point in the code and a given program state. That is, it computes the set of all program points that affect the given program point for the given input. The slice is used to highlight the corresponding code lines in a source code view (Figure 3.5).

Röthlisberger et al. [182] integrate runtime information into the Eclipse IDE. Via pop-up windows developers receive information on call statistics of methods that are selected with the mouse pointer.

Related to the visualization of behavior on a microscale are tools that visualize the memory state at a specific point in time. The Data Display Debugger *DDD* [233] depicts nested variable references. It can be used as an extension to command line debuggers such as GDB on Linux and permits developers to visually unfold nested data structures, e.g., lists. An extension of DDD as proposed by Zimmermann and Zeller [235] visualizes *memory graphs* to show reference dependencies between variables.

Time-Aggregated Visualization Visualization techniques that aggregate trace data over time focus on two distinctive tasks: analyzing performance and assessing test coverage.

Performance analysis tools and systems, such as *Intel*'s *VTune* [89] or the *Unix* tool *gprof* [68], typically present system behavior in textual form or as call graphs that are expanded interactively. Function calls are annotated with their cumulated runtime



```
p = (struct merg *)lspace;
j = 0;
for(i=a; i < b; i++) {
        f = setfil(i);
        if(f == 0)
                p->b = stdin;
        else if((p->b = fopen(f, "r")) == NULL)
                cant(f);
        ibuf[j] = p;
        if(!rline(p))    j++;
        p++;
}

do {
        i = j;
        qsort((char **)ibuf, (char **)(ibuf+i));
        l = 0;
        while(--i) {
                cp = ibuf[i]->l;
                if(*cp == '\0') {
                        l = 1;
                        if(rline(ibuf[i])) {
                                k = i;
                                while(++k < j)
                                        ibuf[k-1] = ibuf[k];
                                j--;
                        }
                }
        }
```

Figure 3.5: xSlice shows which code lines affect a given program point.

costs which permits developers to identify those functions that are on average costly. Moret et al. [145] apply a radial, space-filling hierarchy visualization technique [197] to depict the costs information of a call graph.

For visualizing test coverage, *Tarantula* [99] and *Gammatella* [161] use minituarized source code views and encode by color whether a code line has been executed involving tests or not.

3.2 Trace Visualization

In the field of trace visualization, a number of approaches have been proposed.

Shimba [205, 207, 208] is a tool for understanding the behavior of Java systems. The developers explicitly choose artifacts they wish to be traced (e.g., classes, interfaces, methods, etc.) Here, the structure visualization tool Rigi is used. For trace presentation, Shimba uses *SCED* [206], a trace visualization frontend for creating and manipulating sequence diagrams and state machines similar to the respective *UML* diagrams used in forward engineering [155, 183]. SCED is able to detect repeated sequences of identical calls. Shimba's purpose is not to provide views on the overall behavior of a software system but on the behavior of manually selected artifacts. Hence, Shimba does not run into any of the scalability issues related to very large traces.

JaVis [136] extracts method calls in concurrent Java software systems via the *Java Debug Interface* and visualizes the trace with the UML forward engineering tool *Borland Together* [18].

PV [104] visualizes traces generated on *AIX* systems—*IBM*'s version of the *UNIX* operating system—with the aim of exposing performance bottlenecks. Traces contain various kinds of events: from hardware-level performance information to application-level-activity (such as execution time profiles). PV provides time-aggregated graph views on a trace and *colorstrip* views that depict a user-selected event type over time. The *AixProcess Colorstrip* visualizes the activity of all running processes overtime—each process is mapped on a color value.

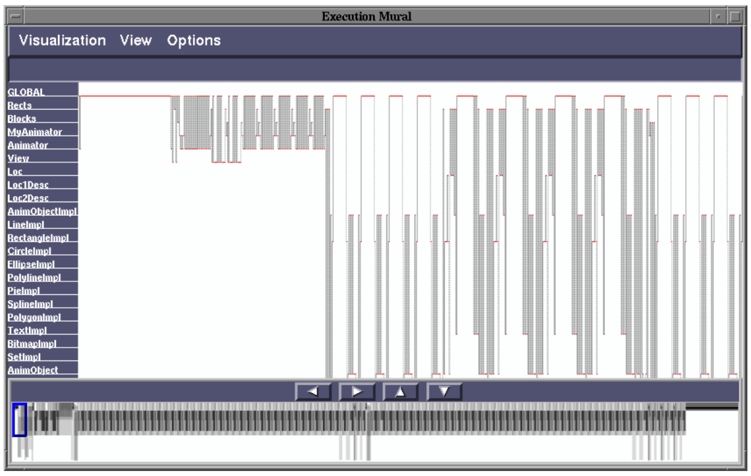


Figure 3.6: The *information mural* visualization technique.

ISVis [94–96] is a trace visualization frontend for object-oriented software systems that introduces the *information mural* visualization concept (Figure 3.6) for presenting traces in a highly condensed way. Objects occupy thin vertical lines, and color encodes along the vertical time dimension when messages are sent. In addition to this coarse-grained view, a view similar to an UML sequence diagram is provided. In this view, a pattern detection algorithm summerizes identical call sequences. Furthermore, ISVis provides a querying mechanism that allows developers to search the trace for patterns using wildcards.

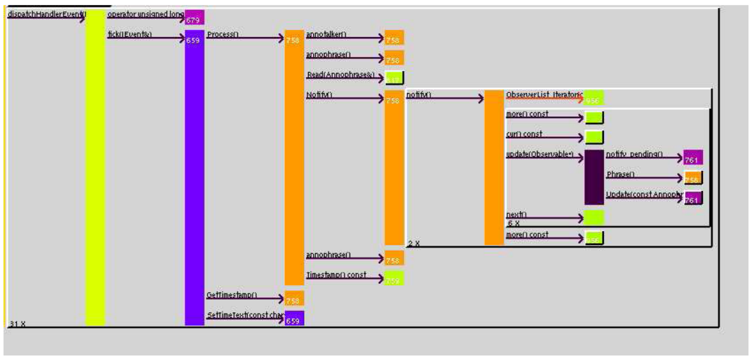


Figure 3.7: Visualization technique of the *Ovation* tool.

Ovation [45] is a trace visualization tool for Java systems. Traces are explored by collapsing and expanding call subtrees. Filtering mechanisms exist based on manually selecting objects or classes. Repetitive parts in sequences of subcalls are compacted by means of identifying similar method calls and detecting call patterns (Figure 3.7). Similarity can thereby be defined in various ways, e.g., methods of the same object, methods of the same class, methods with call stack depth under a given threshold, etc.

Jinsight [44] focuses on performance analysis in Java systems. Some of Jinsight's views aggregate trace data over time. The *histogram view* shows overall CPU or memory consumption of objects or classes. Likewise, the *call tree view* gives a summarized CPU consumption of calls – a view that is provided by most performance profilers. The *execution view* shows the trace over time. Calls are represented by bars that are placed along the vertical dimension (time). In the horizontal dimension, the bars are placed according to their call stack depth. The execution view provides zooming facilities, i.e., it permits choosing the time range to be shown, while the *reference pattern view* depicts interactions between objects. This view essentially implements the Ovation concepts. Some of the concepts having become part of the *Eclipse* project *TPTP* [51] and of *Zinsight* [92], an internal IBM tool for debugging and performance analysis on System Z.

Like Shimba, *Program Explorer* [116] is a tool for understanding C++ systems. For tracing they modify the *GCC* compiler as performed by the *SPYDER* tool [2]. Moreover, they use *IBM*'s *Heap View Debugger* technology available on *IBM*'s *UNIX* operation system *AIX* [91] to trace the creation and deletion of objects. Program Explorer visualizes traces as *object graphs* showing selected objects and their interactions. The authors state that manual pruning of the graph is necessary to obtain an object graph that can cognitively be processed. Program Explorer additionally provides a view similar to an UML sequence diagram.

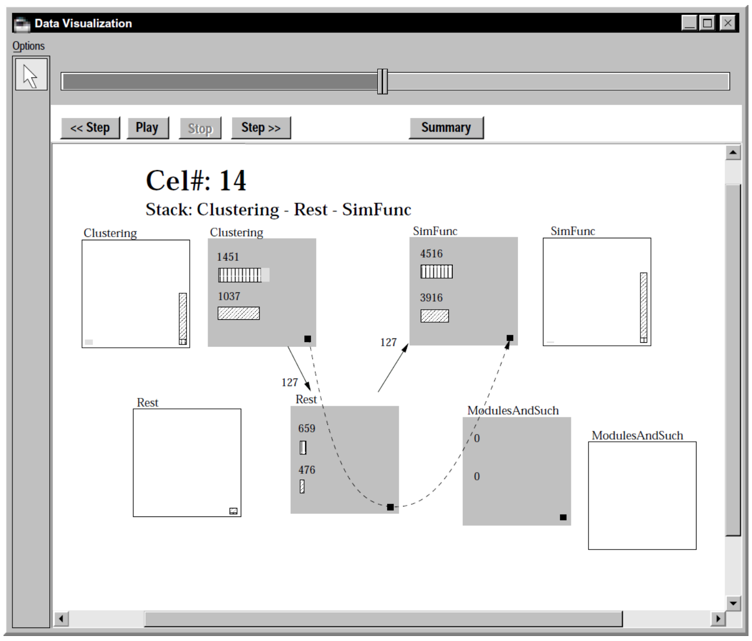


Figure 3.8: Visualization technique of the *Avid* tool.

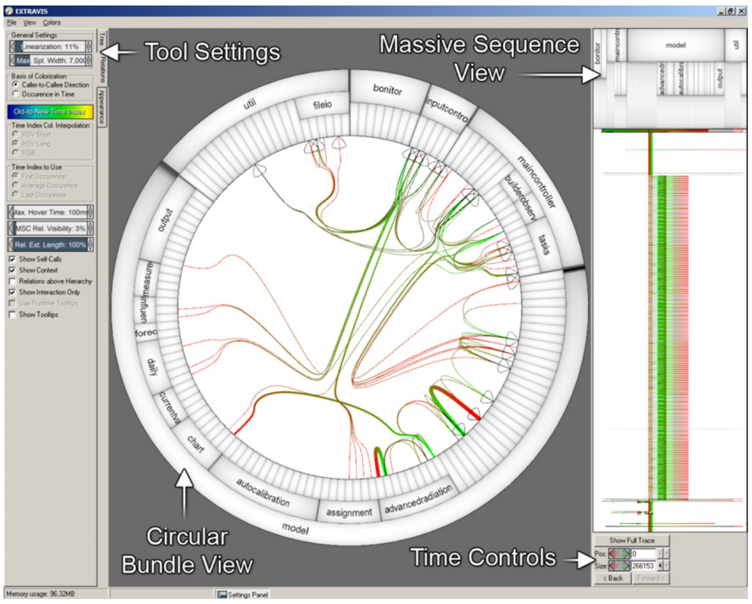
AVID [224] is a trace visualization tool for Smalltalk systems. Trace data is obtained via instrumenting the Smalltalk virtual machine. AVID creates views that primarily show the system's modular structure, which has to be defined manually beforehand. On top of the structural visualization, the dynamic information contained in a trace is overlayed (Figure 3.8). Namely, the function call stack for a given point is lifted up in such a way that it represents a module call stack that connects the displayed modules.

Scene [111] uses source code instrumentation by means of applying a preprocessor before code compilation to gather traces in object-oriented systems. The views that Scene presents are similar to UML sequence diagrams. In order to cope with scalability issues involving large traces, they may be interactively reduced to method calls of user selected objects. Furthermore, calls may be collapsed and expanded interactively allowing for the filtering of subcalls. Hence, these calls are prevented from being shown.

The *Collaboration Browser* [180] operates on Smalltalk systems. A user selects objects or methods to reduce the trace to calls related to selected artifacts. A view similar to a UML sequence diagram shows the reduced trace.

DJVis [193, 194] visualizes the execution of Java systems. A *runtime view* shows the call stacks of all threads. The *class view* depicts a graph of classes and their relationships. Class metrics are encoded on the nodes' shapes and colors.

Greevy et al. [71] propose a trace visualization system for Smalltalk systems. After recordeding a trace, developers are able to step through the trace and analyze in a $2\frac{1}{2}$D graph visualization how objects are created and how they send messages. The third dimension is used to pile up boxes on one another that represent objects of the same class.



General Settings
Linearization: 11%
Max. Spl. Width: 7,000
Basis of Colorization
[✓ Caller-to-Callee Direction]
Occurence in Time
Old-to-New Time Index
Time Index Col. Interpolation
HSV Short
[✓ HSV Long]
RGB
Time Index to Use
[✓ First Occurence]
Average Occurence
Last Occurence
Max. Hover Time: 100ms
MSC Rel. Visibility: 3%
Rel. Ext. Length: 100%
☑ Show Self-Calls
☑ Show Context
☐ Relations above Hierarchy
☑ Show Interaction Only
☐ Use Runtime Tooltips
☐ Show Tooltips
Pos: 0
Size: 266153

Figure 3.9: Screenshot of the *ExtraVis* tool.

ExtraVis [43] is a visualization frontend for presenting trace data. Two synchronized views permit developers to explore the trace (Figure 3.9). The *massive sequence view* is a variation on Jerding's information mural that depicts call events in a time-ordered way along the vertical dimension. The *circular bundle view* depicts the system's structural elements on the circumference of a circle, including their hierarchical structuring. Within the circle the edge bundles between structural elements in the circumference represent call relations. A trace is explored by selecting a time range in the massive sequence view and analyzing on the basis of the circular bundle view which call relations are active within the time range.

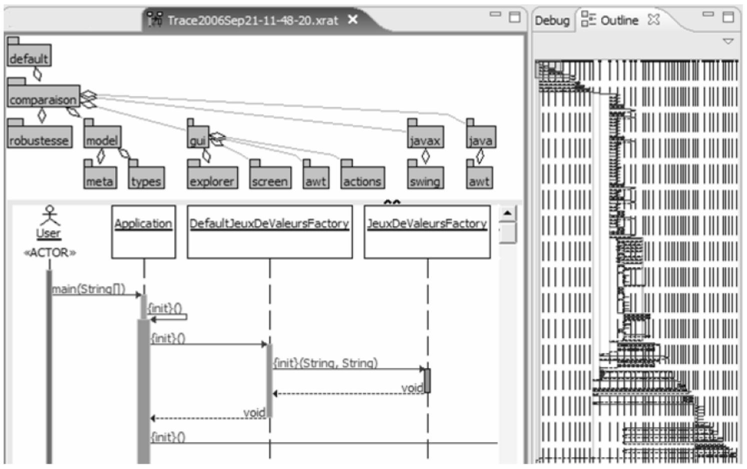


Figure 3.10: Screenshot of the *Zest Sequence Viewer* tool.

The *Zest Sequence Viewer* [6] provides 3 views on traces of Java systems (Figure 3.10): A UML sequence diagram; an information mural-like miniaturized view; and UML class diagram inspired view that shows the structural dependencies between the objects that are referred to in the trace.

The *ARE* tool [73] uses AspectJ to instrument Java applications and creates UML sequence diagrams from small traces of user-selected methods and classes.

Comparison to the Thesis' Tracing Technique

Most of the approaches do not detect *massively called functions*, i.e., functions that have a short execution time and are executed with high frequency. In this work, automatically detecting massively called functions at runtime and subsequently disabling the tracing mechanism, forms a core functionality to cope with the scalability problem of large traces because it reduces the amount of trace data right from the start of the trace visualization process. Hence, this technique could quite reasonably be integrated into the mentioned trace visualization approaches.

Hamou-Lhadj and Lethbridge [74] propose a technique that uses statically obtained fan-in metrics on functions to classify functions as *utility components* and remove them from a trace. This approach has the disadvantage that the developer needs to perform a static analysis on the source code, which may take a long time. In the case of a mid-sized 300k+ LOC code base, for instance, this may take some tens of minutes. Another disadvantage is that the technique does not guarantee any reduction on the size of the trace. Not all functions that are executed with high frequency and are therefore responsible for the size of the trace have a high fan-in metrics value.

Tracing Techniques in General There is broad research activity in the field of tracing as basis for a dynamic analysis. A variety of tools and systems record trace data to build debugger tools that permit developers to step back in execution. Such tools are referred to as *reverse debuggers* or *omniscient debuggers*. Most of these tools operate on virtual machine based languages such as Java or Smalltalk: *ODB* [120], *JIVE* [65], *Whyline* [107], *Unstuck* [84], a tool developed by Lienhard et al. [121], and *TOD* [172]. An omniscient debugger for embedded systems is *TimeMachine* [69]. Native languages on the *Linux* operating system are supported by *UndoDB* [217] and *Chronicle* [158].

Tracing techniques for C/C++ software systems have in common that they "implant" event generating code into the software system's binaries, i.e., they perform *code instrumentation*[1]. For C/C++ software systems, code instrumentation can be implemented by modifying source code, modifying binary code while or directly after building the executable, and modifying binary code at execution-time.

The tracing technique proposed in this thesis uses a hybrid approach. It applies both compile-time and execution-time instrumentation. Execution-time instrumentation is done by redirecting control, which is conceptually the execution-time approach with the smallest runtime overhead. The disadvantage of redirecting control is, however, that it is difficult to implement in a robust way. Redirecting control usually relies on *code splicing* [211], which means that instructions from the original binary code are cut out, augmented, and moved to a newly allocated code block.

The tracing technique as proposed in this thesis introduces a new strategy: After obtaining a fully instrumentated binary, the event generating code, which is inserted by the compiler, is replaced with `NOP` assembler instructions. Essentially this reverts the instrumentation. At execution-time, instrumentation becomes trivial and highly robust. It can be done employing standard facilities provided by debuggers. One only needs to replace the `NOP`s with the original instrumentation code. This strategy avoids code splicing by using the compiler for the execution-time instrumentation part.

Existing tracing techniques based on execution-time instrumentation perform instrumentation either by *execution interposing* or by *code editing*:

- Interposing execution means that code is seen as data and is analyzed and modified by a controlling process before being passed on to the processor. Examples of tools working this way are *Valgrind* [152], *Pin* [124], *Shade* [37], and *DynamoRIO* [24].

- Code editing techniques modify binary code within a running process.
  One particular way of code editing is to insert traps, e.g., `int3` instructions, into

1 This discussion ignores approaches based on hardware-generated events because they are either imprecise, e.g., sampling profilers, or require specialized hardware, e.g., last branch recording.

the code and enforce a trap handler to react when the processor executes the trap. Debuggers are very popular tools that work in this way. Some debuggers, e.g., *WinDBG*, which is part of the *Microsoft's Debugging Tools for Windows* [140], provide tracing facilitities based on traps. Another instrumentation tool based on traps is *DTrace* [28]. Tracing via traps evokes a larger performance overhead than techniques based on direct redirection of control [83].
A more general way of implementing runtime code editing is to do *code splicing*. Tools which use this approach are *Paradyn* [85], *Detours* [87], and on some platforms *DTrace* [28].

Comparison to the Thesis' Trace Pruning Technique

Existing trace visualization tools basically provide two kinds of views for trace exploration: (1) Macroscopic views present trace data in a cumulative, overview-like way. These views are necessary for identifying those time ranges in the trace that are relevant to the maintenance task at hand. (2) Microscopic views depict with detail sequences of selected function calls. These views are essential, if specific artifacts or time ranges have already been identified as task relevant and the developer needs to understand details of them.

Navigation from a macroscopic to a microscopic level is implemented in two ways: (1) A macroscopic and a microscopic view are linked and the macroscopic view is used to define that section of the trace which is visualized within the microscopic view. This process is driven by visually recognizing patterns in the macroscopic view. (2) Starting from the root call in a call tree, developers successively expand previously collapsed function calls, thereby navigating to calls located deeply in the call tree.

The trace pruning algorithm and the respective visualization technique as presented in this thesis, bridge the gap between coarse-grained trace overviews and detailed views that depict only short selected time ranges. Developers receive compact intermediate trace visualizations that guide them during top-down exploration which enable them to find task relevant parts of the trace—even in very large traces. The algorithm for automatically reducing the calls in the trace to those that represent key decision points during top-down exploration is a novel approach that could in effect be integrated into the mentioned trace visualization tools.

The call similarity metrics proposed in this thesis as a basis for being able to provide compact presentations of pruned traces obtained by applying the pruning algorithm, is an extension of the call generalization approaches of *Ovation*. It differs from the existing approaches as it defines similarity by means of comparing call fingerprints in a fuzzy way, which represents a more coarse-grained metrics than those used in *Ovation*. Additionally, the metrics' continuous and clearly defined value range permits precise control where two calls are classified as being similar, which, in turn, facilitates detecting outlier calls.

Phase Detection

The calls that remain in a *pruned trace* after applying the pruning algorithm split the time range that the trace spans into phases. Reiss [175, 176] proposes *JIVE*, a visualization framework for showing performance characteristics of running Java software systems. A phase detection algorithm identifies cost-intensive parts of the execution. This approach is useful for performance analysis. It cannot be applied to program comprehension because the algorithm misses short phases that are important for understanding a system's behavior.

In the field of program optimization, various phase detection algorithms have been suggested. The goal of these algorithms differ, however, from the goal of splitting a trace into phases for *program understanding.* The purpose if the algorithms is to ascertain the system being in a specific execution phase and to predict the succeeding phase. This way, the system can anticipatorily be prepared for faster execution of the succeeding phase. The algorithms are not designed to detect the exact boundaries between the phases. This, however, is essential for comprehension purposes [108]. Nagpurkar et al. [150] propose an online phase detection algorithm for program optimization. Sherwood et al. [188] introduce an off-line algorithm that uses clustering applied to basic block vectors to calculate phases.

Comparison to the Thesis' Linked Trace Presentation Techniques

The trace presentation techniques put forward in this thesis are motivated by existing techniques. The *temporal overview*, for instance, is a variation of the *information mural* technique proposed by Jerding et al. [94]. The *call stack view* is similar to the *execution view* in Ovation [45]. The *enriched code view* is an application of the *SeeSoft* technique proposed by Ball et al. [52].

This thesis introduces a uniform framework for trace presentation techniques. It provides solutions to the question of how a basic set of trace views can be implemented which assist developers in performing top-down and bottom-up comprehension strategies. In particular, the thesis shows how trace data taken from a fact base is filtered, transformed into a geometry model, and finally converted into an image. Additionally, the approach presented shows how developers may use different views on a trace simultaneously enabling them to cross-reference findings made in a single view with context information provided by other views.

Comparison to the Thesis' Technique for Combining Trace Visualization with other Analysis Techniques

Most trace visualization approaches rely on detailed pre-existing knowledge on the software system's implementation when having to cope with large traces. Developers explicitly need to choose artifacts of interest, e.g., by selecting classes, objects, or functions. The trace is then reduced to the calls that are related to the selected artifacts. The problem with this approach is that in many cases the developer is not aware of the task-relevant artifacts beforehand.

The concepts proposed in this thesis show systematically how existing analysis techniques can be combined with trace visualization with the aim of solving this problem. For many maintenance tasks, specialized analysis techniques exist and produce sets of artifacts that may serve as a starting point for fine-grained trace analysis. Hence, the proposed concepts can be integrated into all of the trace visualization approaches mentioned above in those instances where a developer starts exploring the trace on a microscopic level.

3.3 Maintenance Tasks

Trace visualization is an essential instrument for supporting specific maintenance tasks, namely *fault localization* and *feature location.*

3.3.1 Fault Localization Techniques

Various automatic techniques exist to localize faults in those cases where a set of regression tests is to hand. Some approaches calculate the difference of statement coverage between passing and failing test cases [1, 178]. The identified code locations then serve as starting points for manually performing fine-grained code analysis based on the system dependency graph (SDG) [86]. Jones et al. [99] calculate a suspiciousness value for every line of code, based on their execution frequencies in passing and failing test cases. To support developers in their efforts to identify highly suspicious code lines, they go on to propose a visualization technique that encodes the suspiciousness value into color. Cleve and Zeller [36] propose a technique that runs both a passing and a failing test case in a debugger. The system is stopped at multiple code locations and part of its memory state is swapped between the two runs until the smallest memory state has been isolated that causes the test case to fail. The technique is repeated at several code locations in order to identify the ones that are critical to the test's outcome. These locations then serve as a starting point for manual code analysis, using the SDG in a similar way to that proposed by Renieris and Reiss [178].

All these approaches differ from the fault localization technique proposed in this thesis in that they operate on different input data, namely a set of test cases. The approach here, however, exploits code change information stored in a *software configuration management* system (*SCM*) and combines it with runtime information from a single run. A similar approach that uses code change information to localize the cause for a failing test case is proposed by Ren et al. [177]. Their idea is to model recently introduced changes as a set of atomic changes such as add class or delete method. Subsequently, a semi-automated process is started where an intermediate executable is built by applying different sets of atomic changes to the original version of the system. Next, the test case is applied to this intermediate version of the system, isolating the minimal set of atomic changes that causes the test case to fail. The approach differs from the one given in this thesis that explicitly maps runtime information onto code change data with the objective of filtering irrelevant changes.

In this thesis, two key concepts are complementary to related approaches: (1) Mapping code change data onto executed parts of the system implementation. (2) Providing a means of visual exploration as to when and how introduced modifications are executed. The concepts presented are of a general nature and can be adopted by other approaches as well.

3.3.2 Feature Location Techniques

Wilde et al. introduce *Software Reconnaissance* [228, 229], a feature locating technique based on a comparison of traces from test cases with and without feature execution. Eisenberg and DeVolder [55] extend this concept by comparing automatically produced traces based on pre-existing test suites. Other research on locating feature implementation is based on *Program Dependency Graphs* [162], which have been extracted by static analysis. *RIPPLES* [31] supports the user during manual exploration of a dependency graph by way of 2D graph visualization. The user decides whether a component, e.g., function, basic block, or statement, is relevant to the feature and adds it to the search graph that finally represents the feature implementation. Eisenbarth et al. [54] first compare 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.

The concept of scalable trace visualization proposed in this thesis is complementary to the approaches referred to and can be used in combination with them.

CHAPTER 4

A Scalable Technique for Tracing Function Calls in C/C++ Systems

Tracing visualization techniques denote techniques for extracting function call sequences from a running software system. The applied tracing technique represents a crucial bottleneck in the trace visualization process. Major difficulties when tracing techniques include: (1) It is difficult to "weave" the technique into existing build processes. (2) The amount of data collected during tracing is generally huge. (3) The runtime overhead of the tracing technique can be disturbing.

This chapter introduces a scalable and robust technique for C/C++ software systems that is easily integrated into complex build processes and applies to a wide range of platforms. The key ideas are:

1. Exploiting common instrumentation facilities provided by most compilers and common runtime code-modification facilities provided by most debuggers. By relying on common compiler and debugger facilities, a high degree of applicability, easy integration into existing build processes, and a measure of robustness is attained.

2. Providing a technique that analyzes how frequent functions are executed during tracing and which selectively disables—at runtime—functions from being traced whose call frequency exceeds a developer defined threshold. Automatically eliminating functions called with high frequency reduces (1) the size of the resulting trace and (2) the runtime overhead of the tracing technique.

Section 4.1 elaborates on the requirements of tracing techniques that are to be applied as part of a scalable trace visualization tool. In Section 4.2, a tracing technique is proposed that reverts standard compile-time instrumentation to enable developers to apply robust execution-time instrumentation. Finally, in Section 4.3, a technique is proposed for identifying functions called with high frequency and for disabling tracing for them at execution-time.

4.1 Tracing Techniques as Part of Scalable Trace Visualization

A tracing technique that provides a basis for a scalable trace visualization tool needs to fulfill a variety of requirements related to integration into both existing build and maintenance processes, to tracing granularity, to applicability on different platforms, and to ways of being extended to or combined with techniques for gathering system state information.

Integration into build processes. Gathering traces from existing (legacy) software systems in an industrial environment is far from trivial. Zaidman enumerates challenges and constraints [232]:

- *The tracing technique should be applicable to the source code "as is". Otherwise, one would require knowledge of what is in the sources, and this is exactly what needs to be recovered.*
- *The existing build processes should remain in place, with only minimal alterations. To refactor the build system, considerable knowledge of its current internals is needed, but again this is lacking.*
- *The semantics of the original software system should remain intact.*

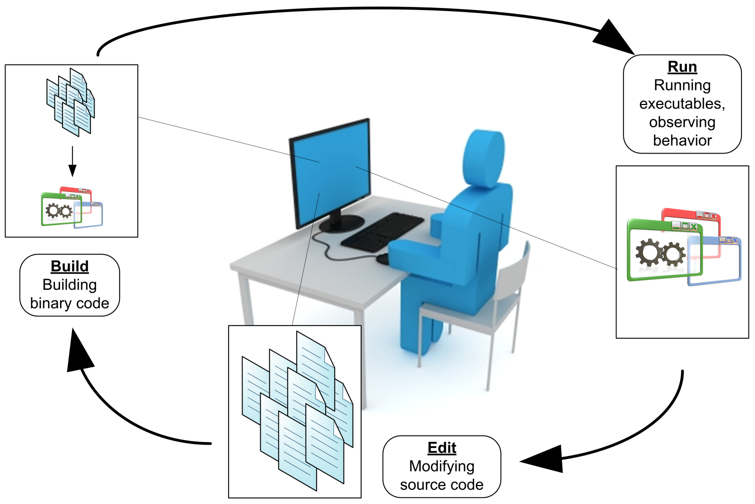


Figure 4.1: In the *edit-build-run* cycle, a developer modifies source code, builds executables, and checks whether the system behaves as expected.

Integration into existing maintenance processes. Performing a maintenance task generally means running through an *edit-build-run* cycle, i.e., modifying source code, building executable code, running the system, and observing its behavior (Figure 4.1). Integrating trace visualization into the edit-build-run cycle must not slow down the overall performance of the developer when performing the cycle. Hence, important requirements for the tracing technique include:

- The runtime overhead with disabled tracing should be minimal. "Disabled" means that all preparations, e.g., compile-time instrumentation, have already been applied, however, no information is captured. A noticeable general decrease in system performance is likely to be perceived as disturbing by developers and will not appeal to them.

- The tracing technique should be compatible with *the* state-of-the-art tools for understanding behavior: debuggers. Hence, it should be possible to seamlessly start and stop tracing from within a debugging session: Using a standard debugger, developers go step by step through execution. At each call statement they decide whether to follow the function call or to step over it. It is often difficult to choose between the two actions. On the one hand, following each call is highly time consuming, and on the other hand, a developer who steps over a call runs the risk of skipping parts of the execution that are relevant to the given maintenance task. With a tracing technique that can be started within a debugging session one can implement a "trace over" functionality that traces indirectly triggered function calls while stepping over a single call.
  Seen from the trace visualization perspective, using a debugger to first navigate to the interesting parts of the execution before activating tracing, is an effective way of coping with the scalability problem of trace visualization: Traces only capture the behavior that is triggered by a single call. Hence, traces are by several orders of magnitude smaller than if tracing is started and stopped on the "freely" running system.

- Enabling and disabling the tracing technique must be fast. Developers are not likely to apply trace visualization, if they have to wait for several minutes or even a complete rebuild of the system before they can take advantage of visualization. If tracing is to be applicable from within a debugging session, then the tracing technique must represent a technique that performs execution-time binary code instrumentation.

Adjustable tracing granularity. To reduce the often large size of a trace right from the start, developers need to be able to adjust the granularity level of trace data according to the given maintenance task [85]. This means, they should be able to choose between tracing only selected functions (low runtime overhead and less detail) and tracing most functions (higher overhead, but more detail).

- If developers are in possesion of a-priori knowledge of specific functions or whole modules that are not relevant to the given maintenance task, the respective functions should be excluded from the tracing technique.

- Additionally, there should be an automated technique that identifies functions called with high frequency during tracing and excludes them automatically.

High degree of applicability and robustness. Given the trace visualization process, applying the tracing technique is the only step that is conceptually platform and processor dependent. The tracing technique, therefore, limits the range of applicability of a trace visualization tool. Hence, it is advantageous to use concepts for the tracing technique that are implementable on a wide range of platforms. Furthermore, one advantage should be taken of already available common instrumentation techniques with a view to achieving a high degree of robustness [184].

Compatibility with state tracing techniques. So far, we have limited the scope to understanding system behavior by means of analyzing control flow, which is considered to be *the* fundamental comprehension task [169]. However, understanding system behavior goes further. Mental models of system state and state changes are built on top of a mental model of control flow. Hence, an additional requirement on the tracing technique arises, if one gets beyond the scope of control flow: The technique should either be able to allow the gathering of state information or it should prove operational in combination with another technique that focuses on gathering state information.

It would be interesting, for instance, to be able to trace function calls while the system is under the control of a scriptable debugger that logs accesses of task relevant variables, or, more precisely, the respective memory locations. The combined captured data would give developers a good overview as to when and how the values are read and written. In a debugging scenario, for instance, this helps a developer to understand the contexts in which erroneous values are written.

4.2 Robust Execution-Time Instrumentation by Reverting Compile-Time Instrumentation

As mentioned in the requirements section, starting and stopping the tracing technique from within a debugging session has the advantage of creating smaller and more precise traces and achieves better integration in the edit-build-run cycle. However, this requires execution-time instrumentation. Existing techniques use code splicing for this[1], which means, new memory is allocated for additional code and original code is partly relocated. The implementation of code splicing is platform dependend. Implementing it in a robust way is a challenging task [167].

The proposed tracing technique is based on a hybrid instrumentation approach, i.e., it exploits the robustness of compile-time instrumentation provided by most compilers and the ease of replacing binary code at execution-time by using a standard debugger.

Trace visualization becomes part of the *run* step of the edit-build-run cycle:

1. Edit: The developer modifies source code as usual.

2. Build: The build step is slightly modified by means of adding global compiler and linker options, and a post-build step. These modification have the following effects:

    - Compiler-supported function entry-point instrumentation is performed.

1 Other techniques avoid code splicing by trap insertion. However, these techniques on the whole lead to a much higher performance overhead due to costly trap handling mechanisms [83].

- Compiler-supported instrumentation is reverted afterwards.

3. Run: The developer runs the system as usual to check its externally visible behavior. If unexpected behavior is experienced, the developer tries to link the behavior with the code—generally done by means of using a debugger. This is where trace visualization comes in:
   - Functions are activated for tracing by using standard debugger facilities.
   - The system runs until a developer-defined execution point is reached.
   - After recording a trace, the developer explores what happened during runtime by analyzing the visualized trace.

4.2.1 Compiler-Supported Function Entry-Point Instrumentation

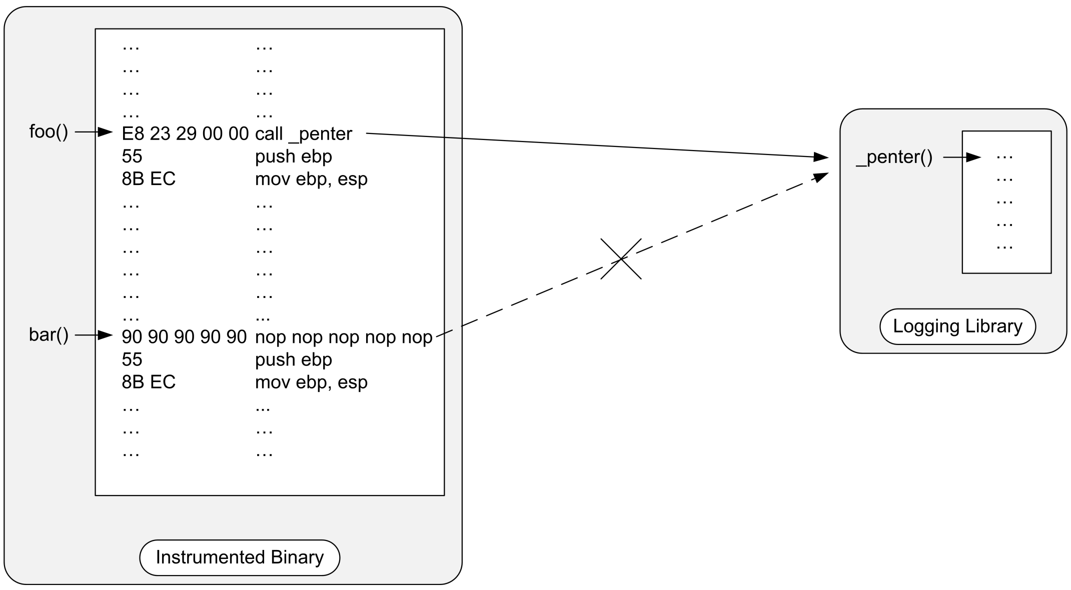


Figure 4.2: With function entry-point instrumentation, the compiler inserts a call instruction at the beginning of each function to redirect control to a hook function. Replacing the call by `NOP` instructions deactivates control redirection.

A wide range of compilers provide options for instrumenting the entry-point of functions. With this, an assembler call instruction is placed at the beginning of the binary code of each function. The inserted calls redirect control to a hook function. For building a tracing tool, one only needs to provide the hook function's implementation (Figure 4.2). The *Microsoft Visual Studio Compilers* (from version *6.0* to version *2010*), for instance, provide the `Gh` compiler option for inserting hook function calls [139]; the *GNU Compiler Collection GCC* provides the option `finstrument-functions` [98]. Many compilers also provide options for instrumenting function exit-points. The proposed tracing technique only uses function entry-point instrumentation, however. Instead of exit-point instrumentation, *return address rewriting* as described by Brown [22] is applied. Limiting the technique to function entry-point instrumentation increases its applicability because not many compilers provide support for exit-point instrumentation—the *Microsoft Visual Studio 6.0 Compiler* is an example.

Building a tracing technique based on common compiler features has several advantages:

- Instrumenting function entry-points does not increase compile time significantly because it is part of the usual compilation process anyway.
- For the same reason, the instrumentation is robust and does not interfer with code optimizations.
- It is easily integrated into existing build processes. Only global compiler and linker options need to be set, which can generally be done without any knowledge of the internals of the build process.

4.2.2 Reverting Compile-Time Instrumentation

After building the fully instrumented binary code, all additionally inserted assembler call instructions are removed and are replaced with assembler `NOP` instructions. The replacement reverts the effects of compiler-based function instrumentation. The resulting binary file does not create any tracing events at runtime and behaves as usual. Only a slight performance decrease ($< 2\%$) is noticeable due to the additional `NOP` instructions (cf Chapter 8).

The address locations of the instructions to be replaced can be obtained via the binary's debug information, which provide the entry-points of all functions contained in the binary. During replacement, the call instructions are stored in a database, where they remain until needed to restore the calls later when reactivating the tracing.

4.2.3 Tracing by Execution-Time Instrumentation

During the period when the developer runs the software system—either "freely" or by means of stepping through its execution with a debugger—tracing may be activated at any time. Throughout this phase, the software system's process is brought under control of a debugger (if it has not already been brought under control) and all the threads are suspended. Then, for all those functions of interest, standard memory rewriting facilities of the debugger are used to put the original assembler call instructions from compile-time instrumentation in place again. During instruction replacement, one needs to check that none of the process' threads points with its instruction pointer to an `NOP` that is about to be replaced. If this is the case, the respective thread is stepped forward with the debugger until the instruction pointer has left the `NOP`s.

Unsuspending the threads causes the system to continue executing. Now, call entries and exits of the activated functions are captured during execution. The developer may stop tracing at any time—either by breaking into the software system's process with the debugger or by setting a breakpoint at a code location by indicating where tracing should be stopped if the process runs under debugger control. When execution is halted, the debugger is used to put `NOP`s in place again to deactivate tracing.

Next, the captured trace is visualized. The developer explores which functions were executed and how they interacted. With this knowledge of task relevant code locations, the developer either continues with the edit-build-run cycle and modifies code; alternatively, well-placed breakpoints are set and the developer uses the debugger to get a fine-grained understanding of system behavior.

4.3 Automatically Detecting and Excluding Massively Called Functions during Tracing

Traces are typically large and frequently consist of hundreds of thousands of calls—even where the developer restricts the functions to be traced by excluding modules. Analysis on how each function in the trace contributes to the size of the trace have confirmed that as a rule only a small number of functions are responsible for a large fraction of the calls. The reason for this unequal call distribution is that traces reflect the execution of nested control loop structures in the code. In the inner loops, fast low-level functions are repeatedly called, which causes the trace to grow rapidly. We use the term *massively called functions* for these functions.

Definition 18 (Massively Called Function) A massively called function is a function whose call frequency in a time window of a developer-defined length exceeds a developer-defined threshold.

Massively called functions typically implement low-level functionality in the software system. For example, a trace capturing the execution of a *resizing* feature of an image processing application are likely to contain massively called functions performing low-level operations on color values. In most cases, massively called functions can be ignored in captured trace data. The definition as to which functions are classified as massively called functions depends on a threshold value set by the developer according to the given maintenance task.

There is, however, one limitation with this approach. We assume that a function that at one stage ascertained to be a massively called function implements only the functionality that this classification is based on. This assumption loses its validity in connection with weak modularized functions that implement, for example in a large switch case, a set of different functionalities, some of them representing lower-level and some higher-level system functionality. The technique would probably exclude the function from the trace by mistake. However, although the execution of erroneously classified massively called functions are not contained in the trace, the subcalls that are triggered by the "invisible" call are still contained in the trace.

In order to measure the call frequency of functions during tracing-time and to compare the frequency value ν against a threshold ν_{max}, functions with frequencies $\nu \geq \nu_{max}$ are classified as massively called functions and are not traced further in the execution. With this technique, the size of a trace is automatically reduced by several orders of magnitude without the availability of any pre-existing knowledge of the system's implementation and without loosing information on higher-level system behavior. We conserve information on high-level system behavior, because only those functions are excluded that are executed in the inner most control loops.

For an example of this consider the *transform* functionality of a 3D content management system. Listing 4.1 shows a simple implementation of the *transform* functionality: The "high-level" function `transform3DScene()` delegates work to the "mid-level" function `transformSceneObj()`, which in turn calls a "lower-level" function `transformVertex()`. Within this function, the "lowest-level" function `Vertex::setPos()` is used to update a vertex' newly calculated position.

```
void transform3DScene(Scene scene, Vector center, double factor)
{
  SceneObjs objs = scene.getObjs();
  foreach (obj in objs) {
    transformSceneObj(obj, center, factor);
  }
}

void transformSceneObj(SceneObj obj, Vector center, double factor)
{
  Vertices vertices = obj.getVertices();
  foreach (vertex in vertices) {
    transformVertex(vertex, center, factor);
  }
}

void transformVertex(Vertex v, Vector center, double factor)
{
  Vector pos = v.getVector();

  double newXPos = ... // calculate new x Position
  double newYPos = ... // calculate new y Position
  double newZPos = ... // calculate new z Position

  v.setPos(0, newXPos);
  v.setPos(1, newYPos);
  v.setPos(2, newZPos);
}
```

Listing 4.1: Code example to illustrate identification of massively called functions.

In a trace that captures the execution of `transform3DScene()`, the functions mentioned in the example are called with different frequencies. Table 4.1 discusses the call frequencies and the call counts—given that the `Scene` object contains 350 `SceneObj` objects each being built of 5000 `Vertex` objects in average. Setting ν_{max} to a value between the "mid" and the "high" frequency classification, for instance, removes all calls to `Vertex::setPos()`, `Vertex::getVector()`, and `transformVertex()`. This reduces the number of calls in the trace from 8.750.702 to 702 and retains the

Call Frequency ν	Call Count	Function
low	1	`transform3DScene()`
low	1	`Scene::getObjects()`
mid	350	`transformSceneObj()`
mid	350	`SceneObj::getVertices()`
high	1.750.000	`transformVertex()`
high	1.750.000	`Vertex::getVector()`
very high	5.250.000	`Vertex::setPos()`

Table 4.1: Call counts and frequencies of the functions mentioned in the code example.

information as to how the "high-level" function `transform3DScene()` applies the transformation to each of the objects contained in the scene.

A complete code example would in addition contain functions, such as the trigonometric functions `sin()` and `cos()`. These functions would be called at least as frequently as `transformVertex()` and would, therefore, also be classified as being massively called functions.

As the threshold ν_{max} is chosen by the developers, they may choose the tracing granularity level that is suitable for the given maintenance task. On the one hand, they may choose a very low threshold value, if they need to analyze system behavior on a coarse-grained level over a long execution time—for instance if they perform a feature location task. On the other hand, a very high threshold value may be chosen for capturing many details in the trace—for instance, if a debugging task is being performed.

A positive effect of the automated deactivation of massively called functions is—besides the reduced size of the trace—that the runtime overhead introduced by the tracing technique is also reduced. Capturing function calls means executing additional binary code for event registration and serialization. Even if this code is highly optimized with regard to performance, tracing still leads to a noticeable performance decrease. Due to the unequal distribution of calls of massively called functions, the lion's share of tracing overhead is a consequence of executing these functions. Hence, if tracing is deactivated for these functions, the absolute performance overhead is drastically reduced (cf Chapter 8), making it possible to apply the tracing technique even on a deployed system running in a production environment.

In Chapter 8, various examples show how the proposed concept applies to large industrially developed software systems. To give an impression of which functions are found as being massively called in a real-world scenario, Table 4.2 lists results from applying the concept to the *Blender* [7] software system comprising 460.000 lines of C code. As can be seen, functions with high call frequencies are functions implementing lowest-level functionality. On the contrary, functions such as those contained in the GUI layer representing entry points for *Blender*'s features have low call frequencies.

Call Frequency ν	Function
3936	add_v3_v3()
3330	_CTX_data_equals()
2012	_EM_remove_selection()
2012	_CustomData_from_em_block()
1968	add_v3_v3()
1583	_cent_quad_v3()
1564	edge_normal_compare()
1512	_findedgelist()
1512	_addedgelist()
1383	normalize_v3()
1383	normalize_v3_v3()
1383	mul_v3_v3fl()
1383	dot_v3v3()
1303	_normal_quad_v3()
1114	calloc_em()
1014	normalize_v3()
1014	normalize_v3_v3()
1014	mul_v3_v3fl()
1014	dot_v3v3()
1007	_CustomData_em_free_block()
1005	_free_editedge()
...	...
...	...
...	...
4	_ui_handle_menu_event()
4	ui_handler_region_menu()
4	ui_item_local_sublayout()
4	ui_mouse_motion_towards_check()
4	_unit_m4()
4	ui_mouse_motion_towards_init()
4	len_v2v2()
4	ui_but_find_mouse_over()
4	ui_handle_button_event()

Table 4.2: Call frequency values of functions of the 460kLOC *Blender* software system. With a threshold value $\nu_{max} = 100$, for instance, low-level functions are excluded from tracing. Higher-level functions such as GUI menu entry points are still contained in the trace, however. The frequency value is given with regard to a time window of 100 million processor ticks. (Functions with the same names are overloaded versions.)

4.3.1 Event Buffer Management

With the intention of clarifying to which data structures the algorithm for identifying massively called functions is applied, this section briefly explains how event registration and serialization concepts are implemented. Function entry and exit events are registered by redirecting control to a hook function. The hook function is part of a logging library that works as follows (Figure 4.3):

- When the library is loaded at runtime, it spawns an additional thread: the *coordinator* thread.

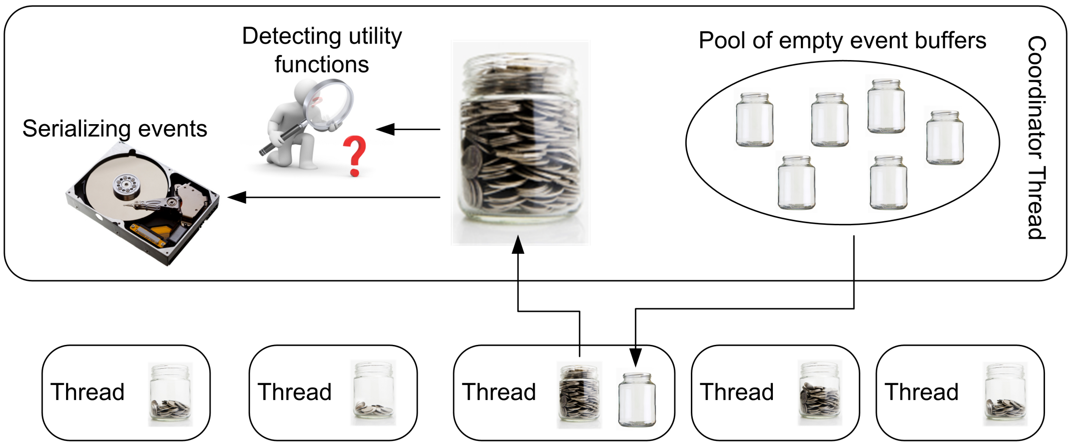


Figure 4.3: The library for registering and serializing function entry/exit events. An additional *coordinator thread* handles buffers of function entry/exit events. Filled buffers are analyzed for massively called functions and serialized to hard disk.

- The coordinator thread creates and manages a pool of empty event buffers.
- Each system process' thread (except for the coordinator thread) has a dedicated event buffer where function entry and exit events are stored when the thread's control is redirected into the logging library.
- If an event buffer is full, the respective thread takes an empty buffer from the pool. The coordinator thread is responsible for serializing the events in the filled buffer to the hard disk.

An *event buffer* sequentially stores *events*, each containing (at least) the following information:

- A flag indicating if it is a function entry or exit event.
- The start address of the respective function in memory space.
- A time stamp.

The events are stored in chronological order in the buffer.

4.3.2 Detecting Massively Called Functions

For detecting and deactivating massively called functions at runtime, each event buffer is analyzed before its events are serialized to disk. The algorithm for detecting massively called functions in the buffer is parameterized by a frequency threshold value ν_{max} and a time window Δt. It operates on a given buffer as follows:

- Let t_{cur} be the time stamp of the first event in the buffer.
  Let i be the iterator pointing to first event in the buffer.
  Let i_{end} be the iterator pointing to the last event in the buffer.

```
Let $AddrCountMap$ be a map of function addresses and counters $\in \mathbb{N}$.
Let $AddrSet$ be an initially empty set of function addresses.

• While $i$ is not equal $i_{end}$ do:
    1. Clear $AddrCountMap$.
    2. Increment $i$ until an event is found with a time stamp $t > t_{cur} + \Delta t$ or
       until the last event in the buffer is reached. While incrementing $i$ do:
        – If the event that $i$ points to is an *exit* event do:
            ∗ Register the event's function address in $AddrCountMap$ with
              count 0, if the address has not been registered yet.
            ∗ Increment the respective count value by 1.
    3. $t_{cur}$ = time stamp of the event that $i$ points to.
    4. For each function address *addr* in $AddrCountMap$ do:
        – If *addr*'s respective count value $\nu > \nu_{max}$ do:
            ∗ Insert *addr* into $AddrSet$.
```

The algorithm having been applied, $AddrSet$ contains the addresses of the functions that are classified as being massively called functions.

4.3.3 Deactivating Massively Called Functions at Execution-Time

To deactivate tracing for massively called functions, the functions' assembler call instructions that redirect control to the hook function needs to removed. The elements in *AddrSet* contain precisely those addresses in memory space where the call instruction is located. Hence, one only needs to overwrite the call instruction with `NOP` instructions.

One possibility of performing the binary code modification is by suspending all threads from within the coordinator thread, putting the `NOP`s in place and unsuspeding the threads again. Suspending the threads is necessary, as replacing the call instructions with multiple `NOP`s is not an atomic operation. Hence, a thread might run into a situation where only some of the `NOP`s are set, causing the application to crash. Another way of performing the binary code modification is to execute an `int3` instruction and allow an attached debugger to handle this *trap*. The debugger then performs the code modification "from outside".

Due to the construction of the algorithm for detecting massively called functions, not all massively called functions can be identified immediately. Rather some can be identified after analyzing several buffers that contain their events. Hence, a post processing operation is applied to the resulting trace that removes the events of all massively called functions.

CHAPTER 5

Pruned Traces - Splitting Traces into Phases

In complex software systems, capturing just a few minutes of behavior commonly results in several hundred million calls. Due to the size of the trace, it is difficult to browse through or view trace data, and hence it is time consuming to identify parts of the trace that are relevant to a given task. In a top-down approach for trace exploration, we start from the root call and navigate step-wise to subcalls deep within the call tree. As calls are nested in time, developers are able to reach more detailed parts of a trace, thereby spanning smaller time intervals. For large traces, performing step-wise navigation is tedious and demands more effective communication and visualization strategies. A key element for their implementation entails reducing trace data to relevant aspects. For this, we introduce a novel concept involving the pruning of traces, which facilitates exploring traces in coarse-grained navigation steps [15]. *Pruning traces* is a technique aimed at reducing a trace to those calls that represent crucial decision points during top-down trace exploration where a developer has to decide which path to follow in the call tree. The technique is based on empirical observations made when developers needed to navigate in large traces taken from event-loop based desktop applications.

The elements of a pruned trace split the original trace into coarser-grained parts, which facilitates rapid exploration and navigation in large traces using a top-down approach. The elements of a pruned trace are called *phases*. The terminology *phase* emphasizes the heavyweight and high-level character of the elements in the pruned trace: Each phase corresponds to a call that either triggers a large amount of indirect calls or that has a long call duration. For a developer exploring a trace, a phase therefore represents a time interval during which coherent system behavior is captured—coherent with regard to the purpose of the function that is executed. That means, it represents a time interval where the system executes a higher-level call that triggers a bunch of subcalls to achieve a higher-level functional goal. With respect to "*caller-called relationships*" Saleh [185] describes that high-level functions are "*typically user interface-driven control modules that are more application-specific and therefore of low reusability*" whereas low-level functions are "*most likely to be reused from or in other software designs*".

Our use of the term *phase* differs from its use in the field of program optimization where the characterization of the system's behavior is driven by resource usage. Madison and Batson [127] describe the challenges of this research field as "*to determine those intervals or phases of the program's execution history that are of significant duration, involve references to a relatively small subset of the information set, and are in some sense 'distinctive' compared to neighboring phases*".

The algorithm for pruning traces falls into the category of data mining. In other words, it is a technique for semiautomatically discovering patterns in the vast amount of data that are meaningful and lead to some advantage—as Witten and Frank [230] define *data mining*.

The key ideas of the concept for pruning traces include:

- Calls are classified according to their subcall and costs characteristics. Two specific classes of calls are in particular identified within a trace. Such calls are referred to as *leaf phases* and *inner phases*.

- Pruning a trace from calls that do not correspond to phases results in a *pruned trace*. A pruned trace provides a valuable hierarchical overview and a recursive high-level description of system behavior. With the help of a pruned trace, developers can navigate with coarser-grained steps than in the original trace.

- A phase similarity metrics permits determining similarities among phases in a configurable way. On this basis, compact trace visualizations can be generated that support developers in identifying outliers in a series of repetitive subphases—When trying to solve a maintenance task, it is often the outlier behavior that is of interest for a developer.

The granularity in which a trace is best partitioned into phases strongly depends on the individual software system being analyzed and on the given maintenance task. Hence, developers adjust the algorithm's parameters accordingly.

5.1 Classifying Function Calls

The trace pruning algorithm is based on observations we made in an experiment where developers had to navigate within large traces. The analyzed traces were taken from applications of a specific domain, namely, the applications were event-loop based desktop GUI applications. During trace analysis, the developers were given an interactive visualization system depicting the detailed sequence of function calls and they were permitted to navigate from call to subcall. The visualization allowed developers to rapidly assess a call's execution costs and the number of subcalls it triggers. The developers' task was to locate those parts of the trace that correspond to the execution of a given functionality of the software system. Hence, they had to solve a feature location task.

An important observation made during the experiment was that some function calls were considered more and others less significant for choosing the subtree of calls to follow when navigating from high-level calls to feature-relevant calls that were located much deeper within the call tree. For each intermediate call during navigation, the developers needed to assess the call's purpose and to decide whether it leads to feature-relevant parts of the trace.

When the developers were following calls that triggered only a few subcalls, they frequently ran into "dead ends". In such a case, they needed to "climb up" in the call tree again and tried another subcall to follow. These *lightweight calls* leading to "dead ends" can algorithmically be identified by comparing the number of triggered calls and the call's costs against two user-defined thresholds: $T_{n_{trig}}$ and T_{costs}[1].

- $T_{n_{trig}}$: Threshold value for the number of triggered calls.
- T_{costs}: Threshold value for the call costs.

Definition 19 (Lightweight Call) A lightweight call is a call c with $|triggeredcalls(c)| < T_{n_{trig}}$ and $costs(c) < T_{costs}$.

Figure 5.1 illustrates where *lightweight calls* typically appear within a trace. Time is depicted along the horizontal axis. Each bar corresponds to a function call and the arrows between the bars indicate when control passes from one call to another.

Another type of calls that could be considered as "dead ends" during navigation were, however, considered to be interesting during exploration by the developers. Such a call had (1) either a large number of triggered calls whereas none of its subcalls had themselves many triggered calls or (2) it had a low number of triggered calls but it had relatively large execution costs. In the latter case, the developers frequently switched to the source code when encountering such a call with large costs. Reading code then often turned out that the respective function either performed time-consuming calculations or it triggered calls that had been excluded from tracing. We classify such calls as a *phase* in the sense as explained at the beginning of the chapter. In particular, we name these calls *leaf phases* because they are leafs in a pruned trace as will be shown later.

Definition 20 (Leaf Phase) A call c is classified as *leaf phase* if $|triggeredcalls(c)| \geq T_{n_{trig}}$ or if $|triggeredcalls(c)| < T_{n_{trig}}$ and $costs(c) \geq T_{costs}$. Additionally, all subcalls of c must not fulfill these conditions.

A further observation obtained from the trace exploration experiment was that developers were frequently quick in deciding which subcall to follow if encountering a *control delegating call.* The characteristics of such a call is that there is only one (direct) subcall with a significantly large number of triggered calls or a large call duration. When *control delegating calls* were encountered, the developers quickly decided to follow the single outstanding subcall.

1 In the following, we use the mathematical convenience operations as defined in Section 2.4.1.

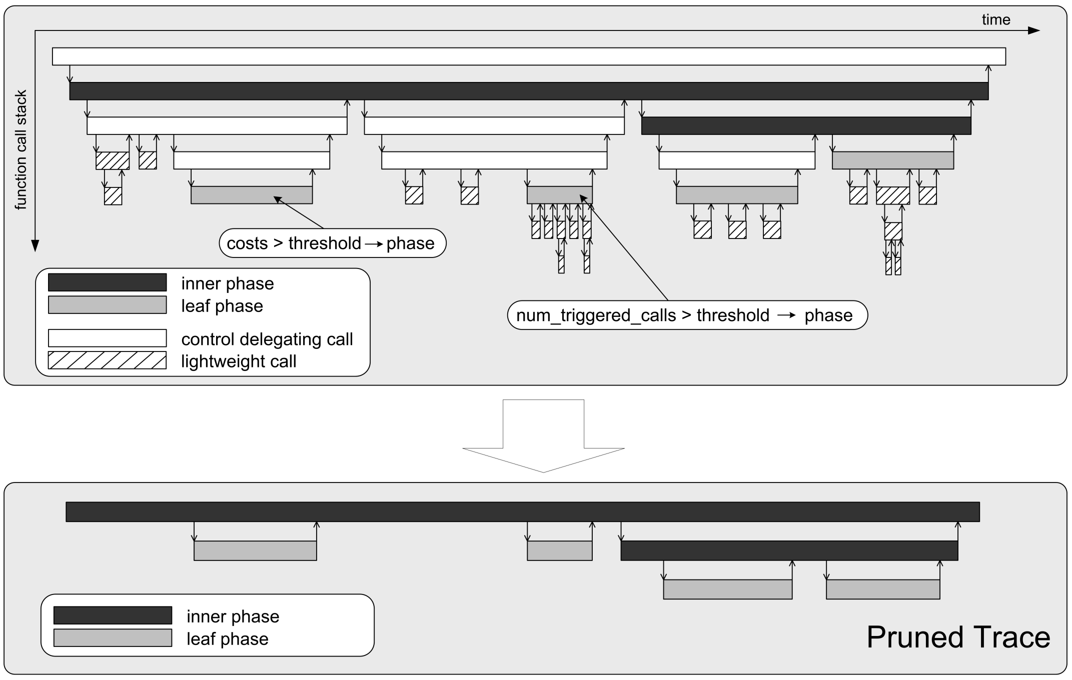


Figure 5.1: Calls captured in a trace can be classified according to their subcall and costs characteristics. Pruning the trace from *lightweight calls* and *delegating calls* allows for a massive reduction in trace size, while retaining important information necessary for top-down trace exploration.

Definition 21 (Control Delegating Call) A call c is classified as *control delegating call* if there is exactly one subcall s of c that has the following characteristics: $|triggeredcalls(s)| \geq T_{n_{trig}}$ or $costs(s) \geq T_{costs}$.

The remaining type of calls was considered to be crucial during navigation as developers needed to decide which control path to follow when reaching such a call. Such calls were similar to *control delegating calls*, however, instead of a single path to follow, developers needed to choose between paths into multiple subtrees comprising many calls or being costly. We name such calls *inner phases*.

Definition 22 (Inner Phase) A call is classified as *inner phase* if it has two or more (direct) subcalls s that fulfill the following criteria: $|triggeredcalls(s)| \geq T_{n_{trig}}$ or $costs(s) \geq T_{costs}$.

As far as we can conclude from the analyzed software systems, *inner phases* seem to be executions of functions that represent important system functionality: To both ensure reuse of system functionality and to enable collaborative development, the system implementation is typically decomposed into modules following the paradigm of information hiding. The functional decomposition forces developers to implement higher-level system functionality by combining and reusing lower-level system functionality[1]. In an image processing application, for instance, the *scale image* functionality is likely to be implemented by executing the *sample pixel* functionality multiple times, which, in turn, executes the *read color* functionality multiple times. *Inner phases* would appear to be the result of this hierarchical way of implementing functionality. An *inner phase* captures the situation that higher-level functionality triggeres multiple times lower-level functionality. Therefore, they seem to be crucial when exploring traces and trying to understand the system behavior that is captured by the trace.

As Figure 5.1 indicates, pruning a trace from all calls except for *phases* reduces the size of the trace drastically.

To compensate for calls missing in the pruned trace, meta-information is attached to the phases and provides statistics on the missing calls. This meta-information permits developers, while exploring a pruned trace, to assess whether the trace pruning algorithm skips calls that are in fact relevant to the given maintenance task. If meta-information indicates that important calls have possibly been skipped, then the developer should analyze the corresponding time range in full detail by means of an appropriate view that depicts the original trace. The meta-information $i \in \mathbb{I} = \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0$ attached to each phase p of a pruned trace is:

- $numcalls_p$: Number of calls that are executed within p's time range.
- $numfuncs_p$: Number of distinctive functions being executed within p's time range.
- $depth_p$: Maximum call depth of all calls contained within p's time range.

Given a trace $T = (F,C)$ (cf mathematical definition of traces in Section 2.4.1), a pruned trace is defined as graph $T^\dagger = (F^\dagger,C^\dagger)$ where $F^\dagger \subseteq F$ is a subset of the functions in the original trace. $C^\dagger$ is a subset of the set of calls C, however, the calls are additionally labeled with meta-information $i \in \mathbb{I}$. To reflect the nested call structure, edges are restricted as defined in Section 2.4.1. The edge labels associated with the calls carry the information on start time, end time and meta-information. As the phases in pruned traces have a nested structure equal to the calls of a trace, the operations *parentphase*, *enclosingphases*, *subphases*, and *triggeredphases* are likewise evenly defined for pruned traces.

For reasons of clarity, the data model of pruned traces is shown in Figure 5.2. The common characteristics of *phases* are captured in a generalization relation with the super class `Phase` and an associated `PhaseInfo` class. Only *inner phases* may have subphases. Each phase is linked to exactly one call of the original trace.

1 Some system functionality is best described in terms of the problem domain (high-level functionality) and some in term of the solution domain (low-level functionality).

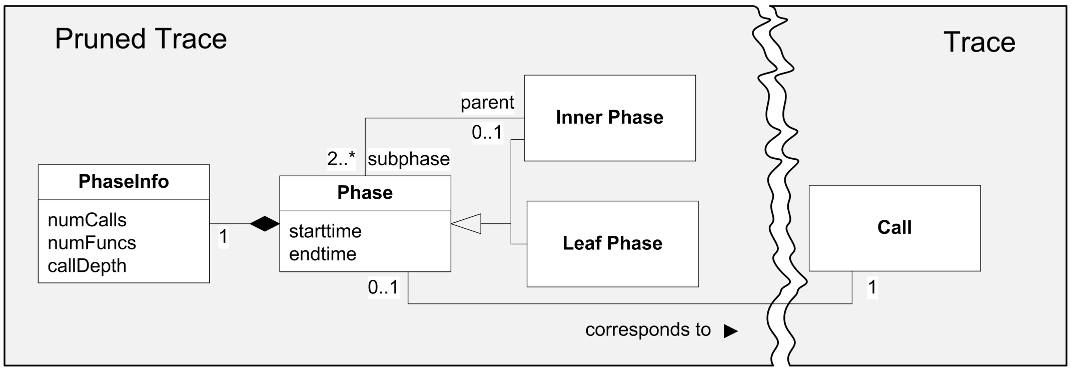


Figure 5.2: Data model of pruned traces.

5.2 The Trace Pruning Algorithm

Based on user-defined thresholds, the trace pruning algorithm automatically identifies calls as *inner phases* or as *leaf phases* within a trace and eliminates all other call types. This significantly reduces trace size. However, the pruned trace still contains valuable call information that gives developers a useful overview of what happens within various sections of the trace.

With the given thresholds $T_{n_{trig}}$ and T_{costs} as defined in the last section, the algorithm calculates a value $n(c) \in \mathbb{N}_0$ indicating the number of subcalls of c that either have a large number of triggered calls or have large execution costs:

- $n(c) = |\{c' \in subcalls(c) : |triggeredcalls(c')| \geq T_{n_{sub}} \vee costs(c') \geq T_{costs}\}|$

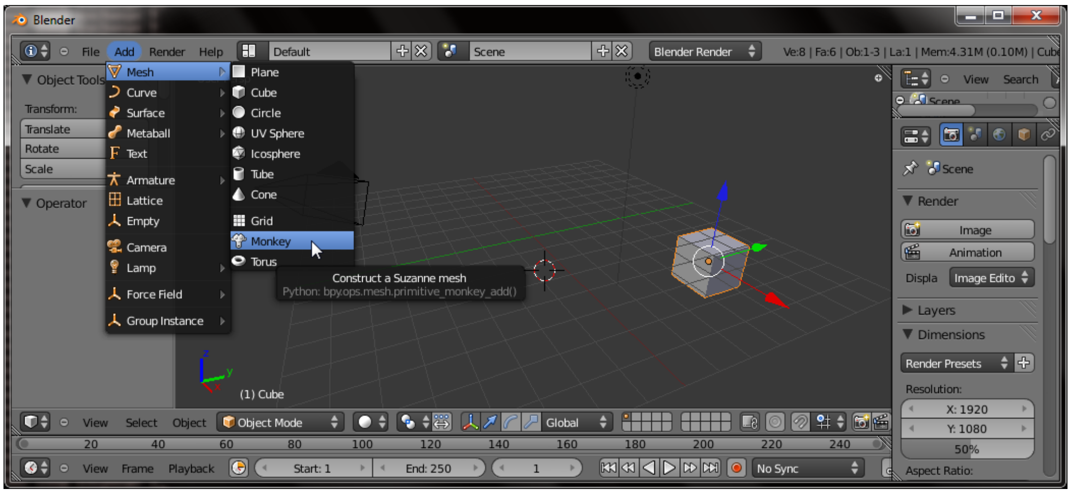
The algorithm uses these definitions to identify *inner phases* and *leaf phases*:

- Call c is an *inner phase* if
  $n(c) \geq 2$.

- Call c is a *leaf phase* if
  $n(c) = 0 \ \wedge \ (|triggeredcalls(c)| \geq T_{n_{trig}} \vee costs(c) \geq T_{costs})$.

To simplify use of the algorithm for any developer, the parameter T_{costs} can be based on the parameter $T_{n_{trig}}$. Hence, a typical short call c_{short} must be identified in the trace. T_{costs} can then be defined as $T_{costs} = costs\,(c_{short}) * T_{n_{trig}}$.

The algorithm parses the sequence of function calls in a single pass, thereby reconstructing the call stack. At the end of each call, the call is checked to ascertain whether it is an *inner phase* or a *leaf phase*. As the algorithm's time complexity is linear, the algorithm is fast enough for developers to recalculate the pruned trace with different threshold values, in the event that the resulting pruned trace is still too large or too reduced—even where the trace is very large. Applying the algorithm on a trace consisting of 100 million calls, for instance, takes an average of 50 seconds with our prototypical implementation on an Intel Core 2 Duo CPU at 2.4GHz. The time aspect can vary, however, due to the operating system's disk caching mechanism.



Construct a Suzanne mesh
Python: bpy.ops.mesh.primitive_monkey_add()

Figure 5.3: Using the *Blender* software system, a *monkey* shape is added to the currently managed 3D content.

To illustrate how a trace can massively be reduced by applying the trace pruning algorithm, we exemplarily analyze the behavior of the *Blender* software system [7] for creating, modeling and rendering 3D content (460.000 lines of C code). A trace is captured while a user adds a *monkey* shape to the current 3D content project (see Figure 5.3). The user's task is to understand which parts of the code are responsible for implementing this feature. The resulting trace comprises 545.276 function calls[1]. Table 5.1 illustrates the characteristics of the trace, i.e., the number of calls per call stack depth.

Applying the trace pruning algorithm with $T_{ntrig} = 5000$ and $T_{costs} = 100$ ticks $* T_{ntrig}$ results in a pruned trace as depicted in Table 5.2. The algorithm found a hierachy of 72 phases within the trace; one of them (`make_prim_ext()` in the source code file `editmesh_add.c`) triggers the creation of the geometric mesh that internally represents the *monkey* shape in *Blender*. Hence, analyzing the pruned trace can permit developers to quickly find important function calls that represent task-relevant parts of the trace. After having identified the call of the `make_prim_ext()` function, the developer needs to explore the complete trace to gather more detailed information on the execution context of this call.

For reasons of clarity, we show in Figure 5.4 visual representations of the trace—the *temporal overview* and the *call stack view* are introduced in Chapter 6. As can be seen, the calls of `wm_method_draw_overlap_all()`, which the trace pruning algorithm identified automatically, correspond to the repetitive visual patterns in the *temporal overview*. Figure 5.5 highlights the visual pattern corresponding to the execution of `make_prim_ext()`. This pattern is also algorithmically found by the trace pruning algorithm.

1 The technique for detection and excluding massively called functions (cf Section 4.3) is used with a threshold $\nu_{max} = 80$ and $\Delta t = 100.000.000$.

Call Stack Depth	Number of Calls	Call Stack Depth	Number of Calls
1	1	19	33496
2	1	20	25017
3	1	21	27461
4	1	22	28662
5	1	23	23430
6	1	24	14631
7	1	25	10581
8	7114	26	4569
9	20881	27	2138
10	37225	28	1497
11	25516	29	712
12	8182	30	267
13	20625	31	178
14	27329	32	147
15	62514	33	48
16	68737	34	56
17	53477	35	54
18	40709	36	12
		37	4

Table 5.1: Characteristics (function calls per call stack depth) of the trace captured during execution of *Blender*'s *add monkey* feature.

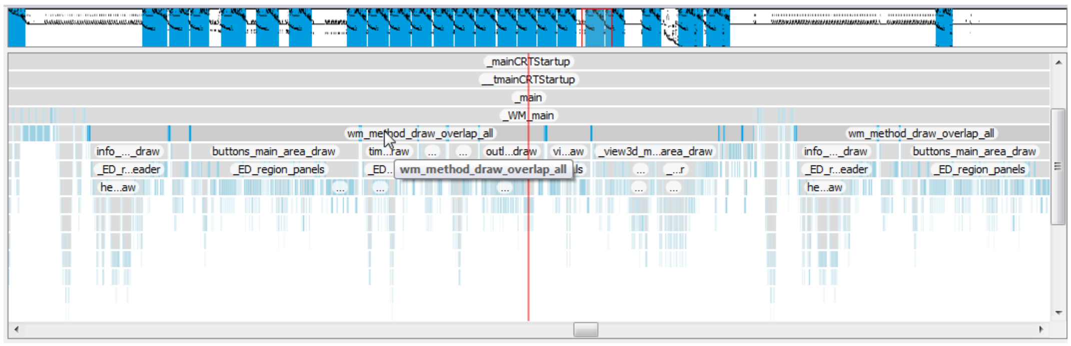


Figure 5.4: The *temporal overview* and the *call stack view* show that the phases `wm_method_draw_overlap_all()`, which were detected by the trace pruning algorithm, correspond to repetitive execution patterns.

Depth	Phasename (Functionname)	Filename	#Calls
1	rootcall	no file	545276
2	_WM_main()	windowmanager/intern/wm.c	545270
3	wm_method_draw_overlap_all()	windowmanager/intern/wm_draw.c	45887
4	_ED_region_header()	editors/screen/area.c	17039
5	bpy_class_call()	python/intern/bpy_rna.c	10651
4	_ED_region_panels()	editors/screen/area.c	18809
3	_wm_event_do_handlers()	windowmanager/intern/wm_event_system.c	5049
3	wm_method_draw_overlap_all()	windowmanager/intern/wm_draw.c	30330
4	_ED_region_header()	editors/screen/area.c	2102
4	_ED_region_panels()	editors/screen/area.c	12914
5	_uiEndPanels()	editors/interface/interface_panel.c	1615
4	_ED_region_header()	editors/screen/area.c	3276
4	_ED_region_header()	editors/screen/area.c	3993
4	_ED_region_panels()	editors/screen/area.c	3846
3	wm_method_draw_overlap_all()	windowmanager/intern/wm_draw.c	25893
4	_ED_region_header()	editors/screen/area.c	2102
4	_ED_region_panels()	editors/screen/area.c	12914
4	_ED_region_header()	editors/screen/area.c	3276
4	_ED_region_header()	editors/screen/area.c	3993
3	wm_method_draw_overlap_all()	windowmanager/intern/wm_draw.c	25899
4	_ED_region_header()	editors/screen/area.c	2108
4	_ED_region_panels()	editors/screen/area.c	12914
4	_ED_region_header()	editors/screen/area.c	3276
4	_ED_region_header()	editors/screen/area.c	3993
3	wm_method_draw_overlap_all()	windowmanager/intern/wm_draw.c	30885
4	_ED_region_header()	editors/screen/area.c	2108
4	_ED_region_panels()	editors/screen/area.c	12914
5	_uiEndPanels()	editors/interface/interface_panel.c	1615
6	_uiDrawBlock()	editors/interface/interface.c	479
4	_ED_region_header()	editors/screen/area.c	3276
4	_ED_region_header()	editors/screen/area.c	3993
4	_ED_region_panels()	editors/screen/area.c	3846
3	wm_method_draw_overlap_all()	windowmanager/intern/wm_draw.c	26866
4	_ED_region_header()	editors/screen/area.c	2108
4	_ED_region_panels()	editors/screen/area.c	12914
4	_ED_region_header()	editors/screen/area.c	3276
4	_ED_region_panels()	editors/screen/area.c	3846
3	wm_method_draw_overlap_all()	windowmanager/intern/wm_draw.c	27289
4	_ED_region_header()	editors/screen/area.c	2108
4	_ED_region_panels()	editors/screen/area.c	12914
4	_ED_region_header()	editors/screen/area.c	3276
4	_ED_region_panels()	editors/screen/area.c	3846
3	wm_method_draw_overlap_all()	windowmanager/intern/wm_draw.c	9852
4	_ED_region_header()	editors/screen/area.c	1174
3	wm_method_draw_overlap_all()	windowmanager/intern/wm_draw.c	9852
4	_ED_region_header()	editors/screen/area.c	1174
3	wm_method_draw_overlap_all()	windowmanager/intern/wm_draw.c	9852
4	_ED_region_header()	editors/screen/area.c	1174
3	wm_method_draw_overlap_all()	windowmanager/intern/wm_draw.c	9852
4	_ED_region_header()	editors/screen/area.c	1174
4	_ED_region_panels()	editors/screen/area.c	1217
3	wm_method_draw_overlap_all()	windowmanager/intern/wm_draw.c	9852
3	wm_method_draw_overlap_all()	windowmanager/intern/wm_draw.c	9852
3	wm_method_draw_overlap_all()	windowmanager/intern/wm_draw.c	9634
3	wm_method_draw_overlap_all()	windowmanager/intern/wm_draw.c	6339
3	wm_method_draw_overlap_all()	windowmanager/intern/wm_draw.c	6339
3	wm_method_draw_overlap_all()	windowmanager/intern/wm_draw.c	6339
3	wm_method_draw_overlap_all()	windowmanager/intern/wm_draw.c	6339
3	wm_method_draw_overlap_all()	windowmanager/intern/wm_draw.c	6339
3	wm_method_draw_overlap_all()	windowmanager/intern/wm_draw.c	6339
3	wm_method_draw_overlap_all()	windowmanager/intern/wm_draw.c	6339
3	_wm_operator_invoke()	windowmanager/intern/wm_event_system.c	52632
4	make_prim_ext()	editors/mesh/editmesh_add.c	51146
5	make_prim()	editors/mesh/editmesh_add.c	20492
5	_ED_object_exit_editmode()	editors/object/object_edit.c	30338
6	_load_editMesh()	editors/mesh/editmesh.c	25246
6	_free_editMesh()	editors/mesh/editmesh.c	5041
3	wm_method_draw_overlap_all()	windowmanager/intern/wm_draw.c	9026
4	_view3d_main_area_draw()	editors/space_view3d/view3d_draw.c	3044
3	wm_method_draw_overlap_all()	windowmanager/intern/wm_draw.c	8995
3	wm_method_draw_overlap_all()	windowmanager/intern/wm_draw.c	4761
3	_wm_window_process_events()	windowmanager/intern/wm_window.c	26

Table 5.2: The (tree-shaped) pruned trace representing *Blender*'s behavior when creating a *monkey* shape. The phase `make_prim_ext()` turns out to be representing the execution of the core implementation of the *monkey creation* feature. The table rows are ordered according to the chronological order of the phases' start times.

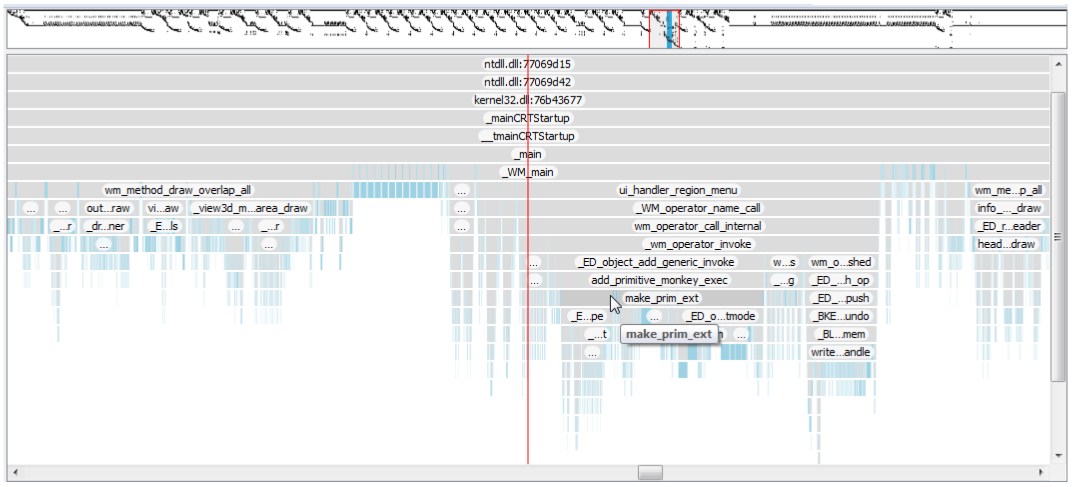


Figure 5.5: The trace pruning algorithm automatically identified the execution phase `make_prim_ext()`. It is responsible for creating the *monkey* shape during *Blender*'s execution.

5.3 Detecting Repetitive Behavior

The pruning algorithm massively reduces the calls contained in the trace. Control loop structures in functions corresponding to inner phases, however, may cause the pruned trace to continue to consist of repetitive structures that make trace exploration difficult (Figure 5.6). The phase corresponding to the *scale image* behavior of the example mentioned in the previous section, for instance, executes the *sample pixel* behavior very often, which results in large sequences of repetitive subphases. To simplify exploration of such repetitive structures, we aim to automatically detect similarity of phases, permitting visualization of the pruned trace in an even more compact way that makes repetitions of similar phases explicitly visible. Furthermore, the similarity detection facilitates identifying outliers in successive subphases. In maintenance task situations, it is often the outlier behavior that needs to be understood by a developer to be able to solve the task. A bug, for instance, may not show up in the regular behavior but in the outlier behavior that corresponds to a special case of system execution for which a correct handling had not been implemented yet.

5.3.1 Phase Similarity

In sequences of repetitive subphases that all pass control to the same callee function, it is often the outlier behavior, i.e., the one that executes in a different manner from the others, that is most interesting for the developer to explore. Being able to seize a "deep" similarity measure algorithmically can create visualizations that support developers in detecting outliers and guide them along these outlier control flows. Hence, we define phases as being *equal*, if they trigger the same sequence of function calls (omitting the time information). Often, however, phases are considered being equal from the developer's point of view even where the call sequences are not identical. Two sequences capturing "string conversion" behavior, for instance, are likely to be considered being equal, even if one string consists of 42 and the other of 15 characters. In order to allow similarity detection in a more "fuzzy" way, a *fingerprint* is assigned to each phase and a similarity metrics is defined based on fingerprints.

5.3.2 Phase Fingerprints

A *fingerprint* of a phase is a bit vector with the same number of dimensions as functions participating in the complete trace. Each dimension's value is 1, provided the corresponding function is active during the phase; otherwise it is 0. Hence, phase fingerprints abstract from (a) repetitive function executions and (b) the order of function executions. Unfortunately, this approach for defining similarity is often still too rigid. With the specific aim of making identifying similarity in an even more fuzzy way possible, we introduce a similarity metrics that operates on fingerprints.

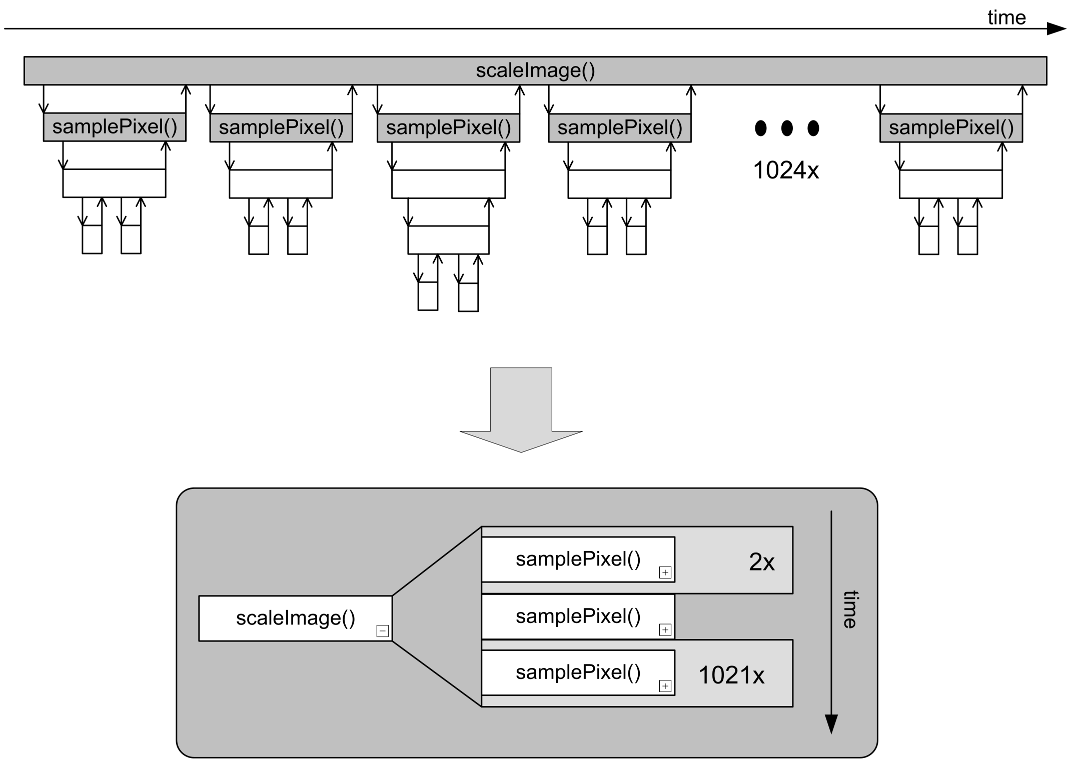


Figure 5.6: Control loop structures in the code may cause highly repetitive structures even in a pruned trace. The phase similarity metrics makes provision for compact visualizations that make repetitions explicit and facilitate outlier detection.

5.3.3 Phase Similarity Metrics

The rationale behind the *phase similarity metrics* is to identify whether two phases (that are equal in terms of caller and callee function) differ only slightly in the set of executed functions. The corresponding system behavior captured by the phases might be considered by developers as being similar. It is not sufficient to define the absolute number of distinctive functions by adopting a metrics value and compare it against a threshold value. If this is done, the decision whether two phases are similar would depend strongly on how many functions are active. Phases executing a small number of functions would be considered being similar to such other phases, even if the set of functions were disjoint. Hence, we build the ratio between the number of distinctive active functions and the total number of executed functions for the two phases p_1 and p_2 to define the phase similarity metrics. The XOR and OR notation refers to a representation of phase fingerprints as bit vectors:

- $sim(p_1, p_2) = \frac{bitCount(XOR(p_1,p_2))}{bitCount(OR(p_1,p_2))}$

If the phases p_1 and p_2 differ in their caller and callee functions, the metrics value is clamped to 1.

Metrics values range from between 0 and 1. $sim\,(p_1, p_2) = 0$ means that p_1 and p_2 execute exactly the same set of functions. $sim\,(p_1, p_2) = 1$ means that the sets of

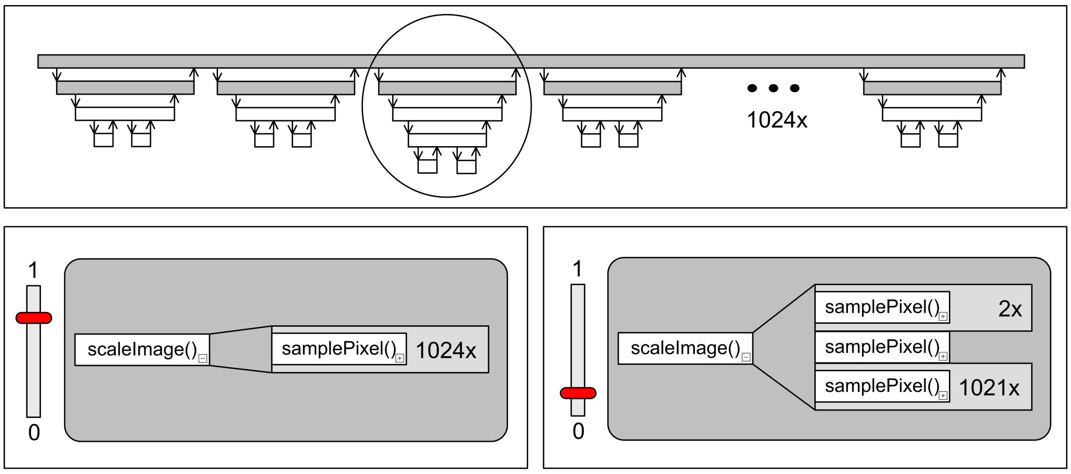


Figure 5.7: Interactively adjusting the similarity metrics threshold permits control as to whether outlier phases are visualized explicitly or not.

functions executed by c_1 and c_2 are disjoined. To classify whether two phases are similar or not, the similarity metrics is compared to a threshold value T_{sim}. Figure 5.7 illustrates how different threshold values permit developers to explore the trace in different granularity levels. With $T_{sim} = 1$ phases leading to the same callee function are considered to be similar, which results in a highly compact visualization. Decreasing T_{sim} then successively reveals differences in the set of functions that are indirectly executed in the phases. In this way developers are able to identify outlier phases in a series of phases. Details of the presentation technique of pruned traces are given in Section 6.2.3. The similarity calculation is a simple operation and is therefore fast enough to interactively change the threshold value T_{sim}, thereby interactively exploring outlier phases.

CHAPTER 6

Visualization Techniques for Traces

This chapter introduces a framework for trace visualization techniques (Figure 6.1). It provides solutions to the question as to how core techniques for viewing trace data can be implemented that developers are supported in performing top-down and bottom-up comprehension strategies. Emphasis is placed on showing how trace data taken from a fact base is filtered, transformed into a geometry model and finally converted into a visual representation. Additionally, it is explained how developers may use different views on a trace simultaneously enabling them to cross-reference findings made in a single view with context information provided by other views.

Depending on the specific maintenance task, developers have different requirements

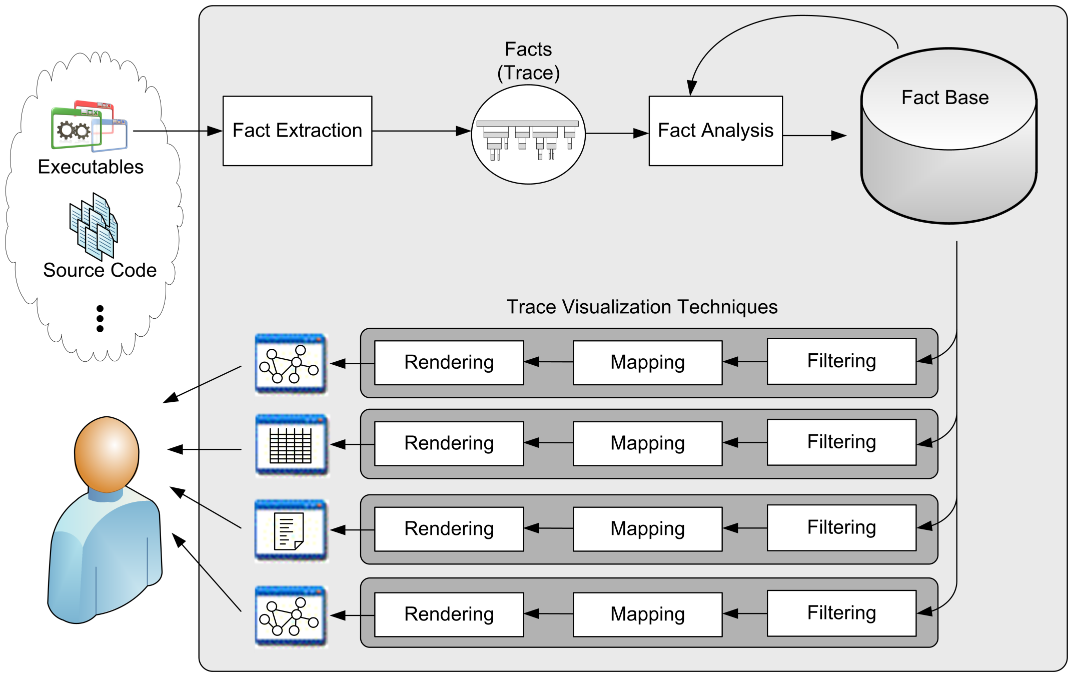


Figure 6.1: Developers are provided with multiple views on a trace by using different visualization techniques.

in respect of visual representations. Facts and fact relations need to be displayed in different ways. When exploring traces (cf Section 2.1.1: e.g., top-down and bottom-up strategies) developers use common comprehension strategies. Therefore, several general trace visualization techniques need to be devised that may prove helpful in a variety of different maintenance task scenarios.

In the context of visualizing software architectures, Clements et al. emphasize the need for multiple views: "*It may be disconcerting that no single view can fully represent an architecture. Additionally, it feels somehow inadequate to see the system only through discrete, multiple views. [...] The essence of architecture is the suppression of information not necessary to the task at hand. [...] This is its strength: Each view emphasizes certain aspects of the system while deemphasizing or ignoring other aspects, all in the interest of making the problem at hand tractable. Nevertheless, no one of these individual views adequately documents the software architecture for the system*" [35]. Favre [60] states that "*large software products are difficult to understand because they are made of many entities of many different types in many concrete representations, usually not designed with software comprehension in mind.*" He emphasizes that developers, hence, need a "*diversity of perspectives on software*".

These statements are particularly true in the case of trace visualization. Developers need to be provided with multiple views on traces that enable them to transfer findings from one view to another. In other words, developers need to be able to cross-reference the various mental models built by analyzing different views.

The visualization techniques can be characterized by answering the following aspects:

- What is the purpose of the view?
- Which comprehension strategies are supported by the view?
- How is the view implemented?
    1. Filtering step
    2. Mapping step
    3. Rendering step: As the rendering step means to apply a standard conversion from the geometry model to an image for all views (cf Section 2.3.3), this step is not explicitly mentioned in the following.
- What are the limitations of the view?
- In what way can the view be combined with other views during trace exploration?

6.1 Mathematical Operations on Traces

Visualization techniques for trace data are based on mathematical operations described in the following.

6.1.1 Trace Filtering Operations

Trace filtering operations take a trace as primary input and result in a trace as output. The operations eliminate calls and functions from the input trace to obtain a smaller output trace.

Time Range Constraint

The *time range constraint* operation $\Psi^{\text{time-range-constraint}} : \mathbb{T} \times \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{T}$ reduces a trace by eliminating calls that do not participate within a given time range (Figure 6.2). It takes a trace $T = (F{,}C)$, a start time t_s, and an end time t_e as input and results in an output trace $T^{out} = (F^{out}, C^{out})$.

The operation is performed in two stages:

1. Reducing the trace to the calls that lie within the given time range: $T' = (F', C')$ with $C' = \{c \in C : \mathit{start}(c) > t_s \text{ and } \mathit{end}(c) < t_e\}$ and $F' = \mathit{funcset}(C')$.
2. The calls that enclose all calls in C' are added to conserve the complete call stack for all calls in T'. $C^{out} = C' \cup \bigcup_{c' \in C'} \mathit{enclosingcalls}(c')$ and $F^{out} = \mathit{funcset}(C^{out})$.

Call Tree Removal

The *call tree removal* operation $\Psi^{\text{call-tree-removal}} : \mathbb{T} \times 2^{\mathbb{C}} \longrightarrow \mathbb{T}$ removes all calls that are triggered by a call out of the given input set C_x (Figure 6.3). The operation takes a trace $T = (F{,}C)$ and a call set $C_x \subset C$ as input and returns an output trace $T^{out} = (F^{out}, C^{out})$.

The operation is performed on a given trace $T = (F{,}C)$ as follows:

- $C^{out} = \{c \in C : \mathit{enclosingcalls}(c) \cap C_x = \emptyset\}$ and $F^{out} = \mathit{funcset}(C^{out})$.

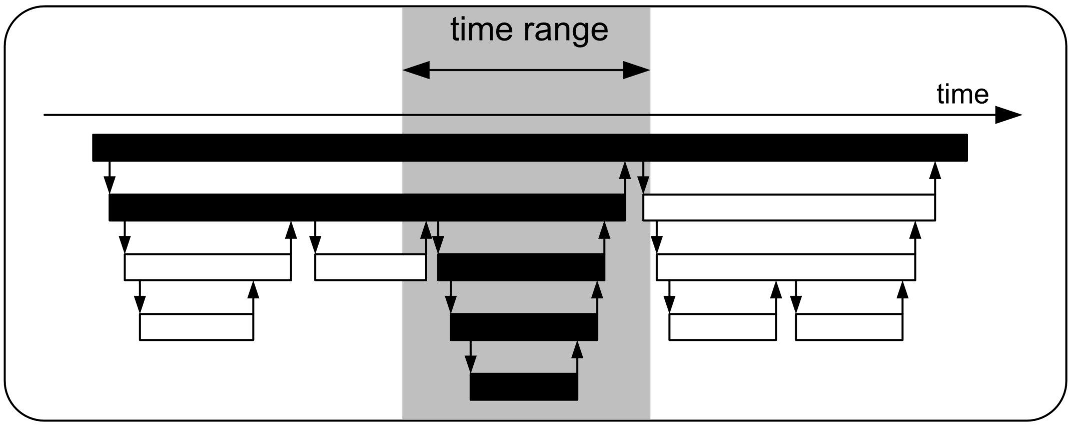


Figure 6.2: The time range constraint operation $\Psi^{\text{time-range-constraint}}$.

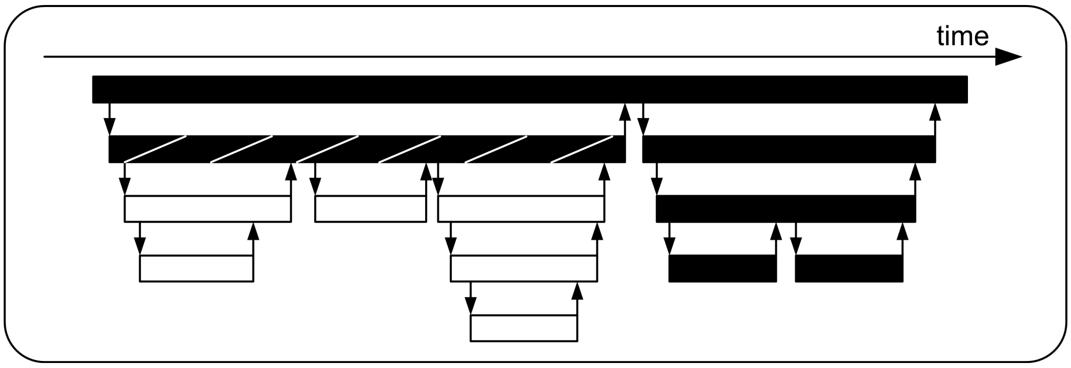


Figure 6.3: The call tree removal operation $\Psi^{\text{call-tree-removal}}$. The call trees triggered by the "striped" call are removed from the trace.

As an alternative to explicitly specifying a call set, one can specify a set of functions $F_x \subset F$ whose direct and indirect outgoing calls are removed from the trace. The call set C_x which is used as input for $\Psi^{\text{call-tree-removal}}$ is then: $C_x = \{c \in C : \mathit{callee}(c) \in F_x\}$.

Module Removal

The module removal operation $\Psi^{\text{module-removal}}$ operates on a trace $T = (F,C)$ and a matching module hierarchy $M = (V,H)$. For any given module $v \in V$, the functions are first determined that are both part of the module and part of the trace and then removed from the trace. The resulting trace $T^{out} = (F^{out}, C^{out})$ can be described as follows:

- $F_x = F \cap (\mathit{descendants}(v) \cup \{v\})$.
- $F^{out} = F \setminus F_x$.
- $C^{out} = \{c \in C : \mathit{callee}(c) \notin F_x\}$.
- In a final step, $\forall c \in C^{out}$ the call relation needs to be recalculated in such a manner that the caller function matches the callee function of c's parent call with regard to the reduced trace T^{out}.

6.1.2 Call Graph Operations

Call graphs are essential building blocks devised to describe traces in a compactified way. In this section, we describe how a trace can be converted into a call graph and how call graphs can be reduced.

Converting a Trace into a Call Graph

A call graph abstracts from the sequential information stored in a trace by aggregating calls from the same caller and callee function to *call relations* (Figure 6.4). Call

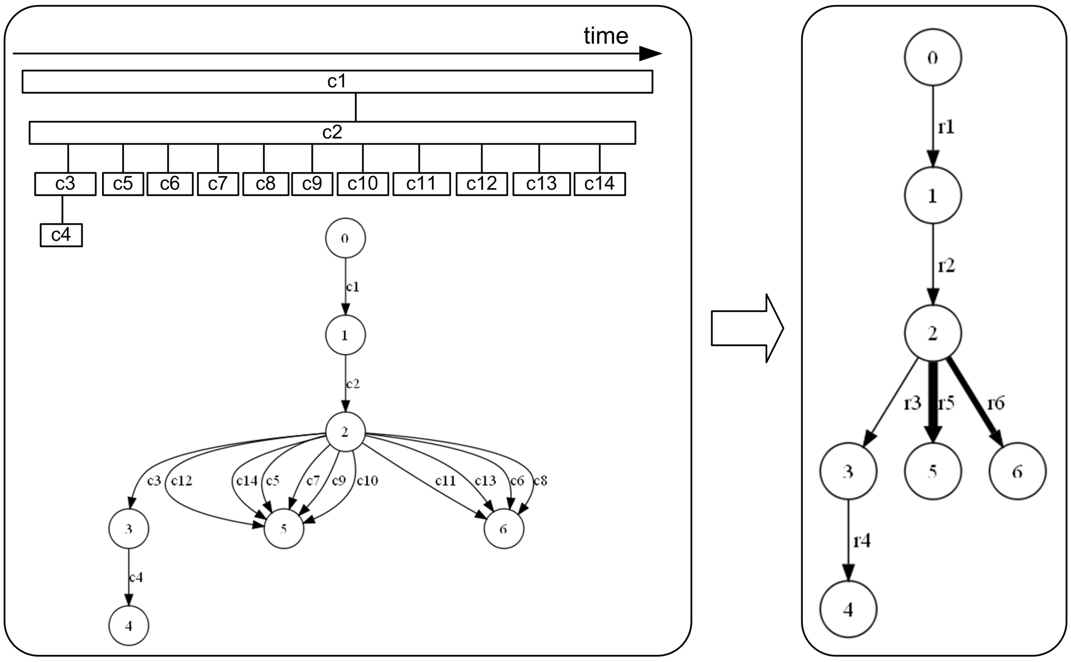


Figure 6.4: A call graph abstracts from the sequential information stored in a trace. Functions are depicted by circles; calls by edges or rectangles, respectively.

graphs permit developers to explore the structural relationships between functions [9, 10, 12].

Let $\mathbb{G}$ be the set of all call graphs. A call graph $G = (V,R) \in \mathbb{G}$ is a graph with the set of modules $V \subset \mathbb{V}$ as nodes and a set of edges $R \subset \mathbb{R} = \mathbb{V} \times \mathbb{D}_R \times \mathbb{V}$ named *call relations*. $\mathbb{D}_R = \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ is a set of labels attached to call relations. They carry information on the *number of calls* that a call relation represents, the aggregated *costs* and the aggregated *selfcosts*. Call graphs are defined on modules rather than on functions only (functions *are* modules, i.e., $\mathbb{F} \subseteq \mathbb{V}$) to make the description of the module collapsing operation clearer that is defined in the next section.

Table 6.1 defines operations on two functions $f,g \in F$ of a given trace $T = (F,C)$ which help to give a concise description of the conversion of traces to call graphs.

The conversion operation Φ takes a trace $T = (F,C)$ as input and results in a call graph $G = (V,R)$ as output. The nodes in G are the same as in T, i.e., $V = F$. R is defined as follows:
$R = \{r \in \mathbb{R} : r = callrel(f,g) \text{ with } f,g \in F \text{ and } callset(f,g) \neq \emptyset\}$

To give a clearer picture of the description of operations on call graphs in the remainder of the chapter, convenience operations are listed in Table 6.2.

Module Collapsing

Module collapsing is a common approach when aiming to reduce the size of a call graph. Call relations between low-level modules, e.g., functions, are aggregated to call relations between higher-level modules. The module collapsing operation Γ operates

Operation	Domain and Range	Result
$callset$	$\mathbb{F} \times \mathbb{F} \rightarrow 2^{\mathbb{C}}$	$\{c \in C : caller(c) = f$ and $callee(c) = g\}$
$count$	$\mathbb{F} \times \mathbb{F} \rightarrow \mathbb{N}$	$\|callset(f,g)\|$
$costs$	$\mathbb{F} \times \mathbb{F} \rightarrow \mathbb{N}$	$\sum\limits_{c \in callset(f,g)} costs(c)$
$selfcosts$	$\mathbb{F} \times \mathbb{F} \rightarrow \mathbb{N}$	$\sum\limits_{c \in callset(f,g)} selfcosts(c)$
$callrel$	$\mathbb{F} \times \mathbb{F} \rightarrow \mathbb{V} \times \mathbb{D}_R \times \mathbb{V}$	(f, d_R, g) with $d_R = (count(f,g), costs(f,g), selfcosts(f,g))$

Table 6.1: Operations on two functions $f,g \in F$ of a given trace $T = (F,C)$ that are used for describing the conversion from traces to call graphs.

Operation	Domain and Range	Result
$source$	$\mathbb{R} \rightarrow \mathbb{V}$	$(v_1,d,v_2) \mapsto v_1$
$destination$	$\mathbb{R} \rightarrow \mathbb{V}$	$(v_1,d,v_2) \mapsto v_2$
$relationlabel$	$\mathbb{V} \times \mathbb{V} \rightarrow \mathbb{D}_R$	For $v_1, v_2 \in R$: returns the label d_R attached to the call relation $r \in R$ with $source(r) = v_1$ and $destination(r) = v_2$ or $d_R = (0,0,0)$ if no such relation exists
$+$	$\mathbb{D}_R \times \mathbb{D}_R \rightarrow \mathbb{D}_R$	$((n_1, c_1, s_1),(n_2, c_2, s_2)) \mapsto (n_1 + n_2, c_1 + c_2, s_1 + s_2)$

Table 6.2: Convenience operations on a call graph $G = (V,R)$.

on a call graph $G = (V,R)$ and a module hierarchy $M = (V_M,H)$ with $V \subseteq V_M$.

Let $V_x \subseteq V_M$ be the set of modules to be collapsed. Then Γ results in a call graph $G^{out} = (V^{out}, R^{out})$ with:

- $V^{out} = \{v \in V : ancestor(v) \cap V_x = \emptyset\} \cup V_x$
- $R^{out} = \{(v_1, d, v_2) \in \mathbb{R} : v_1, v_2 \in V^{out}$ with $v_1 \neq v_2$ and $d = \sum\limits_{v_{1_d}, v_{2_d}} relationlabel(v_{1_d}, v_{2_d})$ with $v_{1_d} \in descendants(v_1) \cup \{v_1\}, v_{2_d} \in descendants(v_2) \cup \{v_2\}\}$

Hence, the labels of higher-level call relations result are the result of aggregating the labels of the lower-level call relations.

6.2 Focusing on Temporal Order

A knowledge of the temporal order of how implementation units of a software system are executed is essential for understanding system behavior. Experiments conducted by Pennington [169] give evidence that developers first build a mental model of the control flow when trying to understand a software system.

The views outlined in this section explicitly depict the temporal order of function calls captured in a trace. The main characteristic of these views' underlying geometric models is that they allocate one physical dimension of the layout space to explicitly encode the temporal order. The main challenge is to cope with the vast amount of calls that are typically contained in a trace. One approach to solve this scalability problem is to provide a set of views displaying the trace on different granularity levels:

- A macroscopic view gives an overview of the complete trace;
- a microscopic view provides details of selected time ranges;
- a third view, which is based on pruned traces (cf Chapter 5), bridges the gap between macroscopic and microscopic views.

6.2.1 Temporal Overview - A Macroscopic View

Purpose of the View

The main purpose of the temporal overview (Figure 6.5) is to provide a degree of orientation during trace exploration. It shows the complete trace in a way that exposes phases, i.e., patterns of function executions become visible. Similar and repeated execution patterns can be identified. Details of function activity at selected

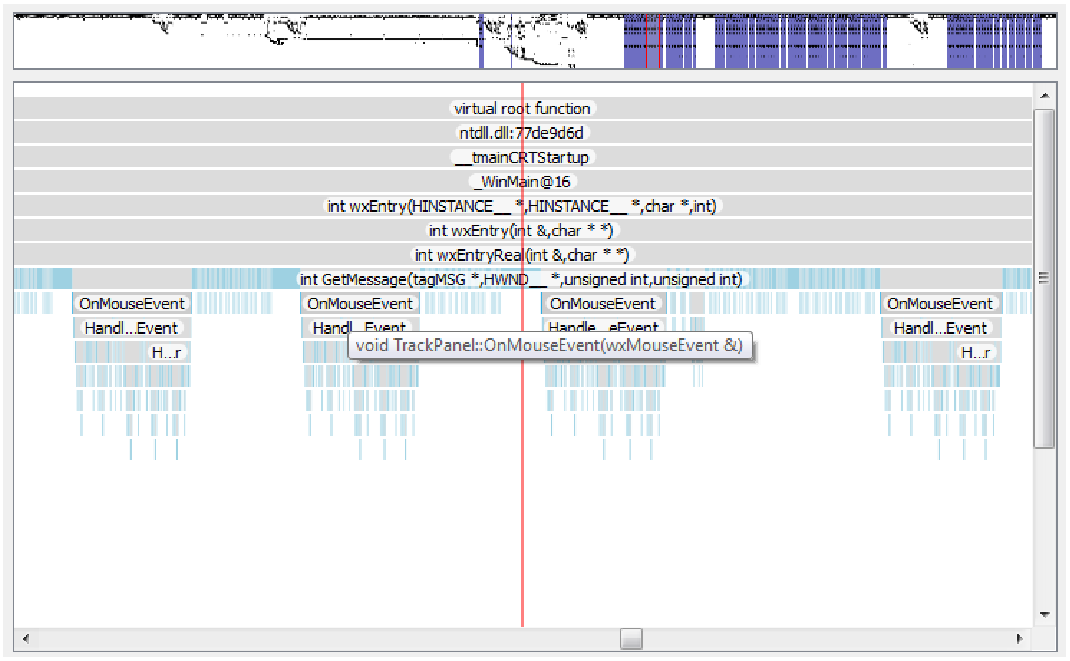

virtual root function
ntdll.dll:77de9d6d
__tmainCRTStartup
_WinMain@16
int wxEntry(HINSTANCE__ *,HINSTANCE__ *,char *,int)
int wxEntry(int &,char * *)
int wxEntryReal(int &,char * *)
int GetMessage(tagMSG *,HWND__ *,unsigned int,unsigned int)
OnMouseEvent
OnMouseEvent
OnMouseEvent
OnMouseEvent
Handl...Event
Handl...Event
Handle...eEvent
Handl...Event
H...r
H...r
void TrackPanel::OnMouseEvent(wxMouseEvent &)

Figure 6.5: The temporal overview (upper part) presents an overview of when functions are active over a certain time (blue shaded areas). Additionally, it depicts which time range (shaded red) is shown in detail in other views, e.g, the call stack view (lower part).

points in time can be obtained on demand by a showing of the corresponding function call stacks. With this on-demand information at hand, a developer can identify system behavior on the basis of a visual pattern.

Moreover, the macroscopic view permits developers to understand at what stage in the complete trace a function or higher-level module is active (by selecting it in another view).

Supported Comprehension Strategies

This view is helpful with both top-down and bottom-up comprehension strategies. Top-down comprehension strategies are supported in such a way that the developer may start by formulating hypotheses on the high-level behavior captured in the trace. When analyzing an *address book import* feature of a mail client software, for instance, a developer may hypothesize that the import feature is implemented by, firstly, parsing an external address book file and, secondly, merging the newly obtained mail contact data with existing mail contacts. Visually analyzing a corresponding trace with the temporal overview makes verifying and rejecting the hypothesized two-step behavior of *parsing* and *merging* possible. The analysis might reveal that an additional processing step between the parsing step and the merging step is converting the contact data from one intermediate parsing related data structure to another. Depending on the type of maintenance task, the developer refines the original hypothesis and continues trace exploration. If the developer's task is to implement an as yet not supported address book format, for instance, the developer might continue exploration with analyzing the revealed conversion step in more detail.

Bottom-up comprehension strategies are supported by the temporal overview, which provides contextual information that helps to aggregate findings obtained via analyzing low-level implementation artifacts into higher-level abstractions: When developers analyze details of a specific function in another view and try to understand the function's role in the captured system execution, the temporal overview provides information on the phases the function participates. In this way, developers are able to distinguish between functions that are frequently used throughout the captured behavior and functions that are specific to one single phase. Knowing the phase-specific functions helps to gain an understanding of how higher-level behavior is implemented using lower-level functions.

Implementation - Filtering Step

The temporal overview is based on the complete trace $T = (F,C)$. No filtering operations are applied.

Implementation - Mapping Step

The geometry model that results from the mapping step is a 2D area that is vertically segmented into strips of equal height. Each strip corresponds to a function $f \in F$. The association between function and strip is obtained by ordering the calls in C

according to their start time. The order of the functions is then determined by the order in which they appear as callee function in the ordered set C.

The horizontal dimension of the strips reflects the temporal order. Each call is mapped—according to its start and end time—onto the strip that corresponds to the call's callee function. In those time regions where a call is the top call on the call stack, the strip is painted black.

Limitations

The strength of this view lies in providing an overview of coarse-grained system behavior. Fine-grained behavior represented by a visual pattern consisting of only a few pixels in horizontal dimension cannot be explored satisfactorily.

Multiple View Interaction

The temporal overview allows for cross-referencing by translating requests on selected functions and modules (being selected in another view) into time ranges where they are active. Developers may analyze details in another view and obtain coarse-grained context information from the temporal overview. Additionally, the view provides orientation and helps developers while exploring a time range in detail in another view by highlighting this time range.

6.2.2 Call Stack View - A Microscopic View

Purpose of the View

The purpose of the call stack view is to provide details of the temporal order of calls for a given, reasonably small time range (Figure 6.6). With it, developers receive answers to questions such as:

- What happened before a specific call is executed?
- What happens after a specific call is executed?
- What is the sequence of (nested) subcalls that a specific call triggers?

Supported Comprehension Strategies

The call stack view aids developers in performing bottom-up comprehension strategies. Starting with a specific call, a developer may use this view to analyze the detailed sequence of calls executed before, after, and during the call. This detailed low-level information is useful, as it helps developers to gain a higher-level of understanding with regard to the analyzed system behavior.

Implementation - Filtering Step

Several filtering operations are applied to reduce the trace T to T^{out} whose calls are taken to build the geometry model.

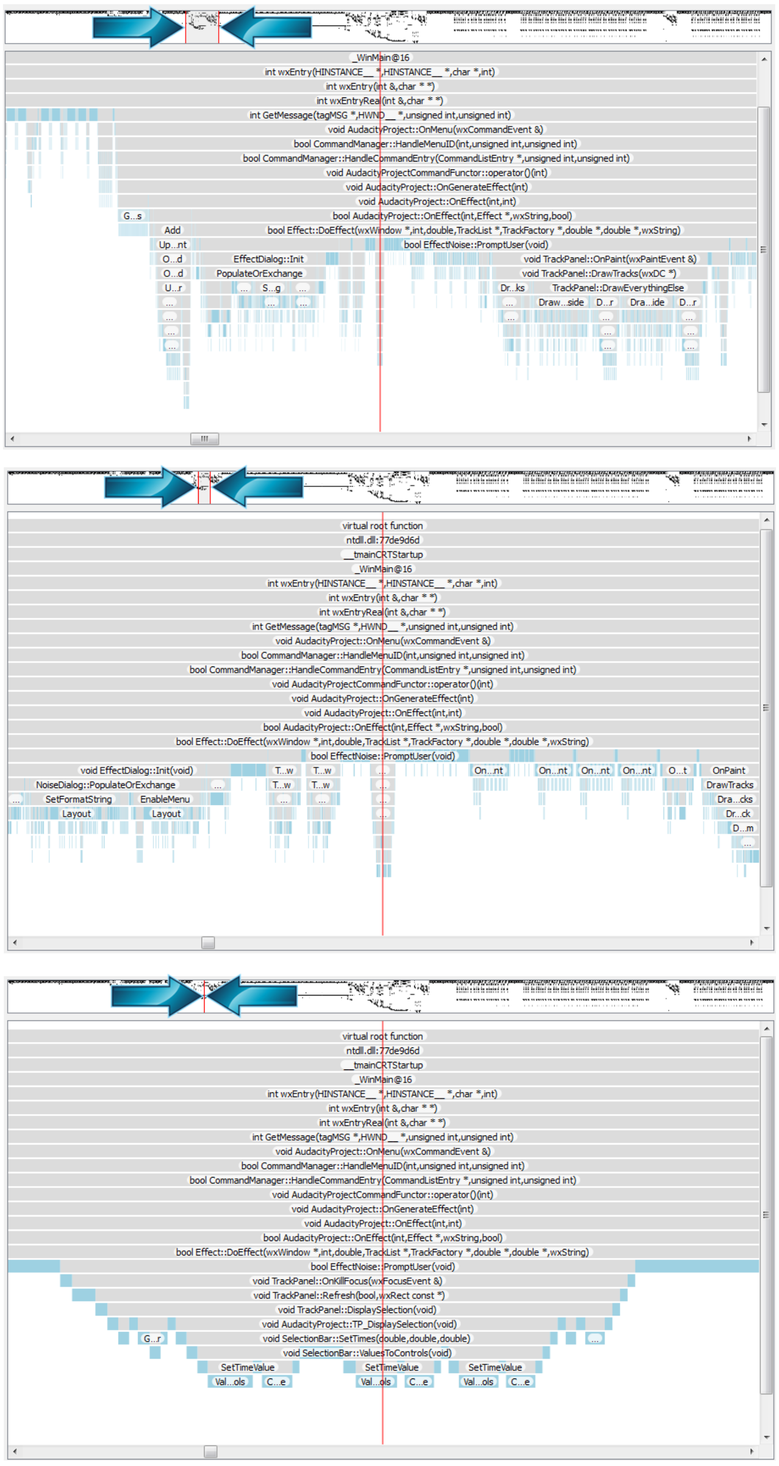

_WinMain@16
int wxEntry(HINSTANCE__ *,HINSTANCE__ *,char *,int)
int wxEntry(int &,char * *)
int wxEntryReal(int &,char * *)
int GetMessage(tagMSG *,HWND__ *,unsigned int,unsigned int)
void AudacityProject::OnMenu(wxCommandEvent &)
bool CommandManager::HandleMenuID(int,unsigned int,unsigned int)
bool CommandManager::HandleCommandEntry(CommandListEntry *,unsigned int,unsigned int)
void AudacityProjectCommandFunctor::operator()(int)
void AudacityProject::OnGenerateEffect(int)
void AudacityProject::OnEffect(int,int)
bool AudacityProject::OnEffect(int,Effect *,wxString,bool)
bool Effect::DoEffect(wxWindow *,int,double,TrackList *,TrackFactory *,double *,double *,wxString)
bool EffectNoise::PromptUser(void)
G...s
Add
Up...nt
O...d
EffectDialog::Init
PopulateOrExchange
U...r
S...g
void TrackPanel::OnPaint(wxPaintEvent &)
void TrackPanel::DrawTracks(wxDC *)
Dr...ks
TrackPanel::DrawEverythingElse
Draw...side
D...r
Dra...ide
virtual root function
ntdll.dll::77de9d6d
__tmainCRTStartup
void EffectDialog::Init(void)
NoiseDialog::PopulateOrExchange
SetFormatString
EnableMenu
Layout
T...w
On...nt
O...t
OnPaint
DrawTracks
Dra...cks
Dr...ck
D...m
void TrackPanel::OnKillFocus(wxFocusEvent &)
void TrackPanel::Refresh(bool,wxRect const *)
void TrackPanel::DisplaySelection(void)
void AudacityProject::TP_DisplaySelection(void)
G...r
void SelectionBar::SetTimes(double,double,double)
void SelectionBar::ValuesToControls(void)
SetTimeValue
Val...ols
C...e

Figure 6.6: The call stack view presents the call stack for a time range of the trace in detail. Three differed zoom levels are shown, i.e., different time ranges are shown. The time marker in the center points to the same time value in all diagrams.

- time range constraint: $T' = \Psi^{\text{time-range-constraint}}(T)$
- call tree removal: $T'' = \Psi^{\text{call-tree-removal}}(T')$
- module removal: $T^{out} = \Psi^{\text{module-removal}}(T'')$

Implementation - Mapping Step

The geometry model is obtained by (linearly) mapping the time range that parameterizes $\Psi^{\text{time-range-constraint}}$ to the horizontal dimension. Calls with their start and end time are represented as horizontal bars. A bar's vertical position is determined by its depth within the call stack. Depending on the horizontal range that a bar spans, the bar is annotated with the name of the corresponding callee function.

Limitations

The concept places no limitations on the chosen time range. As to large time ranges with many calls (e.g., > 50.000), scalability issues may appear regarding computation performance. One way of coping with this problem is to sample the call sequence and display only the sampled calls. Consequently, not all calls but only randomly picked ones are visualized.

Multiple View Interaction

The call stack view supports cross-referencing by way of highlighting, if they are selected in another view. Likewise, calls of selected functions and (higher-level) modules are determined and highlighted.

6.2.3 Phases View - Bridging the Gap between Microscopic and Macroscopic Views

Purpose of the View

The purpose of the *phases view* (Figure 6.7) is to provide a compact view on the trace that permits to rapidly navigate in a top-down approach to those parts of the trace that are relevant to the maintenance task at hand. The data that the view is based on is a *pruned trace* (cf Chapter 5). Pruned traces contain those calls that crucial decision points when navigating within a trace in a top-down fashion. Pruned traces contain:

- *Inner phases*, which represent the execution of higher-level system functionality by means of triggering multiple times lower-level functionality.
- *Leaf phases*, which represent lowest-level functionality. Leaf phases may represent, for instance, costly calls that do not trigger many further calls.

A major benefit to be derived from automatically identifying *phases* (cf Chapter 5) is that their callee functions' names often provide a helpful description of the system behavior that is executed in the phase. Hence, the view is able to depict sequences of named phases on various levels of abstraction.

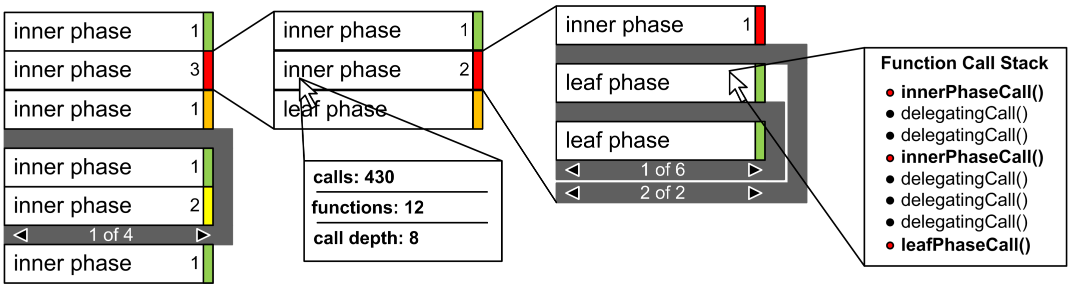


Figure 6.7: The phases view presents the trace in a highly compact way that permits quick navigation to those parts of the trace that are relevant to a developer's maintenance task at hand.

As a sequence of phases may be composed of repetitive patterns, the phases view applies a pattern detection algorithm on the sequence so that patterns can be visualized explicitly and in a compact form.

Despite the view's purpose of providing an efficient navigation means, the view provides high-level execution context information in combination with other views. As with the temporal overview, the phases view provides answers to questions such as:

- In which phases is a specific function (or are functions of a specific module) executed?
- To which higher-level behavior does the time range belong that is seen in the call stack view?

In contrast to the temporal overview, the phases view provides execution context information on various abstraction levels—and not only on the highest-level, i.e., regarding the complete trace's time range.

Supported Comprehension Strategies

As described in the temporal overview section, the phases view also supports both top-down and bottom-up comprehension strategies. Developers perform top-down strategies by first analyzing the sequence of highest-level phases and forming hypotheses about the global system behavior captured in the trace. Then the hypotheses are successively refined by requesting details of specific phases: The process of unfolding an inner phase provides an execution overview over a smaller time range and on a more fine-grained abstraction level.

Bottom-up strategies are made more accessible, as the phases view provides information on the execution context of functions and higher-level modules while developers are analyzing a different view. A knowledge of the execution context helps developers to gain a higher level of understanding of system behavior while analyzing low-level facts (cf discussion in Section 6.2.1).

Implementation - Filtering Step

The starting point of the filtering step is a pruned trace $T^{\dagger} = (F^{\dagger}, C^{\dagger})$. Given a phase $p \in C^{\dagger}$, the filtering step reduces the $T^{\dagger}$ to a filtered, pruned trace $T^{\dagger out} = (F^{\dagger out}, C^{\dagger out})$ that only contains details of subphases of p and of all its enclosing phases (Figure 6.8):

- $C^{\dagger'} = \{p\} \cup enclosingphases(p)$
- $C^{\dagger out} = C^{\dagger'} \cup \bigcup_{p' \in C^{\dagger'}} subphases(p')$
- $F^{\dagger out} = funcset(C^{\dagger out})$

Implementation - Mapping Step

The mapping step assigns a geometric representation to each phase $p \in C^{\dagger out}$. They are represented by rectangles labeled with the callee function name of the call that the phase stands for (Figure 6.7). Additionally, the phases' meta-information is visually encoded within the rectangle.

The following meta-information is shown:

- A label shows how many calls of the original trace are executed during the phase's time span.
- Colored areas encode the number of triggered calls, the call depth of the call tree that starts from the corresponding call, the execution time of the phase, and the number of distinctive callee functions of the triggered calls. These

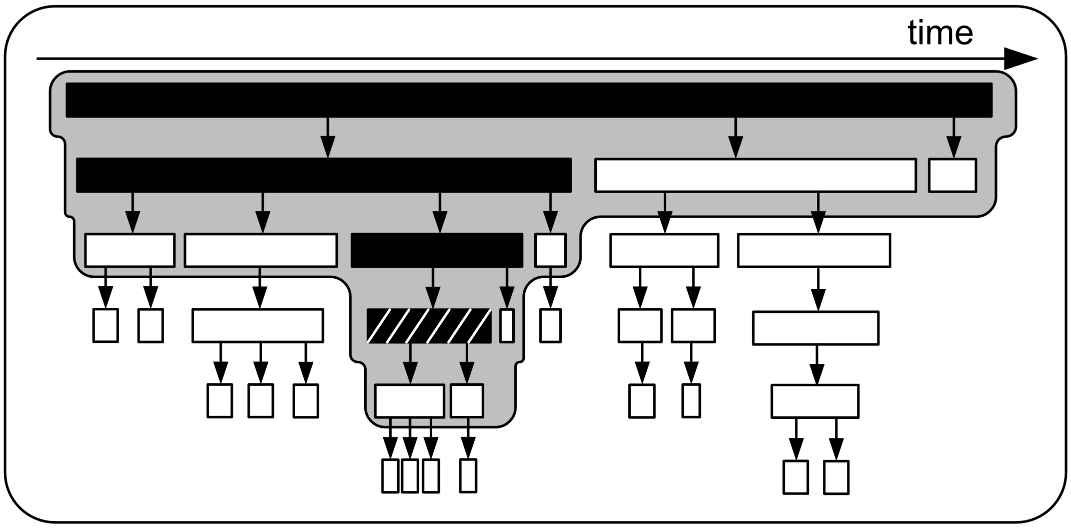


Figure 6.8: The filtering step for the phases view is based on a phase p in a pruned trace (striped bar). All enclosing phases together with their subphases remain in the filtered pruned trace.

visual hints permit developers to assess what to expect when requesting details of the phase (e.g., by unfolding the phase or by consulting another view). We can identify "laborious" parts during system execution by having a closer look at large numbers of triggered calls, deep call trees, or large call costs.

The layout produced by the mapping step follows the hierarchical structuring of the phases. Due to the filtering operation, all phases with equal depth in the phase tree form a sequence of subphases (Figure 6.8). A vertical layer in layout space is allocated to each subphase sequence. However, the subphase sequence is not mapped from this coarse sequential form into a geometry model that fits within the layer. A compact form of the sequence is calculated beforehand by applying phase similarity metrics and detecting repetitive patterns (cf Section 5.3).

Phase Similarity and Pattern Detection Figure 6.9 illustrates the three steps necessary for obtaining a compact representation for a phase sequence:

1. Phases are checked for similarity with respect to the *phase similarity metrics* and are compared to a *similarity threshold* (cf Section 5.3). Unique identifiers are assigned to the phases, whereas similar phases are given the same identifier.
2. By applying a tool called *sequitur* [153] for extracting grammar rules from strings, hierarchical patterns are infered in the sequence of identifiers.
3. The sequence of phases is depicted in a folded form that shows only one instance of multiple occurrences of anyone pattern. Gray boxes represent patterns visually. Arrow symbols in the boxes indicate that the developers may interactively choose the pattern instance to be shown. A label shows the currently depicted pattern instance and the total number of instances.

Being given the opportunity to interactively choose the similarity threshold within its value range of [0,1] helps to adjust the "compactness" of the sequence representation.

Limitations

A limitation of the phases view lies in the underlying algorithm for calculating pruned traces. The algorithm removes *lightweight calls* and *control delegating calls* (cf Chapter 5), which means that calls of the original trace are missing when the trace is being explored via the phases view. Pop-up information, however, still provides aggregated information on the missing calls. This helps developers to decide when to (temporarily) stop analyzing the trace in the phases view and to switch to another view to explore the complete trace data. In a sense, using the phases view is like driving on the "fast lane" to get near to regions of interest in the trace; at some point during exploration of the pruned trace, however, the developers need to switch to more detailed views.

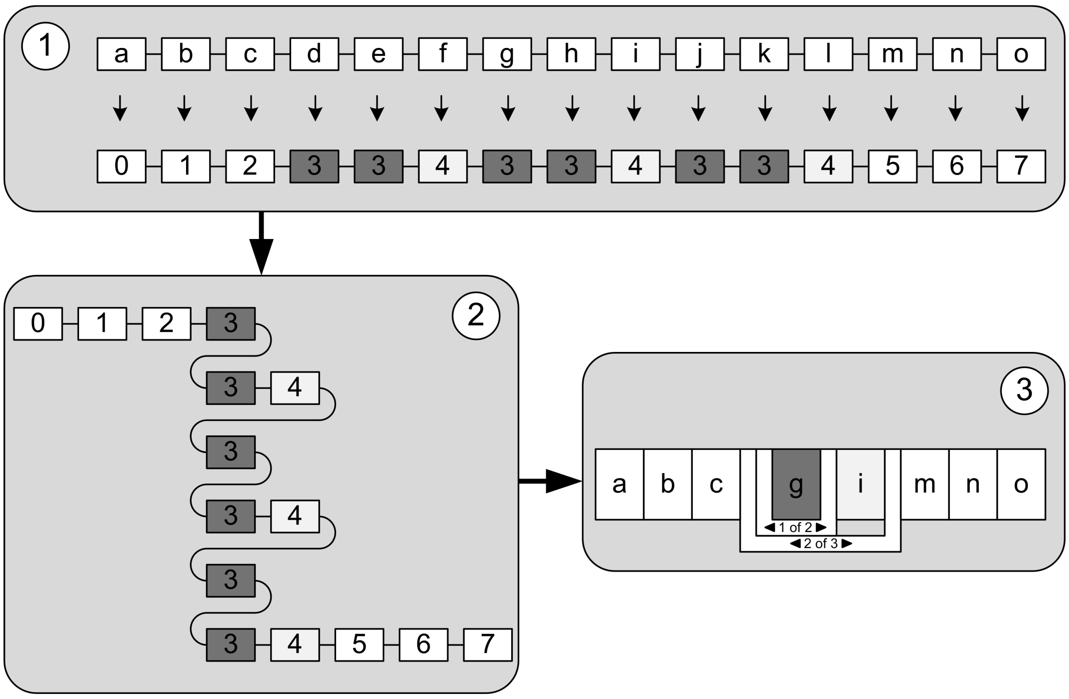


Figure 6.9: A sequence of phases is compacted in three steps: (1) Assigning identifiers according to phase similarity. (2) Calculating repetitive patterns. (3) Folding the patterns.

Multiple View Interaction

As with the *temporal overview*, the phases view provides (1) orientation while exploring the trace in other views and (2) permits developers to analyze when specific functions or higher-level modules are active in time that are selected in another view.

6.3 Focusing on Structure

This section presents views that focus on structure contrary to the views focusing on temporal order. Central elements of these views are (1) call relations contained in call graphs and (2) the functional decomposition of the software system implementation captured in a module hierarchy.

6.3.1 Structure Overview

Purpose of the View

The *structure overview* depicts the complete module hierarchy that the system implementation is composed of (Figure 6.10). When exploring a trace in another view, the structure overview provides help by revealing to which parts of the system implementation a function or module belongs. Organizing the system implementation into a hierarchy of modules is one way of coming to terms with the complexity of the

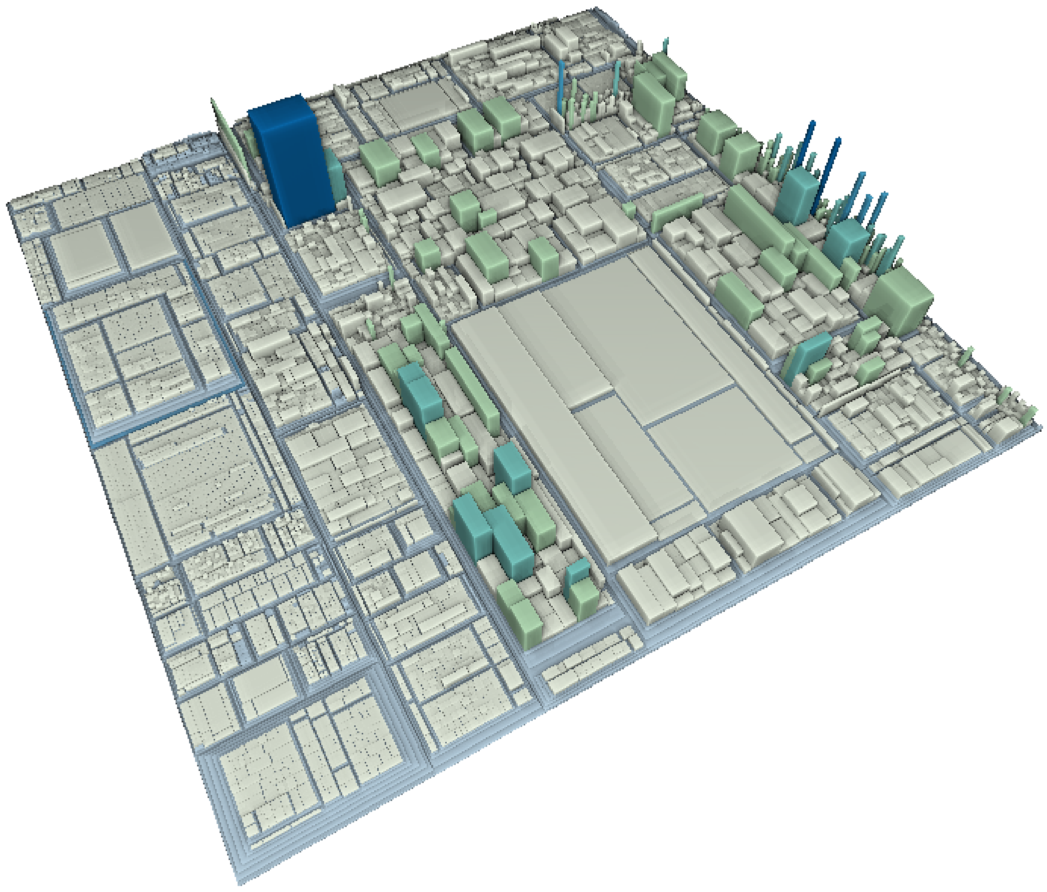


Figure 6.10: The *structure overview* depicts the complete module hierarchy using a $2\frac{1}{2}$D treemap layout technique. Parts of the implementation that are active during a specific execution can be visualized.

system. Modules thereby represent higher-level "*implementation units that provide coherent units of functionality*" [35]. "*Modularization [is] a mechanism for improving the flexibility and comprehensibility of a system*" [165]. Hence, the structure overview helps developers to relate a single function to the modules it belongs to and to derive their purpose and responsibility. Essentially, it is the structure overview that is used during trace exploration to obtain module related context information about functions.

Supported Comprehension Strategies

The structure overview is instrumental to developers while performing bottom-up comprehension strategies. While developers explore detailed and low-level aspects of the trace, e.g., when analyzing a sequence of calls in the call stack view, they need to understand what the functions' roles are in the captured system behavior. Context information showing to which modules a function belongs, helps to clarify the overall role of the function and allows for the building-up of higher-level abstractions of the analyzed system behavior.

Implementation - Filtering Step

No filtering step is necessary for this view.

Implementation - Mapping Step

During the mapping step, the module hierarchy is mapped onto a set of nested rectangular volumes as a *treemap* [80, 189, 218]. Treemaps are "*compact, space-filling displays of hierarchical information, based on recursive subdivision of a rectangular image space*" [218]. Each module is represented by a volume with rectangular extend, all its children recursively mapped onto its top surface. The partitioning of the top surface is set out according to a specific treemap layout such as a *slice-and-dice*, *clustered*, *squarified*, or *ordered* layout [191]. The surface area covered by a module representation is proportional to its lines-of-code metrics (LOC). The height of a module in the treemap corresponds to its depth in the module hierarchy. The resulting nested rectangular volumes help to pinpoint the position of the module within the hierarchy.

In order to further improve perception of the nested rectangular volumes, spezialized shading techniques that simulate physical light distribution can be applied, i.e., ambient occlusion, or, alternatively, a 2D treemap representation can be used. For interactive exploration, however, the 3D variant of the treemap provides essential benefits as it uses depth to indicate the hierarchy level, leverages advanced illumination and shading techniques, and provides 3D interaction tools for user interaction.

Implementation - Rendering Step

In the rendering step, the 3D model that represents the treemap is projected onto a 2D view plane by projective projection. 3D interaction tools such as zooming, rotation, and panning allow users to interactively explore the treemap.

Limitations

The structure overview depicts the complete module hierarchy in a dense way with lower-level modules occupying less and less screen space and, hence, is useful when exploring higher-level modules. It cannot show hierarchies with large depth. However, in typical software implementations, the hierarchical level is usually rather low in contrast with general information visualization contexts. Collberg et al. [39], for instance, have analyzed 1132 open-source Java Jar files and found a maximum module depth of 8 and an average depth of approximately 4.

Multiple View Interaction

In combination with another view, which selects a function or a higher-level module, the structure overview highlights all modules from the selected one to the module hierarchy's root module. Highlighting means changing the color selected for the module representations and annotating them with their module name labels.

6.3.2 Collaboration View

Purpose of the View

The *collaboration view* shows call relations between modules (Figure 6.11). It enables developers to understand which modules interact via calls with each other. This analysis is performed either on the complete trace or on a selected time range. The time range restriction is an important feature and has been installed to obtain a clear picture of what happens in the analyzed software system during execution of a specific functionality.

Supported Comprehension Strategies

The view supports a top-down comprehension strategy. Developers have the option to choose a time interval of interest within another view, e.g., temporal overview or call stack view. Subsequentially, collaboration between the highest-level modules is shown. The developer may then in succession unfold modules to obtain a more and more detailed view on how specific modules down in the module hierarchy interact with the other modules. This involves developers starting with a global hypothesis about system behavior and successively refining their hypotheses to gain a detailed understanding of which lower-level module is important within the analyzed time range and how it interacts.

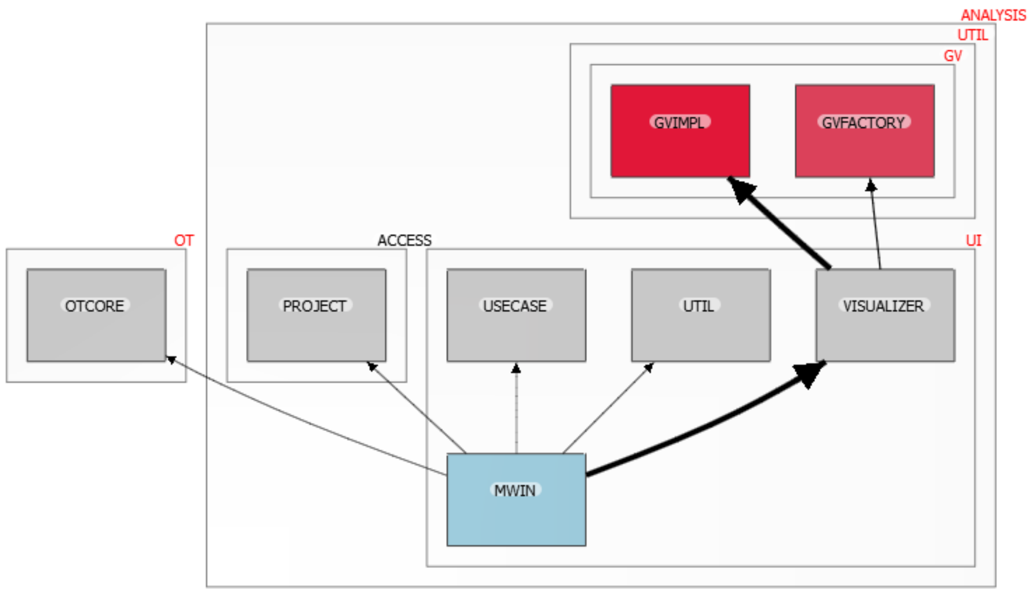


Figure 6.11: The *collaboration view* depicts how modules interact during a selected part of the trace.

Implementation - Filtering Step

Several operations are applied to the trace T to obtain a call graph $G^{out} = (V^{out}, R^{out})$ containing call relations between higher-level modules:

- time range constraint: $T' = \Psi^{\text{time-range-constraint}}(T)$
- module removal: $T'' = \Psi^{\text{module-removal}}(T')$
- call graph conversion: $G = \Phi(T'')$
- module collapsing: $G^{out} = \Gamma(G)$

Implementation - Mapping Step

The mapping step results in a 2D layout space that contains rectangles for representing modules and arrows for call relations. The containment relations between modules are represented by embedding child rectangles into parent rectangles. The layout of the rectangles and arrows is calculated by algorithms for clustered graphs that are provided by the graph drawing *Graphviz* [64] library.

Limitations

One limitation of this type of view is that the number of depicted modules needs to be reasonably small; we have experienced that up to 50 modules can be shown using standard displays, otherwise the resulting image will suffer from visual clutter. However, in concrete trace exploration scenarios, a small number of modules can be obtained by successively narrowing down the "interesting" time range that needs to be visualized. Furthermore, modules can be removed, if they are identified as not being relevant to the given maintenance task.

Multiple View Interaction

This view does not provide any context information. However, it serves as view for requesting context information from other views by selecting a module.

6.3.3 Call Neighborhood View

Purpose of the View

The call neighborhood view depicts for a given time range and a given function f, which functions it calls and how it is called by other functions (Figure 6.12). On the one hand, the view permits developers to understand how the function is coordinated by other functions, and on the other hand, the view shows how the function f coordinates other functions to implement its functionality in the software system. The presented functions and their call relations are embedded within the module hierarchy. This way, developers are assisted in understanding the functions' role during system execution: The information to which modules a function belongs

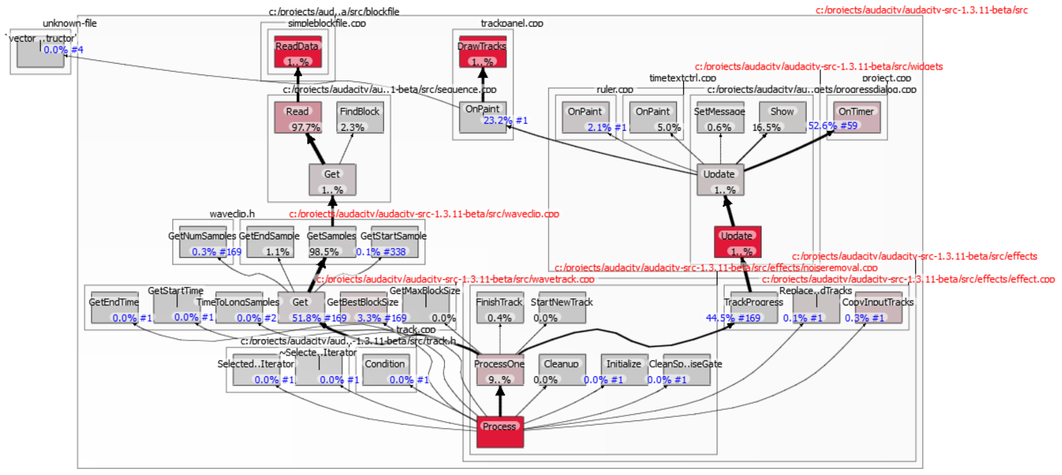


Figure 6.12: The *call neighborhood view* depicts for a given function the caller and callee relations to other functions. Additionally, it depicts runtime statistics on call relations. Furthermore, it shows how the functions are part of higher-level modules.

provides valuable clues when endeavoring to understand the function's semantics (cf Section 2.5.1).

Supported Comprehension Strategies

This type of view supports bottom-up comprehension strategies. Developers can analyze for a small function set how the functions interact. Moreover, by analyzing the higher-level context, i.e., how functions are embedded within higher-level modules, developers can gain a more global understanding of interaction in the system. In particular, they can see how the specific function interactions displayed are parts of a more general module interaction.

Implementation - Filtering Step

Several operations are applied to a trace T to obtain a call graph $G^{out} = (V^{out}, R^{out})$:

- time range constraint: $T' = \Psi^{\text{time-range-constraint}}(T)$
- module removal: $T'' = \Psi^{\text{module-removal}}(T')$
- call graph conversion: $G = \Phi(T'')$

Call graph G provides the basis for deriving the graph $G^{out} = (V^{out}, R^{out})$ to be shown in the call neighborhood view. Starting point is a function $f \in F$ that becomes the first element in V^{out}. Next, f's outgoing call relations $R_f^{\rightarrow}$ to the callee functions $V_f^{\rightarrow}$ are determined and added to V^{out} and R^{out}, respectively. $R_f^{\rightarrow} = \{r \in R : \mathit{source}(r) = f\}$ and $V_f^{\rightarrow} = \{\mathit{destination}(r) \in V : r \in R_f^{\rightarrow}\}$.

In the same way, incoming call relations $R_f^{\leftarrow}$ from the caller functions $V_f^{\leftarrow}$ are determined and added to V^{out} and R^{out}, respectively. $R_f^{\leftarrow} = \{r \in R : destination(r) = f\}$ and $V_f^{\leftarrow} = \{source(r) \in V : r \in R_f^{\leftarrow}\}$.

This basic call neighborhood graph is augmented by indirect call relations on demand by the developers. For a function $f \in V_f^{\rightarrow}$ the developer may analyze further outgoing call relations. In the case of a function $f \in V_f^{\leftarrow}$ the incoming call relations are made available.

Implementation - Mapping Step

The geometry model uses a 2D layout space. Similar to the collaboration view, rectangles represent functions and higher-level modules. Call relations are represented by directed arrows. Containment relations are expressed by embedding child rectangles into parent rectangles. The thickness of the arrows corresponds to the costs of a call relation. In addition, the arrows are annotated with labels that show how many calls are cumulated by each call relation.

The layout of the rectangles and arrows is calculated by algorithms for clustered graphs that are provided by the graph drawing *Graphviz* [64] library.

Limitations

The number of functions that can be shown in the call neighborhood view is limited. In our experience, graphs with more than approximately 50 functions generally have too many call relations and the resulting images start to contain too much visual clutter.

Multiple View Interaction

If a point in time is selected in another view (e.g., call stack view), all the functions and call relations that are part of the call stack of the selected point in time are highlighted.

For a point in time (selected in another view) the call neighborhood view highlights the functions and call relations that correspond to the call stack of the given point in time. This facilitates understanding the call stack in terms of higher-level modules, e.g., developers can explore how control is passed from module to module. A further discussion on how to integrate the time dependent call stack information is given in [221].

6.4 Focusing on Source Code

The source code represents the prime artifact type created and modified by developers in software development processes. It codifies the structure and behavior of a software system. While the static architecture of the system is represented in a straight forward way, the dynamics of the software system is not intuitively expressed by the source code. Two problems arise when trying to understand system behavior on the basis of reading source code:

1. Programming concepts such as *polymorphism* in object-oriented programming languages or *function pointers* in various procedural programming languages allow developers to postpone from compile-time to execution-time any decision they have to make on which function is to be called at a call statement. Hence, understanding system behavior by reading code is extremely difficult, in fact it is impossible, if these constructs are used. For instance, if the common *observer design pattern* [63] is applied, it is impossible to tell by reading code which the observing objects are that are notified by the observed subject object. In some languages such as C++, operation overloading hides the function call chains that are triggered.

2. Understanding system behavior by reading source code means identifying those parts of the code that participate in the respective behavior. If a "grown" software system needs to be analyzed, it is highly possible that many essential functions consist of large amounts of lines of code (Lanza and Marinescu call them *brain methods* [118]). In this case, it is tedious work having to separate the "wheat from the chaff", i.e., identifying the code lines executed during the analyzed behavior from the large amount of unused code lines.

Another problem in connection with object-oriented software systems is the circumstance that it is difficult to "see" (a) the actual instantiation processes and (b) state changes in the source code.

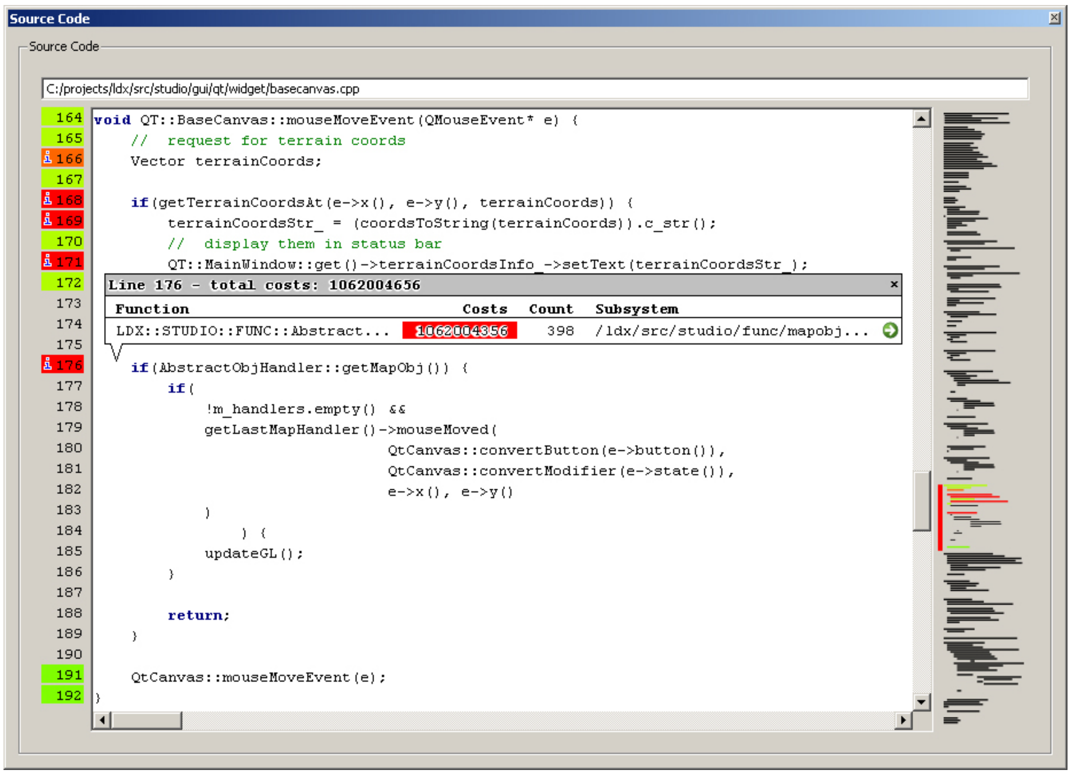
6.4.1 Enriched Code View - Enriching Source Code with Runtime Information

Purpose of the View

The purpose of the *enriched code view* (Figure 6.13) is to provide solutions for coping with the two core problems when reading source code. Two strategies are applied:

- To assist developers in rapidly identifying the executed code lines in large function implementations and to enable them to skip the unexecuted parts, the source code of a function is presented in a miniaturized way, the textual layout of the code thereby being conserved (cf *SeeSoft* metaphor [52]). Code lines are reduced to pixel lines. A line is colored if it contains an executed call statement. Call costs are encoded in color values. The miniaturized code is placed next to the original code. There, the lines with an executed call statement are colored in the same way.

- For code lines with executed call statements, details of the calls can be obtained on demand. A pop-up window reveals (1) which calls "lie" behind the call statement, (2) how often the calls were performed, and (3) how costly the calls were. These detailed descriptions reveal what really happened during runtime—even if polymorphism or function pointers are used.



```
void QT::BaseCanvas::mouseMoveEvent(QMouseEvent* e) {
    // request for terrain coords
    Vector terrainCoords;

    if(getTerrainCoordsAt(e->x(), e->y(), terrainCoords)) {
        terrainCoordsStr_ = (coordsToString(terrainCoords)).c_str();
        // display them in status bar
        QT::MainWindow::get()->terrainCoordsInfo_->setText(terrainCoordsStr_);
```

```
Line 176 - total costs: 1062004656
 Function                        Costs       Count   Subsystem
 LDX::STUDIO::FUNC::Abstract...  1062004656  398     /ldx/src/studio/func/mapobj...
```

```
    if(AbstractObjHandler::getMapObj()) {
        if(
            !m_handlers.empty() &&
            getLastMapHandler()->mouseMoved(
                                QtCanvas::convertButton(e->button()),
                                QtCanvas::convertModifier(e->state()),
                                e->x(), e->y()
            )
                ) {
            updateGL();
        }

        return;
    }

    QtCanvas::mouseMoveEvent(e);
}
```

Figure 6.13: The *enriched code view* presents the source code enriched with additional runtime information.

Supported Comprehension Strategies

The view supports bottom-up comprehension strategies by displaying low-level source code information together with runtime information for specific behavior. Developers do not need to mentally reconstruct several of the many possible control paths in the function's code before they are able to identify the control path relevant to the observed behavior. Instead, the additional runtime information displayed guides them along the relevant control path taken at runtime. As developers skip large parts of the code, they will better find themselves capable of building up a higher-level mental model of system behavior.

Implementation - Filtering Step

As a first step, the time range constraint operation is applied to the trace $T' = (F',C') = \Psi^{\text{time-range-constraint}}(T)$; this step can be skipped if the entire trace is considered. Subsequently, those calls are extracted that are initiated from the function $f \in F$ of interest. $C_f = \{c \in C' : caller(c) = f\}$.

Implementation - Mapping Step

The resulting geometry model is made up of two parts: The first part is the miniaturized source code presentation, devised, as described by Eick et al., for their SeeSoft code visualization technique [52]. The second part is a standard textual presentation of the source code. The miniaturized code presentation serves as an orientation aid by providing additional visual markers that show which part of the code is currently being displayed in the textual code presentation.

The color encoding of code lines is based on the relative cost of calls that originate from the same code line. Therefore, the calls in C_f are grouped according to the call statements' code lines in the set of all code lines $\mathbb{L}$ (cf Section 2.4.1). Let L_f be the code lines where a subcall was initiated from f's implementation: $L_f = \{callsite(c) \in \mathbb{L} : c \in C_f\}$. The set of calls for a given $l \in L_f$ is: $C_{f,l} = \{c \in C_f : callsite(c) = l\}$.

The call costs and relative call costs of a code line l are calculated as follows:

- $callcosts(f,l) = \sum\limits_{c \in C_{f,l}} costs(c)$
- $relcallcosts(f,l) = \frac{callcosts(f,l)}{\sum\limits_{l' \in L_f} callcosts(f,l')}$

The color of each code line is obtained by mapping its relative call costs $\in [0,1]$ to a sequential color scheme (e.g., color scheme by Brewer et al. [20]).

The pop-up windows that display runtime details per code line on demand, further group the calls in $C_{f,l}$ per callee function and list summarized information for each group, i.e., call costs, call count, callee function, module where the function is implemented.

Limitations

One of the aims of this type of view is to allow developers to rapidly distinguish between executed and not executed parts of the code. An even better support would be provided, if the trace contained complete coverage information on basic block level. This is not the case. Hence, if no call statements exist in specific control branches, no runtime information is available and, therefore, developers will still need to analyze control paths that are not relevant to the given system behavior.

Multiple View Interaction

In combination with other views, the enriched code view provides information on the code context of calls. Pointing on a call relation in the call neighborhood view with the mouse, for instance, highlights the associated code line in the code view, provided that the code of the corresponding function is displayed.

6.5 Linking Views - Multiple Perspectives on Traces

Each view is specialized to reveal selected aspects of the trace and to support specific comprehension strategies. When exploring a trace, developers typically switch between

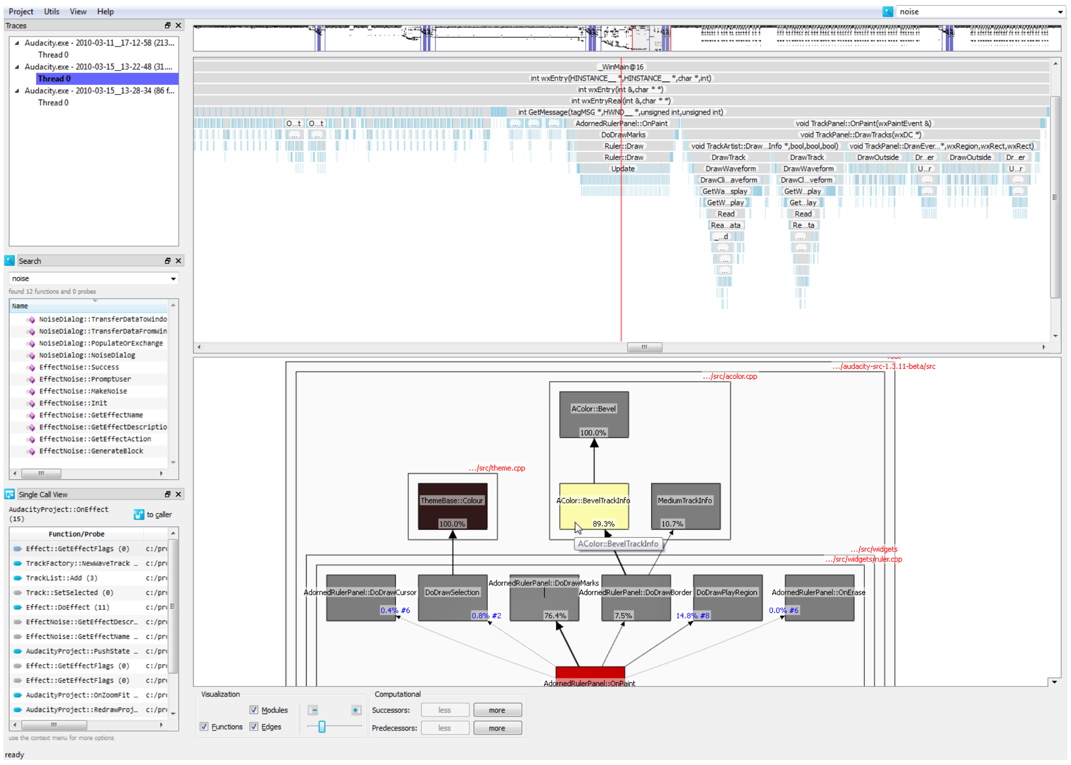

Project Utils View Help
Traces
Audacity.exe - 2010-03-11_17-12-58 (213...
Thread 0
Audacity.exe - 2010-03-15_13-22-48 (31....
Thread 0
Audacity.exe - 2010-03-15_13-28-34 (86 f...
Thread 0
Search
noise
found 12 functions and 0 probes
Name
NoiseDialog::TransferDataToWindo
NoiseDialog::TransferDataFromWin
NoiseDialog::PopulateOrExchange
NoiseDialog::NoiseDialog
EffectNoise::Success
EffectNoise::PromptUser
EffectNoise::MakeNoise
EffectNoise::Init
EffectNoise::GetEffectName
EffectNoise::GetEffectDescriptio
EffectNoise::GetEffectAction
EffectNoise::GenerateBlock
Single Call View
AudacityProject::OnEffect (15)
to caller
Function/Probe
Effect::GetEffectFlags (0)
TrackFactory::NewWaveTrack …
TrackList::Add (3)
Track::SetSelected (0)
Effect::DoEffect (11)
EffectNoise::GetEffectDescr…
EffectNoise::GetEffectName …
AudacityProject::PushState …
Effect::GetEffectFlags (0)
Effect::GetEffectFlags (0)
AudacityProject::OnZoomFit …
AudacityProject::RedrawProj…
use the context menu for more options
ready
noise
_WinMain@16
int wxEntry(HINSTANCE__ *,HINSTANCE__ *,char *,int)
int wxEntry(int &,char * *)
int wxEntryReal(int &,char * *)
int GetMessage(tagMSG *,HWND__ *,unsigned int,unsigned int)
AdornedRulerPanel::OnPaint
void TrackPanel::OnPaint(wxPaintEvent &)
void TrackPanel::DrawTracks(wxDC *)
DoDrawMarks
Ruler::Draw
void TrackArtist::Draw...Info *,bool,bool,bool)
void TrackPanel::DrawEver...*,wxRegion,wxRect,wxRect)
Update
DrawTrack
DrawWaveform
GetWa...splay
Read
.../audacity-src-1.3.11-beta/src
.../src/acolor.cpp
AColor::Bevel
100.0%
.../src/theme.cpp
ThemeBase::Colour
100.0%
AColor::BevelTrackInfo
89.3%
MediumTrackInfo
10.7%
.../src/widgets
.../src/widgets/ruler.cpp
AdornedRulerPanel::DoDrawCursor
0.4% #6
DoDrawSelection
0.8% #2
AdornedRulerPanel::DoDrawMarks
76.4%
AdornedRulerPanel::DoDrawBorder
7.5%
DoDrawPlayRegion
14.8% #8
AdornedRulerPanel::OnErase
0.0% #6
AdornedRulerPanel::OnPaint
Visualization
Modules
Functions
Edges
Computational
Successors:
Predecessors:
less
more

Figure 6.14: Multiple views are synchronized and linked to provide the developer with different perspectives on trace data.

strategies. Hence, they need multiple synchronized views to reach their conclusions (Figure 6.14). Findings made from one view need to be cross-referenced with the findings obtained from other views [16].

The following tables summarize the cross-referencing opportunities of the views. Table 6.3 shows what kind of artifacts are explicitly depicted within the views and are available for selection. Table 6.4 shows which views provide context information on any one selected artifact.

The linking of views is implemented by highlighting related artifacts in those views where an artifact is selected from within one view. Schumann and Müller use the term *brushing* to describe this way of providing a common context for a set of different graphical representations of a data set [186].

View	Function	Module	Call	CallRelation	TimePoint	TimeRange
Temporal Overview					x	x
Call Stack View	x		x		x	x
Phases View	x		x		x	x
Structure Overview	x	x				
Collaboration View	x	x		x		
Call Neighborhood View	x	x		x		
Enriched Code View				x		

Table 6.3: The table summarizes which artifacts can be selected from within the views.

View	Function	Module	Call	CallRelation	TimePoint	TimeRange
Temporal Overview	x	x	x	x	x	x
Call Stack View	x	x	x	x	x	x
Phases View	x	x	x	x	x	x
Structure Overview	x	x				
Collaboration View	x	x	x	x		
Call Neighborhood View	x	x	x	x	x	
Enriched Code View			x	x		

Table 6.4: The table summarizes for which artifacts the views provide contextual information.

CHAPTER 7

Applying Trace Visualization during Software Maintenance

Trace visualization applied in software maintenance needs to identify the parts of the trace that are relevant to any given task. One approach is to apply top-down exploration strategies on a trace. In the previous chapter, several views are described that support top-down strategies. Another approach is to find starting points for fine-grained trace analysis by querying trace data by means of structured trace query language [164]. Furthermore, we can use techniques that automatically point to task-relevant parts of the trace. This chapter discusses how reverse engineering techniques can be exploited that deliver relevant artifacts, which in turn act as entry points for detailed trace exploration.

This chapter first discusses analysis techniques that support maintenance tasks and can be combined with trace visualization. Section 7.2 demonstrates how these techniques can be combined. For this purpose, a novel technique for fault localization is proposed that makes use of the benefits of trace visualization.

7.1 Combining Trace Visualization with other Analysis Techniques

Various techniques for supporting maintenance tasks perform two steps, namely fact extraction and fact analysis to obtain a set of artifacts, which is returned as a *result set* (Figure 7.1). These steps are, as a rule, performed automatically. However, the result set may contain false positives, making it necessary for the developer to identify the true positives manually. If the maintenance task at hand refers to a specific system behavior, this is usually done by either *code reading* or by stepping through the system execution, using a conventional symbolic debugger. (Throughout the remainder of the thesis this technique is called *live debugging.*) Both techniques are highly time consuming, depending on the amount of false positives within the result set.

Trace visualization can speed up the process of distinguishing between false and true positives if applied as an intermediate step after obtaining the result set and before starting time consuming code reading or live debugging. Trace visualization reveals the execution context of the artifacts in the result set and enables developers to decide more rapidly whether an artifact is task-relevant or not. With code reading

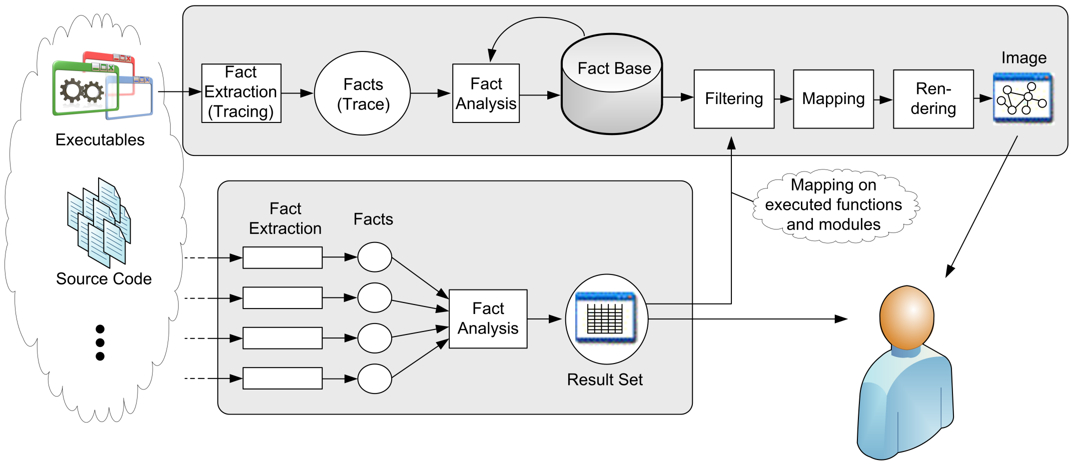


Figure 7.1: Those analysis techniques that support performing a maintenance task (1) related to one specific system behavior and that (2) results in a set of identified artifacts, can reasonably be combined with trace visualization by steering the filtering step of trace presentation. In this way, developers are provided with detailed information on the execution context of the artifacts in the result set and receive support in identifying the true positives in the result set.

on the one hand, this assessment is often difficult because the code represents any possible executions of an artifact. Hence, it is time-consuming to isolate any single execution of interest. With live debugging on the other hand, developers are provided with a local view on system behavior only. They step from one point in time to the next and need to mentally construct a model of execution history—a demanding cognitive task. With trace visualization, developers are presented with explicit models of the behavior. Hence, they can shortcut some of the time consuming steps during code reading and live debugging.

Performing a textual search on the source code is one example of a standard technique for feature location that can be combined with trace visualization. A well-known search tool used for feature location is *grep* [192]. The result set achieved after performing a search query is a list of the code locations that match the search pattern. With this result set, the developer may reduce the set by redefining the search pattern. Later however, the developer needs to manually read the code to understand which code locations are relevant to the searched system feature.

To apply trace visualization for faster identification of relevant code locations in the result set, the hit code locations need to be converted into hit functions (by matching the code locations with the functions' specifying and implementing source code). Afterwards, a trace is taken from the software system while it executes the searched system feature. The function result set obtained from the text search then serves as precise starting points for trace exploration. The visualization provides the execution context of each function and helps developers to decide whether the code location hit is feature relevant or not.

An obvious additional advantage of using trace visualization in combination with static analysis techniques is that the complementary runtime information contained in the trace automatically reduces the result set by removing artifacts from the set that are not active during the observed behavior.

7.2 Maintenance Task: Identifying Recently Introduced Faults

With this novel analysis technique [17, 215] faults in collaboratively developed software systems can be located. Both the maintenance task and the technique conform to the criteria given in the previous section, i.e., the task "locating a fault" refers to a specific system behavior and the analysis technique produces a result set of possibly faulty code locations. Understanding the execution context of the code locations is essential for deciding whether the automatically identified code is related to the faulty behavior or not.

Working on collaboratively developed software systems often leads to situations where a developer enhances or extends system functionality, thereby, introducing faults. At best, the unintentional changes are found immediately by regression tests. Often, however, the faults are detected days or weeks later by other developers who notice strange system behavior while working on different parts of the system. What follows is a highly time-consuming task to trace back this behavior change to code changes in the past.

The proposed technique identifies the recently introduced changes that are responsible for the unexpected behavior. The key idea is to combine dynamic, static, and code change information on the system to reduce possibly large amounts of code modifications in line with those that may affect the system while running its faulty behavior. Following this massive automated filtering step, developers receive support in semi-automatically identifying the "root cause" change by means of trace visualization. Within multiple synchronized views, developers explore when, how, and why modified code locations are executed.

7.2.1 Behavior-Affecting Code Modifications in C/C++

In this section, we discuss which parts of the implementation of a C/C++ software system may affect a specific system behavior if they are modified. The behavior of a system may be changed for various reasons. Changes within source code typically include:

- Code statements in a function's body that are actually executed and modified.
- Data types of variables that are accessed from executed code statements are modified.
  In C/C++, a variable's data type is either declared in a function's body or signature, in the case of a local variable, parameter or return value. Or it is declared outside of "function code" if it is a global variable or an attribute of a class.
- Preprocessor macros are changed.

Changes outside of source code typically include:

- The system environment is/behaves different:
  The system communicates with other systems that behave differently or where persistent data that is being accessed during execution differs.
- The build configuration is modified:
  Third-party libraries are exchanged or preprocessor definitions are modified.

The analysis technique identifies changes performed in source code only. Therefore, developers should be aware of any changes that are manifested outside of source code files. Additionally, as we use a fast and therefore lightweight static code analyzer, macro code dependencies are not tracked [219]. Hence, if a modification of macro code is detected, the developer has to check manually whether the change is responsible for the observed change in system behavior. The main idea is to take advantage of the decision not to analyze whether code changes affect system behavior in general. The goal is to identify whether a code change is responsible for any specific faulty behavior. This behavior is reflected by a trace. With it, the amount of code, i.e., all active functions' code, has to be checked for modifications. Additionally, modifications of data types of variables that are accessed from the executed code need to be checked. This information can be obtained via lightweight static code analysis. Runtime information is exploited, which drastically simplifies the static analysis process because control flow dependencies do not have to be analyzed. Otherwise, it would be necessary to perform a heavyweight static dependency analysis, e.g., by calculating the system dependency graph [86].

7.2.2 Analysis Process

Figure 7.2 illustrates the analysis process a developer has to perform if unexpected system behavior happens to occur. Essentially, the developer must first extract dynamic, static and code change data from the system. The data from the different sources are then combined to identify those functions that may have been affected by recent changes. In a final step, the developer explores how the affected functions are executed in order to be able to assess the impact of their modifications on system behavior. In detail, the steps are:

1. During daily programming work, a developer notices that the software system is behaving in an unexpected way, unlike some days or weeks ago, when the system was still behaving as expected.
2. The developer resorts to tracing and executes the faulty behavior.
3. The developer starts the extraction mechanisms for static and code change data.
   a) A lightweight static analysis of the code is performed.
   b) All code modifications are collected from the time when the system still behaved as expected up to the present.

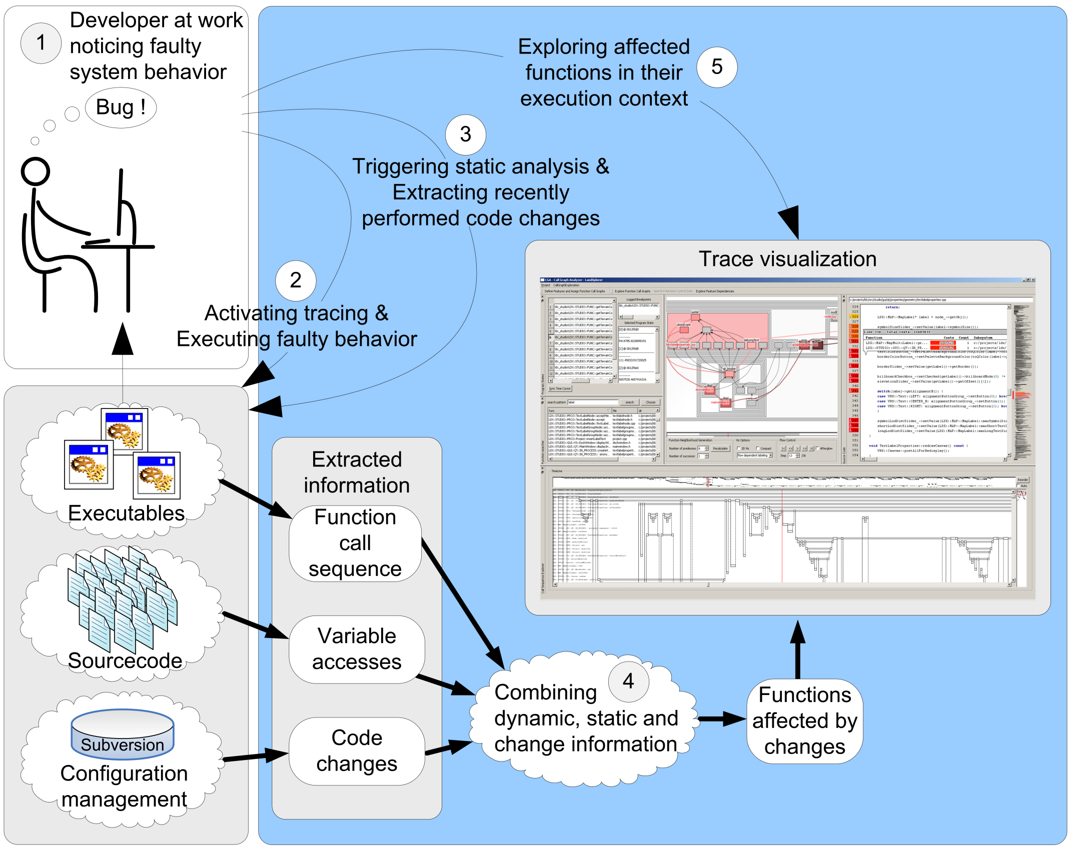

1
Developer at work noticing faulty system behavior
Bug !
5
Exploring affected functions in their execution context
3
Triggering static analysis & Extracting recently performed code changes
2
Activating tracing & Executing faulty behavior
Trace visualization
Executables
Sourcecode
Subversion
Configuration management
Extracted information
Function call sequence
Variable accesses
Code changes
4
Combining dynamic, static and change information
Functions affected by changes

Figure 7.2: The analysis process for identifying recently performed code changes that unintentionally cause faulty system behavior.

4. Dynamic, static, and code change data are combined to identify those functions that have been affected by a code change.

5. By means of trace visualization, the developer explores runtime details of the resulting set of affected functions, giving particular attention to and analyzing how an affected function is executed while the faulty system behavior is being exercised. In this way, the developer can ascertain how the change impacts on the execution of other functions. Another option is to consult those developers who were responsible for the changes in a "suspicious" function.

7.2.3 Fact Extraction

Tracing

The basis of any analysis technique is acquisition of knowledge on the sequence of function calls that are performed while the faulty system behavior is executed. Here, the tracing technique is used that is described in detail in Chapter 4.

Static Code Analysis

As a result of the tracing process (dynamic analysis), the executed functions are known. To resolve the code lines defining data types that are accessed from the executed functions, a static code analysis is performed by exploiting the source code documentation generator tool *doxygen* [157]. The doxygen parser is instrumental in resolving most of the C/C++ preprocessor and template programming peculiarities.

Extracting Code Change Data

Information on the changes in the code from the time when the system last behaved as it was expected can be easily obtained from a software configuration management system (SCM). The implementation of the analysis process supports the SCM *subversion* [170]. Lines of code that are new or modified are modeled by subversion in the shape of added lines in the current version. These lines are checked for modifications together with additional code lines that precede deleted lines. The latter code lines are necessary for establishing that the control flow formerly entered the code but is not being executed anymore in the current version of the system.

7.2.4 Fact Analysis - Detecting Functions Affected by Code Changes

As discussed in Section 7.2.1, code lines have to be checked for modifications that are potentially responsible for unexpected system behavior (Figure 7.3). In other words, (1) lines implementing the functions identified by dynamic analysis have to be checked and (2) lines obtained via static analysis, i.e., lines declaring variables outside of function-related code, however, being accessed in executed functions. If a modified code line matches a line obtained via dynamic or static analysis, the function that is dependent on the hit line is tagged as having been affected by code change. This massive filtering step having been completed, the typically large set of

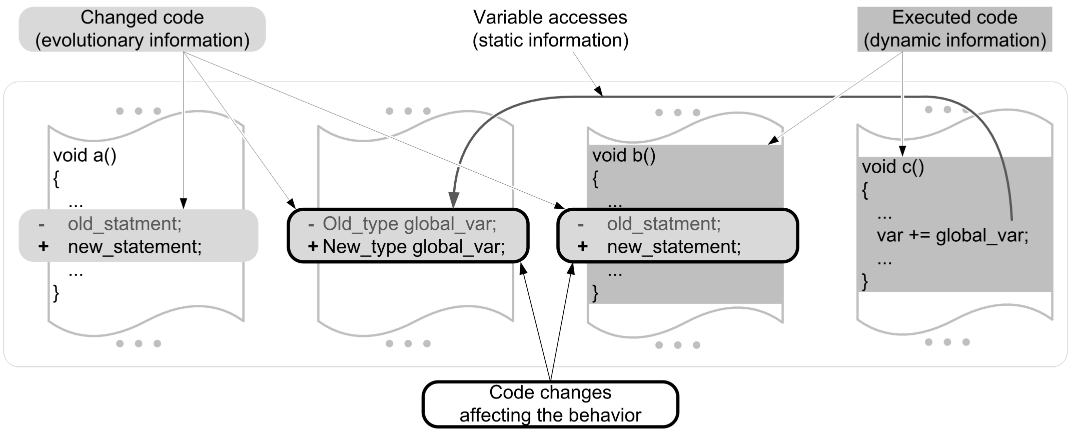


Figure 7.3: For detecting behavior-affecting code changes, both code of executed functions and code defining variables accessed from executed functions have to be checked for modifications.

executed functions is reduced to a small set that only contains the affected functions. Our experience when applying the filtering step on industrially developed software systems with the intention of solving real-world problems indicates that the filtering reduces the set of functions massively in a typical case.

7.2.5 Applying Trace Visualization - Exploring Functions within their Execution Contexts

Within the set of functions affected by code changes, the developer needs to pinpoint the one that is responsible for the observed change in system behavior. For this, multiple views on the trace are provided as discussed in Chapter 6. With the views the developer can explore how control flow passes through the implementation while the faulty behavior is executed. Developers can recognize the context in which a changed code location is executed.

- The temporal overview and the phases view reveal which phases are affected by a code change. During execution of the system behavior, it is typical for different phases to be passed through. Figure 7.4 shows the temporal overview of a trace taken from an industrial software system for large-scale terrain visualization and illustrates the various phases that are performed. Phases may, for instance, be a startup-phase, a phase where specific data is imported, or a phase where a specific calculation is performed. Knowing the phases that are affected by a code change helps developers to assess the change's impact and decide which change needs to be analyzed in detail first and is to be given top priority.
- The call stack view reveals what happens along the control flow after modified code has been executed. The opportunity to analyze the function call stack at the time when modified code is executed and to discover which functions are executed afterwards gives an understanding of the purpose of the modified code. The call neighborhood view shows the structural control dependencies of a modified function (Figure 7.5). In addition, having access to the functions' source code enables developers to understand the purpose of the data that the modified code operates on.

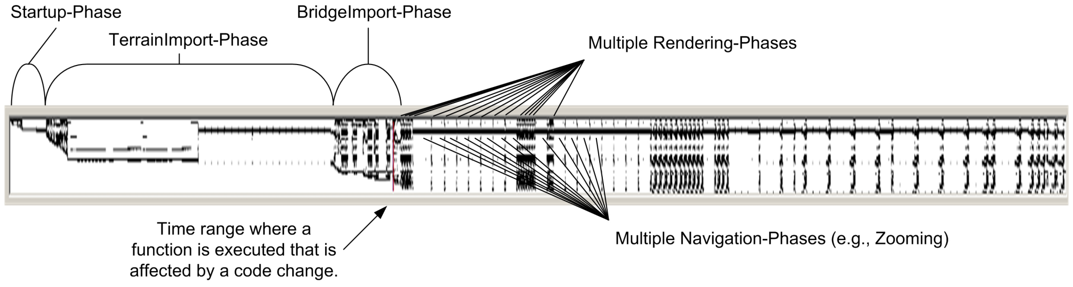

Startup-Phase
BridgeImport-Phase
TerrainImport-Phase
Multiple Rendering-Phases
Time range where a function is executed that is affected by a code change.
Multiple Navigation-Phases (e.g., Zooming)

Figure 7.4: The temporal overview depicts function activity over time and permits the detection of phases that are exercised while the system runs its faulty behavior.

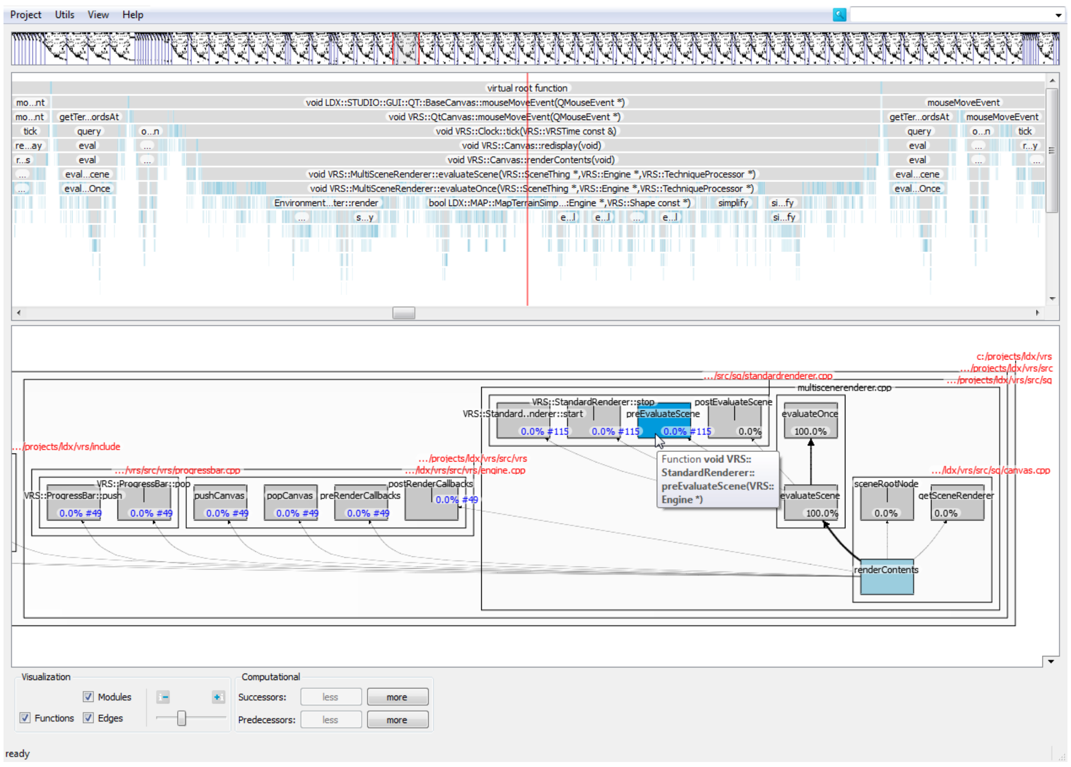

Project
Utils
View
Help
virtual root function
void LDX::STUDIO::GUI::QT::BaseCanvas::mouseMoveEvent(QMouseEvent *)
void VRS::QtCanvas::mouseMoveEvent(QMouseEvent *)
void VRS::Clock::tick(VRS::VRSTime const &)
void VRS::Canvas::redisplay(void)
void VRS::Canvas::renderContents(void)
void VRS::MultiSceneRenderer::evaluateScene(VRS::SceneThing *,VRS::Engine *,VRS::TechniqueProcessor *)
void VRS::MultiSceneRenderer::evaluateOnce(VRS::SceneThing *,VRS::Engine *,VRS::TechniqueProcessor *)
mouseMoveEvent
.../src/sg/standardrenderer.cpp
multiscenerenderer.cpp
VRS::StandardRenderer::stop
VRS::Standard...nderer::start
preEvaluateScene
postEvaluateScene
evaluateOnce
evaluateScene
Function void VRS::StandardRenderer::preEvaluateScene(VRS::Engine *)
.../projects/ldx/vrs/include
.../vrs/src/vrs/progressbar.cpp
.../ldx/vrs/src/vrs/engine.cpp
.../ldx/vrs/src/sg/canvas.cpp
VRS::ProgressBar::push
VRS::ProgressBar::pop
pushCanvas
popCanvas
preRenderCallbacks
postRenderCallbacks
sceneRootNode
getSceneRenderer
renderContents
Visualization
Modules
Functions
Edges
Computational
Successors:
Predecessors:
less
more
ready

Figure 7.5: Trace visualization helps developers to understand the control dependencies of a recently modified function. Trace visualization reveals what happens before and after the function's execution.

CHAPTER 8

CGA - A Trace Visualization Framework

The trace visualization concepts proposed in this thesis are implemented within a software framework called CGA[1]. It comprises the following functionalities:

- Extracting dynamic, static, and evolution facts from implementations.
- Integrating the facts in a combined data model and storing them in a common database.
- Providing a variety of generic trace visualization views.
- Providing several maintenance-specific views on dynamic, static, and evolution facts.
- Composing views as trace visualization tools.

8.1 Functional Decomposition

CGA is implemented in C++ and comprises approximately 130.000 lines of code. As shown in Figure 8.1, it is logically decomposed into two parts: EXTRACTION and ANALYSIS. Runtime components built from the two modules communicate by means of the shared data exchange format for extracted facts and by using a common network communication protocol.

8.1.1 Extraction Module

The EXTRACTION module contains functionality for tracing function calls in C/C++ systems by means of compiler-based instrumentation (cf Chapter 4). This mechanism called *callmon* is implemented in connection with the Microsoft Visual Studio Compiler

1 CGA is an abbreviation for call graph analyzer. Analyzing call graphs is what CGA was first implemented for. This name is now outdated because the functionality of the CGA framework meanwhile comprises a greater degree of functionality than only analyzing call graphs. For reasons of consistency with literature, the name is still used.

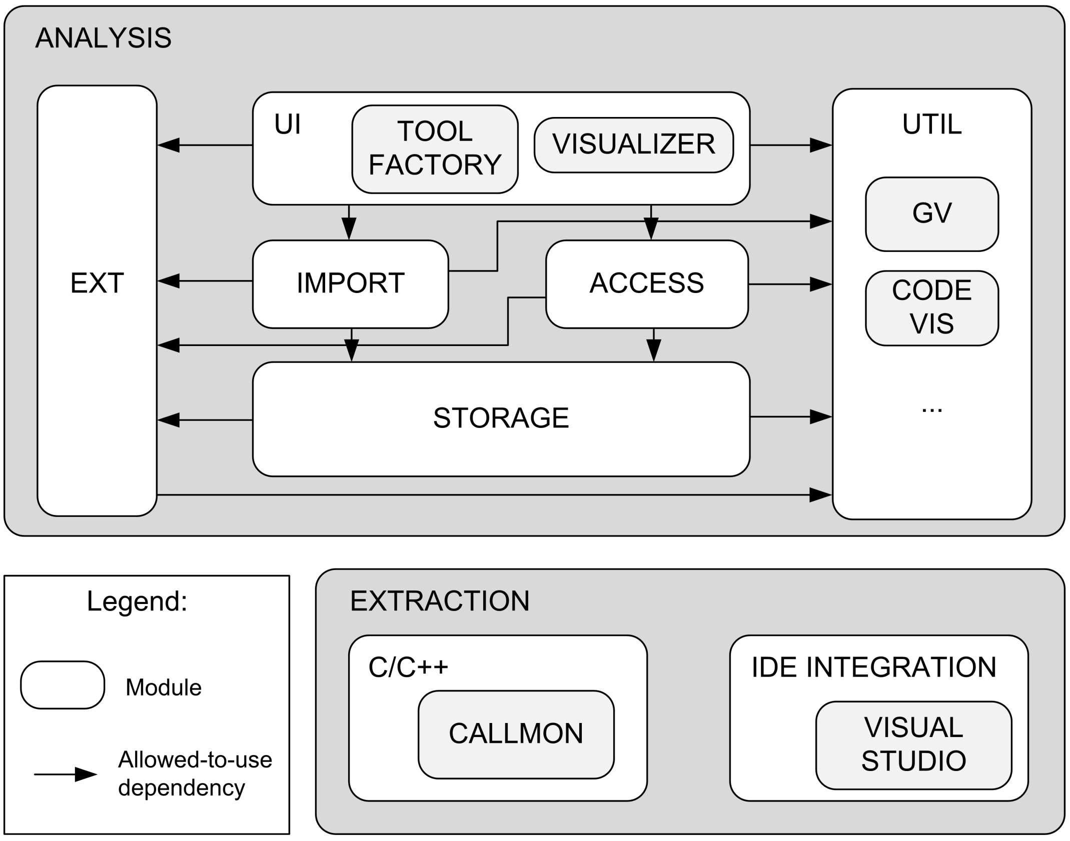


Figure 8.1: The functional decomposition and layered architecture of CGA.

for systems running on Microsoft Windows platforms and for the GCC for systems running on Linux or MacOS platforms.

Furthermore, EXTRACTION contains an IDE integration for the Microsoft Visual Studio IDE. The features of the IDE integration include:

- Configuring callmon, i.e., specifying the functions to be traced.
- Providing a *record start and stop* user interface for activating and deactivating callmon tracing at runtime.
- Remote controlling the Visual Studio Debugger, which facilitates tracing variable states and memory accesses in addition to the control flow information captured by callmon.
- Remote controlling the analysis and visualization back-end (i.e., the runtime components created from the ANALYSIS module):
  - A CGA project is automatically created based on the setting of a Visual Studio solution.
  - Traces are automatically imported and visualized in CGA after the developer has stopped callmon tracing.

8.1.2 `Analysis` Module

Layered Modules

`ANALYSIS` is organized as a layered architecture. The `STORAGE` layer is accessed by the `IMPORT` and `ACCESS` layers. These in turn are accessed by the `UI` layer. Within `UI`, the `VISUALIZER` module contains views on the data stored in `STORAGE` and accessed via `ACCESS`. The module `TOOLFACTORY` within `UI` contains code that uses elements from `VISUALIZER` to build tools for supporting maintenance tasks.

The `IMPORT` module supports the import of facts from a variety of sources:

- Dynamic facts:
  - Control flow tracing with *callmon*.
  - Variable state and memory access tracing via IDE integration.
  - Statement coverage using the Intel's instrumentation framework *PIN* [124].
- Static facts:
  - Facts and fact relations revealed by the static analyzer *doxygen* [157]. This includes namespaces, classes, class relations such as inheritance, functions, global variables, variable and attribute accesses, directories, files, file include relations, and many more. Additionally, code metrics are available on the facts.
- Evolution facts:
  - Code change history and check-in related meta data provided by the software configuration management system *subversion* [170].

Vertical Modules

`UTIL` is a "vertical" module that provides common functionality that can be used by all other modules. It contains, for instance, a graph visualization engine `GV` and a code visualization engine `CODEVIS`. The latter depicts code with embedded dynamic facts. Additionally, `UTIL` provides convenience functionality such as an API for working with XML data.

The graph visualization engine can be customized and instantiated by many visualization components in `UI::VISUALIZER` to implement their fact presentation logic. The features of `GV` include[1]:

- Visualization of clustered graphs, i.e., graphs consisting of nodes and directed edges with nodes being additionally organized in a tree of clusters.
- Customizable shapes of nodes, edges, and clusters and mapping of meta data on nodes, edges, and clusters on visual attributes.

1 A detailed description of some core features of `GV` is found in [220].

- Customizable layouts; providing a set of default 2D layouts such as variants of sugiyama layouts [204], energy based layouts [154], and treemap layouts [191].
- Underlying 3D rendering engine:
  - Fast image creation by exploiting GPU power.
  - Advanced rendering techniques exploiting the GPU rendering pipeline.
  - Switching between orthogonal projections that result in images that appear "pure" 2D and perspective projections that result in landscape-like $2\frac{1}{2}$D presentations.
- Navigation techniques that operate both in 2D and $2\frac{1}{2}$D such as zooming, panning, tilting, and viewpoint animations for automated focusing on nodes, edges, or clusters [25, 26].
- Build-in picking mechanism to react on mouse clicks on node, edge, and cluster shapes.
- Advanced labeling techniques that annotate nodes, edges, and clusters in such a way that labels do not overlap in screen space (cf [125, 126]). Level-of-detail technique that adapts the label text to the screen-space size of the shape to which the label is attached.
- Animated morphing from the currently shown graph to a new graph as input data.

The second vertical module `EXT` in `ANALYSIS` is the "gateway" to external tools, i.e., third-party libraries and third-party processes. It encapsulates the external APIs and data formats to minimize the impact of changes in the third-party tool. In the case of a change, only the corresponding `EXT` module is affected.

8.2 Performance Measurements

To evaluate the proposed trace visualization concept, `CGA` is applied to real-world software systems. Performance measurements show that the concept—even in its prototypically implemented form—applies to complex software systems and comes to terms with scalability issues that trace visualization frequently encounters. The measurements were performed on a *Lenovo Thinkpad X200 Tablet* notebook with an *Intel Core 2 Duo CPU L9400@1.86GHz*.

8.2.1 Performance Overhead with Deactivated Tracing

The proposed tracing technique initially applies compiler-based instrumentation and then neutralizes it by replacing the instrumentation code with `NOP` assembler instructions. This approach enables developers to activate tracing at runtime with standard debugger facilities in a highly robust way.

However due to the `NOP` instructions, the resulting binary code differs from the one built without instrumentation. One important requirement of any trace visualization tool is that it should not hinder the developers from performing their usual development processes. As the tracing technique is integrated into the usual build process, the runtime overhead of the tracing technique needs to be negligible during the usual development process, i.e., if tracing is disabled. To measure the effect of additional `NOP` instructions, we apply the instrumentation technique on the 130kLOC C/C++ software system *brec* of our industrial partner virtualcitySYSTEMS GmbH. One feature of the system is to reconstruct 3D building models from point clouds that were obtained by laser scanning (LiDAR). We apply the instrumentation technique onto the release build configuration of *brec*. In the release build, several functions are inlined or optimized by the compiler which reduces the amount of functions in the binary code, i.e., binary code units that are entered via an assembler `call` instruction. However, 8350 functions are still contained in the binary code and are therefore traceable.

The system functionality being chosen for the measurements is the 3D reconstruction of the first building of a small test data set. Without instrumentation, the execution takes on average 20 seconds. In our experiment, we measure—for multiple runs—the amount of processor ticks consumed during the reconstruction. Ticks are measured using the `RDTSC` processor instruction. Without instrumentation, an average of 36.761.000.000 ticks is measured. With instrumentation, i.e., with additional `NOP` instructions, we obtain an average of 37.407.000.000 ticks. Figure 8.2 shows the histogram of our measurements. As illustration of the average ticks values, the histogram deviations are fitted with gaussian curves. In this specific execution scenario, the performance overhead due to the `NOP` instructions was approximately 1.7%. Feedback received from our industrial partners indicates that such a small performance overhead is acceptable and does not impede the development process.

To calculate the performance overhead per function call, we next measure the amount of calls that are executed during building reconstruction. Therefore, all functions are activated for tracing. That is, the `NOP`s are replaced by the `call` instructions that were originally inserted by compiler-based instrumentation. Our event collecting library counts 26.196.226.749 calls. Hence, the performance overhead per call due to the `NOP` instructions can be estimated as $\frac{37.407.000.000-36.761.000.000}{26.196.226.749}\frac{\text{tick}}{\text{call}} = 0.02\frac{\text{tick}}{\text{call}}$.

To determine the runtime overhead that is introduced by the event collecting mechanism, we divide the time (in ticks) for capturing the call entry and exit events by the total number of events. Recording the trace of 26.196.226.749 calls took 46.496.000.000.000 ticks, which corresponds to approximately 7 hours. To solve the problem that serializing the trace would require an extraordinarily large amount of disk space, we write events of each event buffer only temporarily onto hard disk and free up the used space again afterwards. Provided that we need 20 Bytes for each event, we would need 1000 Terra(!) bytes of disk space for the ≈53 billion events if being stored in a raw format. The event registration overhead per event is $\frac{46.496.000.000.000}{2*26.196.226.749}\frac{\text{tick}}{\text{event}} \approx 890\frac{\text{tick}}{\text{event}}$. This measurement show that the implementation of the event registration mechanism is indeed prototypical—it has not been optimized

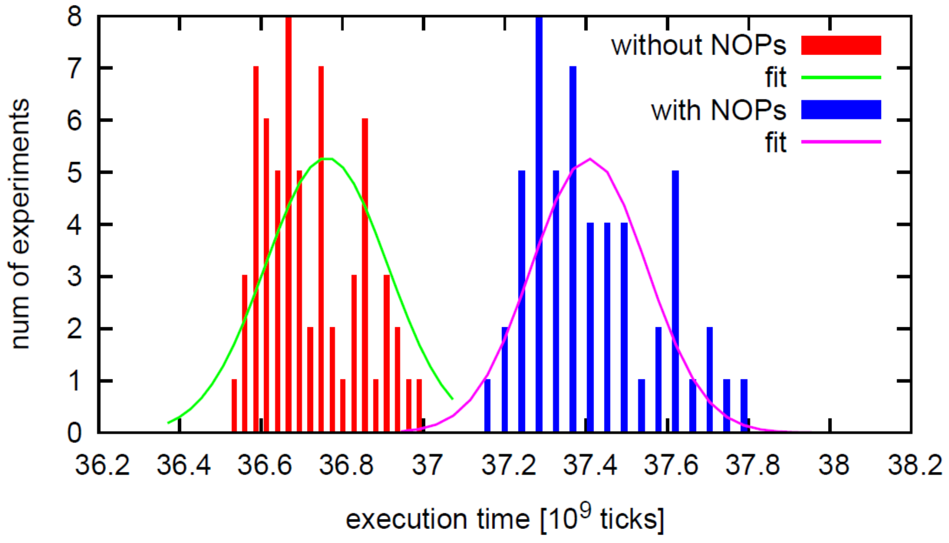


Figure 8.2: Histogram of the performance measurements with and without `NOP` instructions. The experiments were performed multiple times with the building reconstruction feature of the *brec* software system of virtualcitySYSTEMS GmbH.

for performance. However as shown in the next section, by applying the technique for deactivating massively called functions, the large overhead per event registration does not imply a large overall overhead during tracing.

8.2.2 Detecting and Excluding Massively Called Functions

The technique for detecting and excluding massively called functions during tracing (cf Chapter 4) offers an effective means of reducing the amount of captured function calls at tracing-time, i.e., when events are captured in a buffer in memory and before they are serialized to disk. In this section, we apply the technique during tracing multiple features of various industrially developed software systems. Thereby, different values for the massively called function threshold ν_{max} are used. To be able to reproduce the execution scenarios so that the effect of different threshold values ν_{max} for the detection of massively called functions can be compared, we exploit the fact that our instrumentation technique permits the activation of tracing at runtime by replacing binary code with standard debugging facilities. All execution scenarios are traced performing the following steps:

1. Execute the software within the Microsoft Visual Studio debugger and stop it via a breakpoint at the point in execution where tracing should be started.
2. Activate tracing by binary code instrumentation.
3. Execute and trace until a previously set *stop* breakpoint is hit.
4. Deactivate tracing by binary code instrumentation and examine the trace.

Software system: brec | Company: virtualcitySYSTEMS GmbH

The characteristics of the software system *brec* are given in the previous section. For the experiment, the same execution scenario as in the previous section is analyzed: reconstructing the first building in the test data set. However, this time we use the debug build of the system. Running this execution scenario with the debug build takes on average 37.4 seconds.

Before tracing, we exclude all compiler generated functions and all functions from header files of 3rd party libraries such as the C++ standard template library (STL). 2754 functions remain activated for tracing in all executed binary files, i.e., the executable file and the dll files it depends on. Without applying the technique for detecting and excluding massively called functions, the trace comprises 2.300.000.000 calls. The execution time of this run with only temporarily serializing the call entry and exit events to hard disk (cf previous section), takes 91 minutes. Hence, recording all call events slows down the execution by a factor of 145.

Next, we apply the technique for detecting and excluding massively called functions. The time window Δt is set to 100.000.000 ticks. Table 8.1 shows the tracing characteristics for multiple executions with different values of the massively called function threshold ν_{max}. The execution time reported here includes the time for deactivating massively called functions, i.e., stopping the execution and replacing `call` with `NOP` instructions. Hence, for decreasing ν_{max} values an increasing amount of time is spent for deactivating the increasing number of massively called functions. Our prototypical tool analyzes and serializes the collected events in the in-memory buffers every second. During massively called function deactivation, which takes ≈0.5s, all threads of the analyzed software system are suspended. Hence, the time value reported in Table 8.1 includes two kinds of performance overheads: (1) There is an overhead that results from the time delay between collecting events, detecting massively called functions, and deactivating them. Hence, there are many events corresponding to massively called functions first collected—with overhead for event collection and serialization—before this source of overhead is deactivated. (2) There is an overhead during the deactivation process itself: To prevent changing the binary code that is currently executed by a thread of the analyzed software system, all threads are suspended while changing the binary code.

ν_{max}	#Calls	#Massively called funcs	Execution time
∞	2.300.000.000	0	5460s
200	65019	179	58s
150	55657	188	56s
100	38925	222	51s
50	20211	256	48s
25	13093	289	48s
12	5829	325	47s

Table 8.1: Tracing the *brec* software system with different values of ν_{max} results in traces with varying size and varying performance overhead.

To illustrate which functions are detected as massively called functions, Table 8.2 provides a list of excluded functions together with their ν values.

ν	Function
6174	bool SweepInfo::operator<(SweepInfo const &)
5797	double Constellation::Determ(double,double,double,double,double,double)
5110	double Vector3T<double>::operator[](int)const
4819	bool Vector3T<double>::operator==(Vector3T<double> const &)const
4701	std::vector<SamplePoint,std::allocator<SamplePoint> > & Cell::SamplePoints(void)
4071	std::vector<Ring *,std::allocator<...> > const & GroundPlan::InnerRings(void)const
3967	Face * CSG::MergeFaces(Face *,Face *)
3864	double BuildingReconstructionProcess::Edge::EdgeDist(DPoint)
3614	bool IsPointInPolygon(Vector3T<double> const &,Face const *)
3606	double GetPixelFloat(IMAGE_PTR,double,double)
3503	DPoint & DPoint::operator-=(DPoint const &)
3503	void RectHouse::toLocal(DPoint &)
3403	Vector3T<double> SamplePoint::Position(void)const
3388	Triangulation3D const * Face::Triangulation(void)const
3162	IMAGE_PTR RecBuilding::show_dgmImage(void)
2796	Ring * Polygon::OuterRing(void)
2693	Vector2T<double> BoundingRectangle::Min(void)const
2535	std::vector<Vector2T<double>,std::allocator<...> > & Cell::Points(void)
2459	bool Constellation::SetPixel(int,int,bool)
2459	bool IsrCnstlSetPixel(KBV_CONSTELLATION *,int,int,int)
2431	double & Vector3T<double>::operator[](int)
2352	double Line::distance_to(Vector3T<double> const &)const
2250	Face * CSG::MergeConvexFaces(Face *,Face *)
2161	float MinVec(float *,int)
2046	double dotProd(DPoint const &,DPoint const &)
1963	BuildingReconstructionProcess::Edge::Edge(DPoint,DPoint)
1930	BuildingReconstructionProcess::Edge::Edge(void)
1927	DPoint operator+(DPoint const &,DPoint const &)
1904	Vector3T<double>::~Vector3T<double>(void)
1745	DPoint * List<DPoint>::nextPtr(void)
1666	double CellDecomposition::LineBuffer::Significance(void)const
1554	IMAGE_PTR RecBuilding::show_dhmImage(void)
...	...

Table 8.2: Due to their high ν values, the listed functions are classified as massively called functions and excluded from tracing.

Figure 8.3 shows a screenshot of the prototypical trace visualization tool and illustrates that a trace cleaned from massively called functions still contains the information to understand how the higher-level functionality of building reconstruction is implemented in the *brec* software system.

Software system: Google Chrome | Company: Google Inc.

The *Google Chrome* webbrowser is an open-source software system primarily developed by Google Inc. The code comprises 4 million lines-of-code. Thereof, 1.5 million code lines are written in C and C++. We instrument the *Chrome* executable of the debug configuration. 259145 different functions are contained in the executables (one

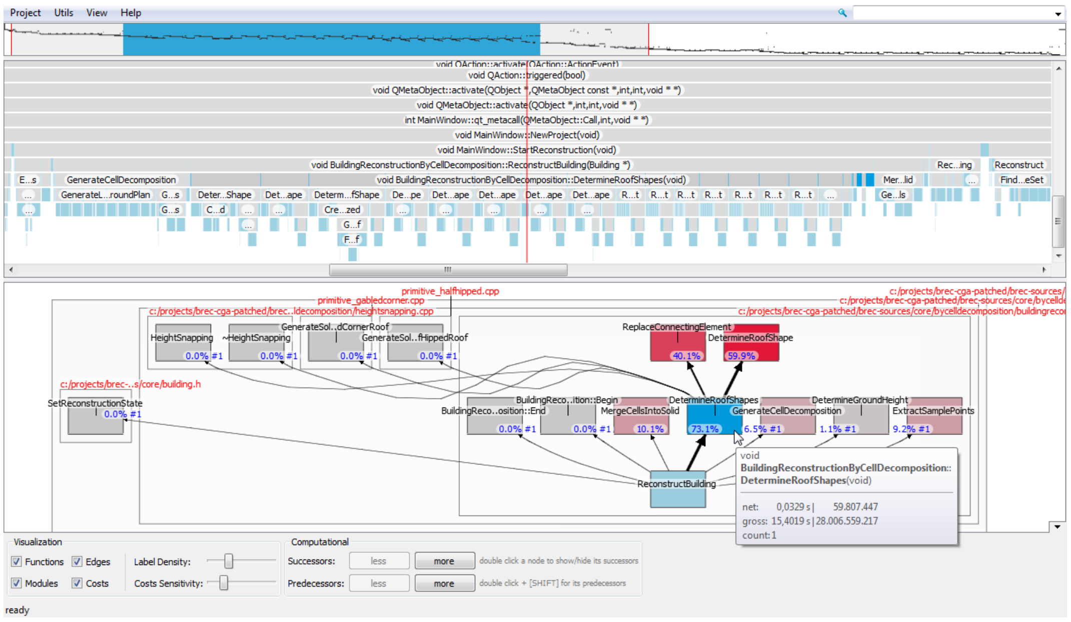

Project
Utils
View
Help
void QAction::triggered(bool)
void QMetaObject::activate(QObject *,QMetaObject const *,int,int,void * *)
void QMetaObject::activate(QObject *,int,int,void * *)
int MainWindow::qt_metacall(QMetaObject::Call,int,void * *)
void MainWindow::NewProject(void)
void MainWindow::StartReconstruction(void)
void BuildingReconstructionByCellDecomposition::ReconstructBuilding(Building *)
void BuildingReconstructionByCellDecomposition::DetermineRoofShapes(void)
GenerateCellDecomposition
primitive_halfhipped.cpp
primitive_gabledcorner.cpp
c:/projects/brec-cga-patched/brec...ldecomposition/heightsnapping.cpp
HeightSnapping
~HeightSnapping
GenerateSol..dCornerRoof
GenerateSol..fHippedRoof
ReplaceConnectingElement
DetermineRoofShape
40.1%
59.9%
c:/projects/brec-...s/core/building.h
SetReconstructionState
BuildingReco..ition::Begin
BuildingReco..osition::End
MergeCellsIntoSolid
DetermineRoofShapes
GenerateCellDecomposition
DetermineGroundHeight
ExtractSamplePoints
10.1%
73.1%
6.5% #1
1.1% #1
9.2% #1
ReconstructBuilding
void
BuildingReconstructionByCellDecomposition::
DetermineRoofShapes(void)
net: 0,0329 s | 59.807.447
gross: 15,4019 s | 28.006.559.217
count: 1
Visualization
Functions
Edges
Modules
Costs
Label Density:
Costs Sensitivity:
Computational
Successors:
Predecessors:
less
more
double click a node to show/hide its successors
double click + [SHIFT] for its predecessors
ready

Figure 8.3: A screenshot of the trace visualization tool illustrating that a trace cleaned from massively called functions still contains the information to understand how the higher-level functionality of building reconstruction is implemented in the *brec* software system.

executable and one dll file) and a can, hence, be activated for tracing.

For our measurements we examine the scenario of rendering the homepage[1] of Google, which involves 193.804 function calls and takes 0.015 seconds without tracing. We use the feature of setting breakpoints in a normal debugger to concisely define the part of execution to be traced.

Table 8.3 summarizes both resulting trace sizes and performance values for different threshold values ν_{max} for the detection of massively called functions.

ν_{max}	#Calls	#Massively called funcs	Execution time
∞	193804	0	2.3s
200	40672	18	1.7s
150	30942	20	1.7s
100	28974	26	1.8s
50	21519	36	1.8s
20	8273	66	1.8s

Table 8.3: Traces captured while *Google Chrome* renders the Google homepage. Traces are taken with different threshold values for detecting massively called functions.

1 `http://www.google.com`

Software system: Blender | Company: Not A Number B.V.

The *Blender* software system developed by Not A Number B.V. and the Blender Foundation is a tool for creating, modeling and rendering 3D content. The system comprises 460.000 lines of C code. After instrumentation, 50803 functions are active for tracing. The analyzed execution scenario is *adding a monkey geometry shape to the 3D scene*. It takes 4.05 seconds to execute without tracing. Without using the technique for detecting massively called functions results in a trace consisting of 371.623.833 function calls. With the technique, a significant reduction of trace size and performance overhead can be achieved. Table 8.4 summarizes the reductions given different threshold values ν_{max}.

ν_{max}	#Calls	#Massively called funcs	Execution time
240	5.361.829	82	158s
200	4.278.518	88	127s
150	2.705.097	112	85s
100	2.291.174	151	78s
50	414.330	277	14s
10	138.757	543	12s

Table 8.4: Characteristics of traces captured during execution of the *Blender* software system.

Software system: LandXplorer | Company: 3D Geo GmbH (now Autodesk Inc.)

The *LandXplorer* software system by 3D Geo GmbH, which has been integrated into Autodesk Inc., is a software solution for processing and visualizing large geodata sets. It comprises 1.1 million lines of C++ code. After instrumentation, 504432 functions are active for tracing. We examine the scenario of loading a specific terrain data file into a LandXplorer project. This execution scenario comprises 621.162.948 calls and takes 3.1 seconds to execute without tracing. Table 8.5 shows the characteristics of captured traces for different applied threshold values ν_{max}.

ν_{max}	#Calls	#Massively called funcs	Execution time
200	41978	31	9.3s
100	36642	36	8.1s
36	21329	84	7.9s
5	6419	320	6.3s

Table 8.5: Characteristics of traces captured during execution of the *LandXplorer* software system.

CHAPTER 9

Case Studies related to Complex Software Systems

The concepts and tools for using trace visualization to facilitate maintenance tasks proposed in this thesis have been subject to evaluation in a number of case studies pertaining to complex software systems. They have been applied within real-world software engineering projects, in particular with regard to maintenance tasks in industrial software development. Their objective being to evaluate the strengths and weaknesses of trace visualization, case studies focus on singular instances of maintenance tasks. The case studies chosen do not intend to prove general improvement in developer performance—that would require stronger evaluation methods. These methods include controlled experiments and surveys. In this thesis, evaluation is restricted to case studies for two reasons. First, carrying out controlled experiments on trace visualization represents a scientific effort in itself; the effort necessary would have exceeded the scope of this thesis. Second, trace visualization tools that can be integrated into professional development processes are as yet not readily available as commercial products or academic prototypes. This thesis aims to bridge the gap between non-scalable, academic trace visualization tool prototypes and robust and scalable tools able to cope with real-world software systems. This thesis, therefore, puts forward concepts and implementations on how to build trace visualization tools that can be effortlessly and smoothly integrated into industrial software development processes enabling professional developers to reap the benefits of trace visualization in their daily work. Once trace visualization tools are adopted by the industry, long-term, in-depth examinations and surveys can be conducted that measure qualitatively and quantitatively the benefits of trace visualization. Only then will it be possible to carry out experiments that distinguish between the effect of the general availability of data on system execution and the effect presenting this data in a specific way. Almost any evaluation of a trace visualization tool only compares the application of the tool with a baseline situation in which developers had no access to the trace data at all.

This chapter provides case studies that were conducted with industrial partners. The studies examine the benefits of trace visualization for two types of maintenance tasks: (1) The maintenance task of locating an existing system feature in a large code base; (2) The maintenance task of locating a fault by means of applying the novel trace visualization based fault localization technique proposed in Section 7.2.

9.1 Visualizing Traces for Locating Features

The approach comprising use of trace visualization for feature location is demonstrated in a case study performed with virtualcitySYSTEMS GmbH. The analyzed software system *brec* is a tool suite for reconstructing 3D building models from point clouds obtained by laser scanning (LiDAR). The software is written in C++ consisting of 130kLOC in 334 source files. The system's main feature, the building model creation, is triggered when a user starts a new project in brec. Input data is: (1) a 2D map of building footprints and (2) a 3D point cloud (Figure 9.1). A crucial task during building model creation is extraction of the building's roof type from the point cloud. In this case study, a developer, who has little knowledge of brec's implementation, needs to extend the set of supported roof types.

9.1.1 Fact Extraction

To begin with, a trace is taken while brec creates building models for a small data set of 111 buildings. The trace log file is growing large rapidly (>8GB $\approx$ >380 million calls) and the computer is running into hard-disk capacity problems. Therefore, brec is stopped in the middle of execution and the functions contained in the trace are analyzed. 10 functions related to 3D vector operations are called with very high frequency. These functions are excluded from the tracing mechanism and brec is executed again, which results in a trace consisting of 520 functions and 30 million calls.[1]

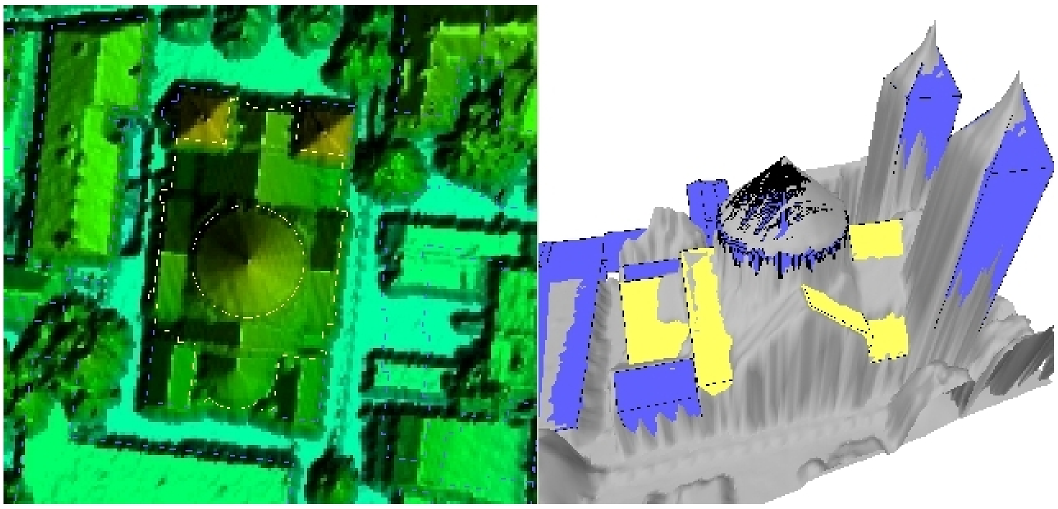


Figure 9.1: The *brec* software system of virtualcitySYSTEMS GmbH reconstructs 3D building models from point clouds.

1 The case study had been performed before the technique for automatically excluding massively called functions from tracing proposed in Chapter 4 was invented. Therefore, the exclusion identification and exclusion step is done manually here.

9.1.2 Fact Analysis - Applying the Trace Pruning Algorithm

To illustrate the complexity of the trace, Figure 9.2 depicts functions that are called from the root function in up to 4 successive call relations out of 15. The diagram does not show the sequential call order any more, i.e., multiple calls between the same two functions are merged into a single call relation. Visualizing calls explicitly would lead to a much higher cognitive load for the developer. The 10 call relations starting from the root function, for instance, would be replaced by 797 calls. Applying the trace pruning algorithm (threshold value T_{ntrig} is 2000; cf Section 5.2) would drastically reduce the graph's size.

9.1.3 Fact Presentation

Explicitly visualizing the call order gives further important insights into system behavior. Figure 9.3 shows the *phases view*—the phase similarity threshold value T_{sim} of the repetition detection mechanism is 0.3 (cf Section 5.3.3). The sequential order of calls reveals that first, ground plans are loaded; second, a digital height model (DHM) is created; and third, the reconstruction process is started. The reconstruction process itself is done per building (111 times). Reconstruction involves execution of the methods for building reconstruction of the classes `ReconstructionByCellDecomposition`, `ReconstructionByGroundPlanExtrusion`, and `BuildingReconstruction`. Unfolding these three steps reveals that each step contains a call that is responsible for determining roof types. The developer who needs to extend the set of supported roof types is now aware of three important code locations that need to be extended.

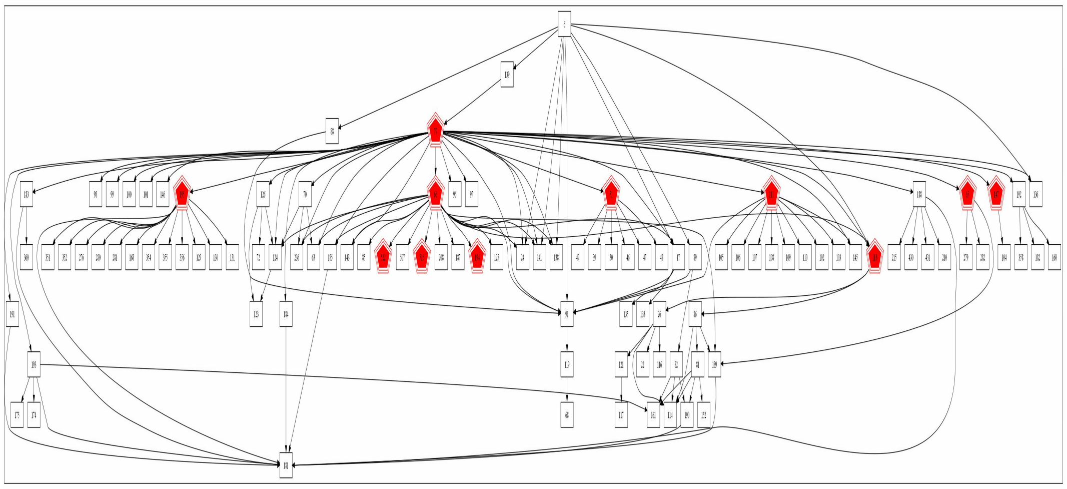


Figure 9.2: The first 4 levels of caller-callee relations of the "timeless" call graph illustrate the complexity of the trace taken from the *brec* software system. The trace pruning algorithm automatically identifies key calls in the trace. The respective functions are highlighted in the shown graph (red diamonds).

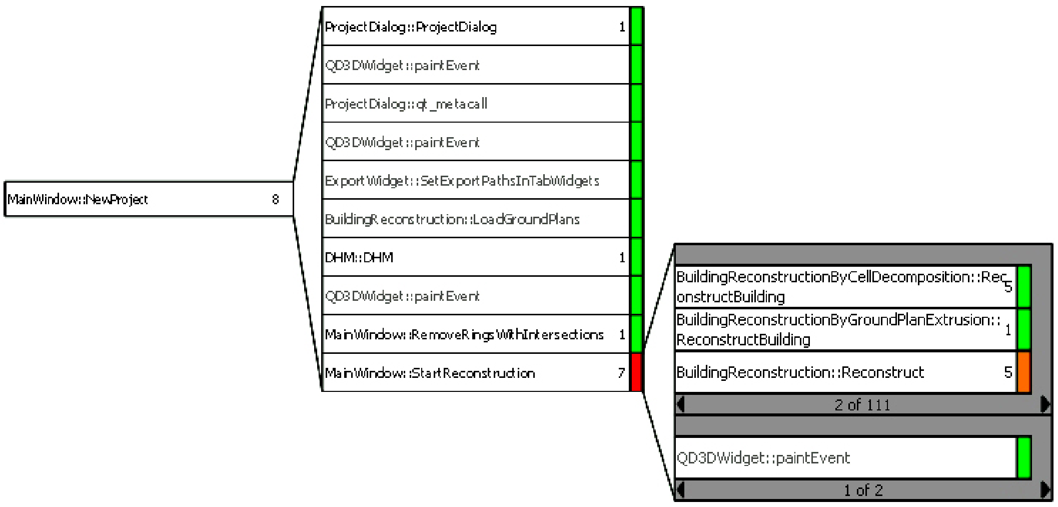

MainWindow::NewProject 8
ProjectDialog::ProjectDialog 1
QD3DWidget::paintEvent
ProjectDialog::qt_metacall
QD3DWidget::paintEvent
ExportWidget::SetExportPathsInTabWidgets
BuildingReconstruction::LoadGroundPlans
DHM::DHM 1
QD3DWidget::paintEvent
MainWindow::RemoveRingsWithIntersections 1
MainWindow::StartReconstruction 7
BuildingReconstructionByCellDecomposition::ReconstructBuilding 5
BuildingReconstructionByGroundPlanExtrusion::ReconstructBuilding 1
BuildingReconstruction::Reconstruct 5
2 of 111
QD3DWidget::paintEvent
1 of 2

Figure 9.3: The *phases view* reveals how the 111 buildings are reconstructed by executing 3 key functions for each building. Each of these functions calls a further function that is responsible for determining the roof type of the building.

9.1.4 Discussion on the Results

For the given task setting, trace visualization has shown itself to be a useful technique for rapidly identifying three code locations within the large code base that implement key functionality for the given maintenance task of extending the roof type recognition feature of the software system. In particular the phases view proved helpful when exploring the trace using a top-down approach. With it, task related parts contained in the trace were rapidly found. The overall time for applying trace visualization was approximately 15 minutes (excluding the nonrecurring initial preparation of the software system for being instrumented and the initial rebuild of the system – cf Chapter 4). The following numbers obtained with *grep*, i.e., with regular expression matching on the source code, convey a feeling of the complexity of the given feature location task: The term `roof` is found 1.713 times in 349 different source code files; `rooftype` is found 314 times in 180 different files.

As mentioned at the beginning of this chapter, this case study does not offer proof that trace visualization outperforms other feature location techniques in general. What this case study does indicate is that exploiting the sequential runtime information contained in a trace can be very helpful in an industrial setting. Moreover, what remains to be examined as follow-up research on this thesis, is the quantitive influence of different trace presentations on the developer's feature location task performance. With regard to this issue, it is absolutely certain that without a scalable way of visualizing trace data, the value contained in the data remains inaccessible. In addition, the case study gives evidence that developers need to be provided with trace visualization techniques that depict the sequential information stored in a trace. However, the techniques need to create and present trace abstractions on various levels of abstraction, as implemented by the phases view.

9.2 Visualizing Traces for Identifying Recently Introduced Faults

Two case studies have been performed with 3D Geo GmbH, which has been integrated into Autodesk Inc. One of the main products represents LandXplorer Studio Professional (LDX), a software solution for processing and visualizing large geodata sets (e.g., landscape and 3D city models). The LDX code has been developed over more than 12 years with currently significantly more than 25 developers. LDX is written in C++ consisting of ≈1.100.000 SLOC-P (7.000 source files). We carried out the case studies by accompanying the developers during their fault localization activities. In the given task settings the developers encountered failures in the software system that had been introduced recently into the code base by other developers (or unconciously by themselves); the system had previously (e.g., before two weeks) been behaving correctly. In the case studies, trace visualization is used as explained in Section 7.2.

9.2.1 Fault: "Why does the bridge become invisible?"

Figure 9.4 illustrates the barely detectable failure that a developer had noticed: When zooming away from a specific building model (the bridge), some parts of it become invisible. The developer knew that approximately one month earlier the bridge had been visualized correctly. To locate the fault in the code, the trace visualization based fault localization technique described in Section 7.2 was applied: First, a trace was recorded while the developer executed LDX and reproduced the failure. The trace consisted of ≈50 million function calls and 1.724 different functions. Then, code modifications within the last month were extracted from the software configuration management (SCM) *subversion*. Within this time period, 273 check-ins had been performed that modified 2.530 contiguous code blocks in 650 files.

In a third step, the static analysis tool *doxygen* is used to identify variable accesses from within functions. After gathering this data, it was combined to reveal those functions that had been both affected by a recent code modification and were executed during system execution. The combination step revealed that only 11 functions matched the two criteria.

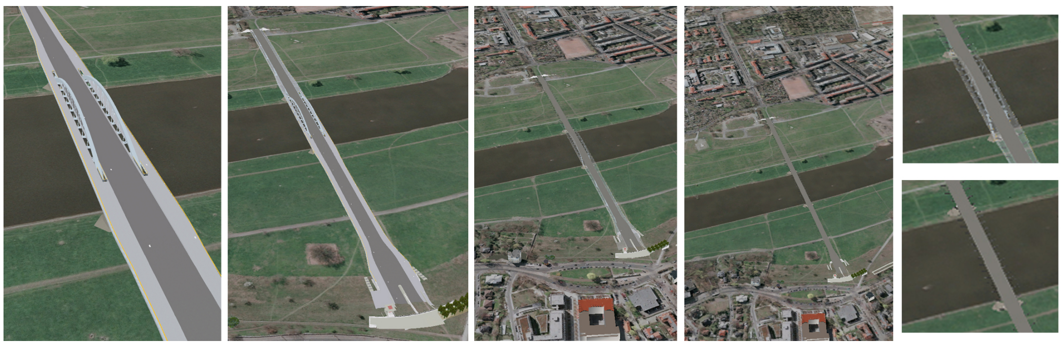


Figure 9.4: A developer detects a subtle failure in the *LandXplorer* software system that did not exist a month earlier: The bridge becomes partly invisible when zooming away from it.

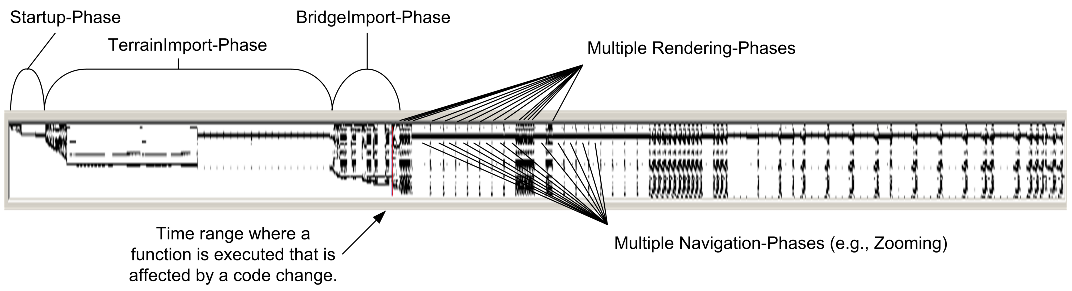

Startup-Phase
TerrainImport-Phase
BridgeImport-Phase
Multiple Rendering-Phases
Time range where a function is executed that is affected by a code change.
Multiple Navigation-Phases (e.g., Zooming)

Figure 9.5: The *temporal overview*, which depicts function activity over time, makes it possible to detect those phases that are exercised while the system is running its faulty behavior.

Next, the developer needed to identify which of the 11 functions was responsible for the failure. However, before directly turning to the source code, trace visualization was used to understand the execution context of the functions. Figure 9.5 shows the *temporal overview* (cf Section 6.2.1) of the trace. A visual marker in the temporal overview indicates the time ranges during which a function is active that is selected in another view. The developer was able to reveal the exact phases of the execution during which the affected functions appeared. This led the developer to be able to analyze first of all details of the one affected function executed during the bridge model import phase. Eventually it was discovered that this was the place where the fault had been brought into the system. Results showed that a coupling between a modification of the texture compression mechanism and the bridge model import mechanism had been responsible for the failure.

9.2.2 Fault: "Why is the Terrain Wizard missing?"

The second case study describes the application of the approach in a situation where a developer had noticed that a specific wizard, the *terrain loading wizard*, had stopped being shown during the import process for a specific type of terrain data. One week earlier, the wizard had still correctly showed up during terrain loading. A trace capturing the failure contains ≈200.000 calls and 1.674 different functions. During the past week, 9 developers had performed code modifications on 65 different source code files. Applying the data combination step reduced the 1.674 functions of the trace to 3 that were being affected by recent code modifications. Visually analyzing where the 3 functions were active in the trace revealed that one function was concerned with initializing the terrain data. One function was called when settings were being defined as to how the terrain was to be visualized. The third function was called shortly after user interaction for loading the terrain had taken place. Code modifications within this third function turned out to be responsible for the missing terrain loading wizard.

9.2.3 Discussion on the Results

The case studies described in this section provoke the following conclusions: (1) In the case of long-living, large, industrial software systems, often subtle couplings between structurally separated code elements exist that developers of the system are not aware of. These couplings lead to situations where a developer alters system behavior without noticing it. The failure is often noticed by other developers or testers after days or weeks. (2) Combining information on code modifications with information on executed functions while the system exhibits the failure can be of significant help in identifying the cause-effect chain, i.e., the fault-failure relation. (3) Even assuming that the fully automated step of combining change information with runtime information proves successful in reducing the amount of functions to inspect, trace visualization provides a useful means of understand the execution context of the functions. Trace visualization is especially helpful when deciding which functions in question to analyze in detail first. Analyzing in detail thus means that a developer analyzes more and more fine-grained trace visualizations and finally ends up reading source code.

The conclusions made on the basis of these case studies in no way wish to make any claim on the general superiority of the fault localization technique introduced here. However, it has been applied in typical industrial collaborative software development settings, which means that there is evidence that this approach can be recommended as a useful approach to be applied when endeavoring to localize faults in similar software projects and maintenance situations. However, precise statistics on the effectiveness of this approach as compared with other techniques require long-term studies to be undertaken. Follow-up research on this thesis must therefore be to (a) further improve this technique in a way that it can be effortlessly integrated into a developer's working process; (b) evaluate the effectiveness of the approach in the field. Currently, the approach would not be adopted by industry as the data collection step is still too time-consuming. Runtime information can be obtained without any waiting time, but the necessary static analysis is slow. Performing lightweight static analyis with *doxygen* on the *LandXplorer* system takes longer than 1 hour. One feasible way of solving this problem would be by developing a server solution for monitoring code check-ins into the SCM. The server would be expected to perform a static analysis after each checkin and provide results instantaneously on demand as soon as a developer notices a failure and wants to apply the fault localization technique.

CHAPTER 10

Summary and Outlook

10.1 Summary

Understanding the structure and behavior of software systems is essential for their effective maintenance. Modifying the implementation of a software system without sufficient understanding is likely to introduce design anomalies and might cause the implementation to degrade, which in turn complicates any understanding of the system [166]. With the aim of making comprehension of a software system's structure more accessible, a wide range of concepts have been proposed which have already been implemented as robust, industry-ready reverse engineering tools. Such tools support developers in their tasks of creating structural higher-level abstractions from the source code (e.g., by creating class diagrams from the code). At the present time there are hardly any trace visualization tools available as commercial products that provide an understanding of system behavior in a way that shows the temporal order of how structural elements interact at runtime. In academic research relating to trace visualization, a variety of concepts have meanwhile been put forward for understanding runtime behavior. However, their prototypical tool implementations are still far from being adopted by industry at large [6]. The major obstacle preventing transition from concepts to usable tools is the scalability issue [232].

Due to the pipeline structure of the trace visualization process, any underlying concepts of a trace visualization tool need to be scalable. In other words, they need to be able to process a large amount of trace data in a user-acceptable time. That is, there should be almost no waiting time to annoy developers before they can start using the trace visualization tool. It should take less than a few minutes to obtain the first visualization results. If trace visualization takes significantly longer, developers will tend to reject the use of trace visualization tools and try to solve the task in hand by resorting to standard techniques such as code reading or stepping through the execution with a symbolic debugger. The scalability issue is not only concerned with computational aspects, where time or space complexity of algorithms becomes too large to be handled by a computer, scalability is also concerned with the cognitive limitations of developers.

This thesis aims to overcome scalability bottlenecks within trace visualization concepts. The intention behind the proposed concept is to move the technique of trace visualization one step closer to a situation, where it can be used as a standard technique during software maintenance.

Understanding the behavior of a software system involves understanding both control flow and data flow. Studies carried out by psychologists give evidence that the fundamental information needed on the system behavior is control flow. Developers first build up a mental model of the control flow while trying to understand the behavior of a software system [168]. An understanding of what data is involved is built on top of this fundamental mental model. This thesis focuses on understanding behavior by means of understanding the control flow in the system on a function level granularity as gaining an understanding of this fundamental task alone is difficult to achieve, if large code code bases are concerned. Mentally reconstructing the control flow is challenging in these systems owing to the delocalized plans involved, i.e., conceptually related code that is implemented in non-contiguous parts of the typically large code base [200].

The main contributions of the thesis include:

- Chapter 4 proposes a function call tracing technique for C/C++ software systems that permits developers to instantly activate or deactivate tracing of a running software system as and when necessary. The technique is integrated into development processes without increasing build or runtime performance if tracing is disabled. Furthermore, a technique has been devised that identifies massively called functions during runtime and automatically disables tracing for them. By these means, a developer can not only choose the granularity level of the resulting trace data but also its size. Hence, the technique enables developers to trace system behavior over long time periods on the basis of a reasonably small amount of trace data.
- Chapter 5 proposes a novel technique for creating a hierarchy of higher-level trace abstractions by recursively splitting the trace into phases. The proposed algorithm for pruning traces enables developers to massively reduce the amount of calls captured in a trace in such a way that merely a skeleton of calls (i.e, phases) remains. With a *pruned trace* of this kind developers may navigate through the trace in coarse-grained steps and are able to quickly identify those parts of the trace that are relevant to their given maintenance task. The technique forms the basis for trace presentation techniques that tackle the scalability issue.
- Chapter 6: Here, a framework for trace visualization techniques is outlined. It provides solutions to the question as to how core techniques for viewing trace data can be implemented in such a way that developers are supported in performing top-down and bottom-up comprehension strategies. The chapter goes on to show how trace data taken from a fact base is filtered, transformed into a geometry model, and finally converted into a visual representation. Moreover, an explanation is given revealing how developers may use different

views on a trace simultaneously for being able to cross-reference findings made in a single view with context information provided by other views.

- Chapter 7: Trace visualization is only one technique from a large selection of other maintenance techniques. With a class of maintenance techniques in mind that relate to a specific system behavior and produce a set of artifacts potentially related to the given maintenance task, this chapter shows how trace visualization can be integrated as an intermediate step between obtaining the result set and the generally time-consuming code reading necessary to identify the true positives in the result set. Besides giving a general description on how to combine trace visualization with other maintenance techniques in general, this chapter introduces a novel fault localization technique that exploits trace visualization and is an example of how to combine trace visualization with "result set" based techniques.
  Seen from the trace visualization perspective, integrating trace visualization with other maintenance techniques solves one of the major problems encountered when exploring traces: finding the task-relevant parts of the trace. With a result set at hand obtained from another maintenance technique, developers are provided with precise entry points into the trace that shortcut an otherwise difficult top-down exploration.

- Chapter 8: The concept submitted in this thesis has been implemented as part of a framework for creating trace visualization tools. To ensure scalability of the concept—the prime subject of this thesis—performance measurements were taken when applying the framework to large industrial C/C++ software systems.

- Chapter 9: Here, details of case studies are given. These report on observations made when applying the concepts in real-world scenarios to solve particular maintenance problems. It has become clear that the concepts help developers in certain maintenance contexts. Case studies in general do not allow us to generalize results. However, these case studies demonstrate that the techniques proposed in this thesis can overcome scalability difficulties and are successful at solving real-world industrial maintenance problems.

10.2 Outlook - Further Research Directions

The thesis' concept lays the foundation for building scalable trace visualization tools suitable for industry. It is recommended that follow-up research be carried out in the following directions:

- Evaluation of the concepts by means of controlled experiments and field studies.

- Enriching trace data by collecting information on system state, e.g., variable values and object identities (in object-oriented languages).

- Visualization of the behavior of multi-threaded software systems.

- Visualization of the behavior of service-oriented software systems.

10.2.1 Evaluations - Controlled Experiments and Field Studies

This thesis, or more precisely the proposed trace visualization concept, gains its validity from the computational measurements of the algorithm implementations and from the quoted case studies. These evaluations show that the concepts meet the requirements of industrial-sized, real-world maintenance problems. The next step for an even more profound validation of the positive effects of trace visualization on developer performance is either to carry through *controlled experiments* [11] or closely monitor the long-term observations of industrial developers, e.g., by performing *field studies* or *surveys*. The latter evaluation methods require the availability of the trace visualization concepts as a mature tool adopted by professional software developers.

10.2.2 Enriching Traces with Information on System State

The concepts in this thesis consider only control flow information as runtime information. As stated in Chapter 2, creating a mental model of control flow is a fundamental task when trying to understand system behavior. Based on this model, developers build up mental models of the system state and how it changes during system execution, i.e., how the system operates on data. Hence, follow-up research should consider incorporating information on variable values and how these are accessed and modified [13].

Object Identifiers Object-oriented software systems (e.g., being written in C++) extend the procedural programming paradigm in such a way that functions and variables belong to objects representing their behavior and state. Seen from the perspective of object-orientation: System behavior results from objects sending messages to each other and reacting to the messages according to their current state. For each group of similar objects a class exists as "blueprint" that defines the objects' behavior (methods, i.e., functions) and possible states (attribute, i.e., variables).

A knowledge on how objects create and destroy each other and how they relate during their life time, reflects coarse-grained information on system state and is important when trying to understand the behavior of object-oriented systems. Hence, follow-up research should extend traces in such a way that method/function calls are associated with object identifiers and provide visualizations that take this additional information into account. First research contributions have been proposed recently [222].

Complementary Tracing Techniques The proposed solution for tracing function calls in C/C++ software systems is a technique that can be combined with a variety of complementary techniques that focus on gathering system state (cf Chapter 4). Hence, follow-up research includes applying such complementary techniques and extending trace presentation techniques with the additional information on state. For example, a useful visualization for developers who need to locate a fault would be a view of control flow with additional information on when specific variables are read or written.

10.2.3 Visualizing Multi-Threaded Software Systems

As a consequence of the tendency in hardware technology to provide CPUs with multiple cores, the focus on using multiple threads will be even greater in the forseeable future. Understanding system behavior for multiple threads is far from trivial. In single-threaded systems, developers need primarily to understand the sequential order of function calls. In the case of the vast majority of most maintenance tasks, the exact timing behavior can be neglected. In multi-threaded systems, understanding control flow is vital and must be done for each thread in parallel. Additionally, the timing plays an essential role when trying to understand system behavior. Hence, important follow-up research must focus on creating trace visualization techniques that enable developers to understand the behavior of multi-threaded software systems. Research contributions in this direction have recently been published [213, 214].

10.2.4 Visualizing Service-Oriented Software Systems

In a service-oriented software system, a suite of components providing *services* and communicating via a network are integrated loosely and can be used in multiple business domains. The services and their consumers interact by passing data in a well-defined format or by coordinating activities between multiple services. Understanding the runtime behavior of such software systems is challenging because, due to the loose coupling, it is difficult to "see" which concrete components are interacting. Furthermore, components can be exchanged at any time. Tracing and visualizing concrete interaction between the components is promising when seeking to understand such systems.

Bibliography

[1] Agrawal, H. ; Horgan, J. ; London, S. ; Wong, W.: Fault localization using execution slices and dataflow tests. In: *Proceedings of IEEE Software Reliability Engineering* (1995), pp. 143–151

[2] Agrawal, Hiralal ; Demillo, Richard A. ; Spafford, Eugene H.: Debugging with Dynamic Slicing and Backtracking. In: *Software - Practice and Experience* 23 (1993), No. 6, pp. 589–616

[3] Ball, Thomas ; Eick, Stephen G.: Software Visualization in the Large. In: *Computer* 29 (1996), No. 4, pp. 33–43

[4] Balzer, Michael ; Noack, Andreas ; Deussen, Oliver ; Lewerentz, Claus: Software Landscapes: Visualizing the Structure of Large Software Systems. In: *Proceedings of the Eurographics Symposium on Visualization*, 2004, pp. 261–266

[5] Basili, Victor R.: Evolving and packaging reading technologies. In: *Journal of Systems and Software* 38 (1997), No. 1, pp. 3–12

[6] Bennett, Chris ; Myers, Del ; Storey, Margaret-Anne ; German, Daniel: Working with 'Monster' Traces: Building a Scalable, Usable Sequence Viewer. In: *Proceedings of the 3rd International Workshop on Program Comprehension through Dynamic Analysis*, 2007, pp. 1–5

[7] Blender: *www.blender.org*. `www.blender.org`. Version: Blender Foundation

[8] Bohnet, Johannes ; Döllner, Jürgen: Analyzing feature implementation by visual exploration of architecturally-embedded call-graphs. In: *Proceedings of the ACM International Workshop on Dynamic Systems Analysis*. ACM, 2006, pp. 41–48

[9] Bohnet, Johannes ; Döllner, Jürgen: CGA Call Graph Analyzer - Locating and Understanding Functionality within the GNU Compiler Collection's Million Lines of Code. In: *Proceedings of the IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 2007, pp. 161–162

[10] Bohnet, Johannes ; Döllner, Jürgen: Facilitating Exploration of Unfamiliar Source Code by Providing 2.5D Visualizations of Dynamic Call Graphs. In:

Proceedings of the IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2007, pp. 63–66

[11] Bohnet, Johannes ; Döllner, Jürgen: Planning an Experiment on User Performance for Exploration of Diagrams Displayed in 2.5 Dimensions. In: *Proceedings der Software Engineering 2007 Fachtagung des GI-Fachbereichs Softwaretechnik (Workshops)*, GI, 2007, pp. 223–230

[12] Bohnet, Johannes ; Döllner, Jürgen: Visually exploring control flow graphs to support legacy software migration. In: *Proceedings der Software Engineering Konferenz der Gesellschaft für Informatik*, 2007, pp. 245–246

[13] Bohnet, Johannes ; Döllner, Jürgen: Analyzing dynamic call graphs enhanced with program state information for feature location and understanding. In: *Proceedings of the 30th IEEE/ACM International Conference on Software Engineering.* ACM, 2008, pp. 915–916

[14] Bohnet, Johannes ; Döllner, Jürgen: Visual exploration of function call graphs for feature location in complex software systems. In: *Proceedings of the ACM Symposium on Software Visualization.* ACM, 2006, pp. 95–104

[15] Bohnet, Johannes ; Koeleman, Martin ; Döllner, Jürgen: Visualizing Massively Pruned Execution Traces to Facilitate Trace Exploration. In: *Proceedings of the IEEE International Workshop on Visualizing Software for Understanding and Analysis*, IEEE, 2009, pp. 57–64

[16] Bohnet, Johannes ; Voigt, Stefan ; Döllner, Jürgen: Locating and Understanding Features of Complex Software Systems by Synchronizing Time-, Collaboration- and Code-focused Views on Execution Traces. In: *Proceedings of the 16th IEEE International Conference on Program Comprehension*, IEEE, 2008, 268–271

[17] Bohnet, Johannes ; Voigt, Stefan ; Döllner, Jürgen: Projecting code changes onto execution traces to support localization of recently introduced bugs. In: *Proceedings of the 24th ACM Symposium on Applied Computing.* ACM, 2009, pp. 438–442

[18] Borland Software Corporation: *Borland Together.* `http://www.borland.com/de/products/together`, retrieved 2. January 2010

[19] Brainsys Informatiksysteme GmbH: *Graphlet.* `http://www.infosun.fim.uni-passau.de/Graphlet`, retrieved 13. March 2010

[20] Brewer, Cynthia A. ; Hachard, Geoffrey W. ; Harrower, Mark A.: ColorBrewer in print: A catalog of color schemes for maps. In: *Cartographic and Geographic Information Science* 30 (2003), No. 1, pp. 5–32

[21] Brooks, Ruven E.: Towards a Theory of the Comprehension of Computer Programs. In: *International Journal of Man-Machine Studies* 18 (1983), No. 6, pp. 543–554

[22] Brown, Keith: Building a Lightweight COM Interception Framework Part 1: The Universal Delegator. In: *Microsoft Systems Journal* 14 (1999), January, No. 1. `http://www.microsoft.com/msj/0199/intercept/intercept.aspx`

[23] Brown, Marc H. ; Sedgewick, Robert: A system for algorithm animation. In: *SIGGRAPH Comput. Graph.* 18 (1984), No. 3, pp. 177–186

[24] Bruening, Derek L.: *Efficient, transparent, and comprehensive runtime code manipulation.* Cambridge, MA, USA, Diss., 2004

[25] Buchholz, Henrik ; Bohnet, Johannes ; Döllner, Jürgen: Smart and Physically-Based Navigation in 3D Geovirtual Environments. In: *9th International Conference on Information Visualization*, IEEE Computer Society Press, 2005, pp. 629–635

[26] Buchholz, Henrik ; Bohnet, Johannes ; Döllner, Jürgen: Smart Navigation Strategies for Virtual Landscapes. In: Buhmann, E. (Hrsg.) ; Paar, P. (Hrsg.) ; Bishop, I.D. (Hrsg.) ; Lange, E. (Hrsg.): *Trends in Real-time Visualization and Participation. Proceedings at Anhalt University of Applied Sciences*, Wichmann, 2005, pp. 124–131

[27] Butler, David M. ; Almond, James C. ; Bergeron, R. D. ; Brodlie, Ken W. ; Haber, Robert B.: Visualization reference models. In: *Proceedings of the 4th conference on Visualization*, 1993, pp. 337–342

[28] Cantrill, Bryan M. ; Shapiro, Michael W. ; Leventhal, Adam H.: Dynamic instrumentation of production systems. In: *Proceedings of the annual conference on USENIX Annual Technical Conference.* USENIX Association, 2004, pp. 15–28

[29] Card, Stuart K. ; Mackinlay, Jock D. ; Shneiderman, Ben: *Readings in information visualization: using vision to think.* Morgan Kaufmann Publishers Inc., 1999

[30] Carr, David A.: Guidelines for designing information visualization applications. In: *Ericsson Conference on Usability Engineering*, 1999, pp. 1–7

[31] Chen, Kunrong ; Rajlich, Vaclav: RIPPLES: Tool for Change in Legacy Software. In: *Proceedings of the IEEE International Conference on Software Maintenance* (2001), pp. 230–239

[32] Chi, Ed H.: A Taxonomy of Visualization Techniques Using the Data State Reference Model. In: *Proceedings of the IEEE Symposium on Information Vizualization.* IEEE Computer Society, 2000, pp. 69–76

[33] Chikofsky, Elliot J. ; Cross II, James H.: Reverse Engineering and Design Recovery: A Taxonomy. In: *IEEE Software* 7 (1990), No. 1, pp. 13–17

[34] Cleanscape Software International: *xSlice*. `http://legacy.cleanscape.net/products/testwise/tools_xslice.html`, retrieved 25. March 2010

[35] Clements, Paul ; Garlan, David ; Bass, Len ; Stafford, Judith ; Nord, Robert ; Ivers, James ; Little, Reed: *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002

[36] Cleve, Holger ; Zeller, Andreas: Locating Causes of Program Failures. In: *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 342–351

[37] Cmelik, Bob ; Keppel, David: Shade: a fast instruction-set simulator for execution profiling. In: *Proceedings of the ACM SIGMETRICS conference on Measurement and modeling of computer systems*. ACM, 1994, pp. 128–137

[38] Collberg, Christian ; Kobourov, Stephen ; Nagra, Jasvir ; Pitts, Jacob ; Wampler, Kevin: A system for graph-based visualization of the evolution of software. In: *Proceedings of the 2003 ACM symposium on Software visualization*. ACM, 2003, pp. 77–86

[39] Collberg, Christian ; Myles, Ginger ; Stepp, Michael: An empirical study of Java bytecode programs. In: *Software—Practice & Experience* 37 (2007), No. 6, pp. 581–641

[40] Corbi, T. A.: Program understanding: challenge for the 1990's. In: *IBM Systems Journal* 28 (1989), No. 2, pp. 294–306

[41] Cornelissen, Bas ; Holten, Danny ; Zaidman, Andy ; Moonen, Leon ; Wijk, Jarke J. ; Deursen, Arie van: Understanding Execution Traces Using Massive Sequence and Circular Bundle Views. In: *Proc. 15th IEEE International Conference on Program Comprehension ICPC '07*, 2007, pp. 49–58

[42] Cornelissen, Bas ; Zaidman, Andy ; Deursen, Arie van ; Moonen, Leon ; Koschke, Rainer: A Systematic Survey of Program Comprehension through Dynamic Analysis. In: *IEEE Transactions on Software Engineering* 35 (2009), No. 5, pp. 684–702

[43] Cornelissen, Bas ; Zaidman, Andy ; Holten, Danny ; Moonen, Leon ; Deursen, Arie van ; Wijk, Jarke J.: Execution trace analysis through massive sequence and circular bundle views. In: *Journal of Systems and Software* 81 (2008), No. 12, pp. 2252–2268

[44] De Pauw, Wim ; Jensen, Erik ; Mitchell, Nick ; Sevitsky, Gary ; Vlissides, John M. ; Yang, Jeaha: Visualizing the Execution of Java Programs. In: *Revised Lectures on Software Visualization, International Seminar*. Springer-Verlag, 2002, pp. 151–162

[45] De Pauw, Wim ; Lorenz, David ; Vlissides, John ; Wegman, Mark: Execution Patterns in Object-Oriented Visualization. In: *Proceedings of the Conference on Object-Oriented Technologies and Systems*, USENIX, 1998, 219–234

[46] Demeyer, Serge ; Ducasse, Stephane ; Lanza, Michele: A Hybrid Reverse Engineering Approach Combining Metrics and Program Visualization. In: *Working Conference on Reverse Engineering*, 1999, 175–186

[47] Demeyer, Serge ; Ducasse, Stéphane ; Nierstrasz, Oscar: *Object Oriented Reengineering Patterns.* Morgan Kaufmann Publishers Inc., 2002

[48] Deursen, Arie van ; Moonen, Leon: Exploring Legacy Systems Using Types. In: *Proceedings of the 7th Working Conference on Reverse Engineering.* IEEE Computer Society, 2000, pp. 32–41

[49] Diehl, Stefan: *Software Visualization. Visualizing the Structure, Behaviour, and Evolution of Software.* Springer, Berlin, 2007

[50] Ducasse, Stéphane ; Lanza, Michele ; Bertuli, Roland: High-Level Polymetric Views of Condensed Run-time Information. In: *Proceedings of the 8th Euromicro Working Conference on Software Maintenance and Reengineering.* IEEE Computer Society, 2004, pp. 309–318

[51] Eclipse Foundation: *Eclipse Test & Performance Tools Platform Project.* `http://www.eclipse.org/tptp`, retrieved 26. December 2009

[52] Eick, Stephen G. ; Steffen, Joseph L. ; Sumner, Eric E. Jr.: Seesoft—A Tool for Visualizing Line Oriented Software Statistics. In: *IEEE Transactions on Software Engineering* 18 (1992), No. 11, pp. 957–968

[53] Eiglsperger, Markus ; Gutwenger, Carsten ; Kaufmann, Michael ; Kupke, Joachim ; Jünger, Michael ; Leipert, Sebastian ; Klein, Karsten ; Mutzel, Petra ; Siebenhaller, Martin: Automatic layout of UML class diagrams in orthogonal style. In: *Information Visualization* 3 (2004), No. 3, pp. 189–208

[54] Eisenbarth, Thomas ; Koschke, Rainer ; Simon, Daniel: Locating Features in Source Code. In: *IEEE Transactions on Software Engineering* 29 (2003), No. 3, pp. 210–224

[55] Eisenberg, Andrew D. ; De Volder, Kris: Dynamic Feature Traces: Finding Features in Unfamiliar Code. In: *Proceedings of the 21st IEEE International Conference on Software Maintenance.* IEEE Computer Society, 2005, pp. 337–346

[56] Eldean AB: *ESS-Model.* `http://essmodel.sourceforge.net`, retrieved 2. January 2010

[57] Erlikh, Len: Leveraging Legacy System Dollars for E-Business. In: *IT Professional* 2 (2000), No. 3, pp. 17–23

[58] Ernst, Michael D.: Invited Talk Static and dynamic analysis: synergy and duality. In: *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering.* ACM, 2004, pp. 35–35

[59] Eysenck, Michael W. ; Keane, Mark T.: *Cognitive Psychology - A Student's Handbook.* Psychology Press, 2005

[60] Favre, Jean-Marie: GSEE: A Generic Software Exploration Environment. In: *Proceedings of the 9th International Workshop on Program Comprehension*, 2001, pp. 233–244

[61] Finnigan, P. J. ; Holt, R. C. ; Kalas, I. ; Kerr, S. ; Kontogiannis, K. ; Müller, H. A. ; Mylopoulos, J. ; Perelgut, S. G. ; Stanley, M. ; Wong, K.: The software bookshelf. In: *IBM Systems Journal* 36 (1997), No. 4, pp. 564–593

[62] Froehlich, Jon ; Dourish, Paul: Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams. In: *Proceedings of the 26th International Conference on Software Engineering.* IEEE Computer Society, 2004, pp. 387–396

[63] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns.* Addison-Wesley, 1995

[64] Gansner, Emden R. ; North, Stephen C.: An Open Graph Visualization System and Its Applications to Software Engineering. In: *Software - Practice and Experience* 30 (1999), pp. 1203–1233

[65] Gestwicki, Paul ; Jayaraman, Bharat: Methodology and architecture of JIVE. In: *Proceedings of the ACM symposium on Software visualization.* ACM, 2005, pp. 95–104

[66] Gilmore, D. J. ; Green, T. R. G.: Programming Plans and Programming Expertise. In: *The Quarterly Journal of Experimental Psychology* 40A (1988), No. 3

[67] Goldstine, Herman H. ; Neumann, John von: Planning and coding of problems for an electronic computing instrument. Part II, Vol. I / Institute for Advanced Study. 1948 (2). – Technical report

[68] Graham, Susan L. ; Kessler, Peter B. ; McKusick, Marshall K.: gprof: a call graph execution profiler. In: *SIGPLAN Notes* 39 (2004), No. 4, pp. 49–57

[69] Green Hills Software Inc.: *Time Machine Debugging Suite.* `http://www.ghs.com/products/timemachine.html`, retrieved 25. March 2010

[70] Greevy, Orla: *Enriching Reverse Engineering with Feature Analysis*, Universität Bern, Diss., 2007

[71] Greevy, Orla ; Lanza, Michele ; Wysseier, Christoph: Visualizing live software systems in 3D. In: *Proceedings of the ACM symposium on Software visualization.* ACM, 2006, pp. 47–56

[72] Griswold, William G. ; Yuan, Jimmy J. ; Kato, Yoshikiyo: Exploiting the map metaphor in a tool for software evolution. In: *Proceedings of the 23rd International Conference on Software Engineering.* IEEE Computer Society, 2001, pp. 265–274

[73] Gschwind, Thomas ; Oberleitner, Johann: Improving Dynamic Data Analysis with Aspect-Oriented Programming. In: *Proceedings of the 7th European Conference on Software Maintenance and Reengineering*, IEEE Computer Society, 2003, pp. 259–268

[74] Hamou-Lhadj, Abdelwahab ; Lethbridge, Timothy: Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System. In: *IEEE International Conference on Program Comprehension*, 2006, pp. 181–190

[75] Han, Jiawei ; Kamber, Micheline: *Data Mining: Concepts and Techniques.* 2 edition. Morgan Kaufmann, 2006

[76] Harris, David R. ; Reubenstein, Howard B. ; Yeh, Alexander S.: Reverse engineering to the architectural level. In: *Proceedings of the 17th international conference on Software engineering.* ACM, 1995, pp. 186–195

[77] Heer, Jeffrey ; Card, Stuart K. ; Landay, James A.: prefuse: a toolkit for interactive information visualization. In: *Proceedings of the SIGCHI conference on Human factors in computing systems.* ACM, 2005, pp. 421–430

[78] hello2morrow GmbH: *Sotograph.* `http://www.hello2morrow.com`, retrieved 7. February 2010

[79] Henkler, Stefan ; Greenyer, Joel ; Hirsch, Martin ; Schafer, Wilhelm ; Alhawash, Kahtan ; Eckardt, Tobias ; Heinzemann, Christian ; Loffler, Renate ; Seibel, Andreas ; Giese, Holger: Synthesis of timed behavior from scenarios in the Fujaba Real-Time Tool Suite. In: *Proceedings of the IEEE 31st International Conference on Software Engineering.* IEEE Computer Society, 2009, pp. 615–618

[80] Herman, Ivan ; Melançon, Guy ; Marshall, M. S.: Graph Visualization and Navigation in Information Visualization: A Survey. In: *IEEE Transactions on Visualization and Computer Graphics* 6 (2000), No. 1, pp. 24–43

[81] *Chapter* Understanding Architecture Through Structure and Behavior Visualization. In: Heuzeroth, Dirk ; Löwe, Welf: *Software Visualization - From Theory to Practice.* Kluwer Academic Publishers, 2003, pp. 243–286

[82] Hindle, A. ; Hindle, A. ; Jiang, Zhen M. ; Koleilat, W. ; Godfrey, M.W. ; Holt, R.C.: YARN: Animating Software Evolution. In: Jiang, Zhen M. (Hrsg.): *Proceedings ot the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 2007, pp. 129–136

[83] Hiramatsu, Masami: *Overhead Evaluation about KProbes and Djprobe (Direct Jump Probe).* `http://lkst.sourceforge.net/docs/probes-eval-report.pdf`, retrieved November 30, 2005

[84] Hofer, Christoph ; Denker, Marcus ; Ducasse, Stéphane: Design and Implementation of a Backward-In-Time Debugger. In: *Proceedings of NODE'06* Bd. P-88 Gesellschaft für Informatik (GI), 2006 (Lecture Notes in Informatics), 17–32

[85] Hollingsworth, Jeffrey K. ; Miller, Barton P. ; Cargille, Jon: Dynamic Program Instrumentation for Scalable Performance Tools. In: *Scalable High Performance Computing Conference*, 1994, pp. 841–850

[86] Horwitz, S. ; Reps, T. ; Binkley, D.: Interprocedural slicing using dependence graphs. In: *Proceedings of the ACM SIGPLAN conference on Programming Language design and Implementation.* ACM, 1988, pp. 35–46

[87] Hunt, Galen ; Brubacher, Doug: Detours: binary interception of Win32 functions. In: *Proceedings of the 3rd conference on USENIX Windows NT Symposium.* USENIX Association, 1998, pp. 135–143

[88] Imagix Corp.: *Imagix 4D.* `http://www.imagix.com`, retrieved 13. March 2010

[89] Intel Corporation: *VTune Performance Analyzer.* `http://software.intel.com/en-us/intel-vtune`, retrieved 25. March 2010

[90] International Business Machines Corporation: *IBM Rational.* `http://www.ibm.com/software/rational`, retrieved 2. January 2010

[91] International Business Machines Corporation: *AIX Version 6.1.* `http://www-03.ibm.com/systems/de/p/os/aix/v61`, retrieved 25. December 2009

[92] International Business Machines Corporation: *Zinsight.* `https://researcher.ibm.com/researcher/view_project.php?id=613`, retrieved 5. January 2010

[93] Jackson, Michael A.: *Principles of Program Design.* Academic Press, Inc., 1975

[94] Jerding, Dean ; Rugaber, Spencer: Using Visualization for Architectural Localization and Extraction. In: *Proceedings of the Working Conference on Reverse Engineering* (1997), pp. 56–65

[95] Jerding, Dean F. ; Stasko, John T.: The Information Mural: A Technique for Displaying and Navigating Large Information Spaces. In: *IEEE Transactions on Visualization and Computer Graphics* 4 (1998), pp. 257–271

[96] Jerding, Dean F. ; Stasko, John T. ; Ball, Thomas: Visualizing Interactions in Program Executions. In: *Proceedings of the International Conference on Software Engineering*, 1997, pp. 360–370

[97] Johnson-Laird, P. N. ; Legrenzi, P. ; Girotto, V. ; Legrenzi, M. S. ; Caverni, J. P.: Naive probability: a mental model theory of extensional reasoning. In: *Psychological review* 106 (1999), January, No. 1, pp. 62–88

[98] Jones, James A. ; Harrold, Mary J.: Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In: *Proceedings of the International Conference on Automated Software Engineering*, 2005, pp. 273–282

[99] Jones, James A. ; Harrold, Mary J. ; Stasko, John: Visualization of Test Information to Assist Fault Localization. In: *Proceedings of the Interational Conference on Software Engineering*, 2002, pp. 467–477

[100] Kazman, R. ; Carrière, S.J.: View Extraction and View Fusion in Architectural Understanding. In: *Proceedings of the 5th International Conference on Software Reuse* (1998), pp. 290–299

[101] Kazman, Rick ; Woods, Steven G. ; Carrière, S. J.: Requirements for Integrating Software Architecture and Reengineering Models: CORUM II. In: *Proceedings of the Working Conference on Reverse Engineering.* IEEE Computer Society, 1998, pp. 154–163

[102] Kerren, Andreas ; Stasko, John T.: Algorithm Animation - Introduction. In: *Software Visualization*, 2002, pp. 1–15

[103] Kienle, Holger: *Building Reverse Engineering Tools with Software Components*, University of Victoria, Diss., 2006

[104] In: Kimelman, Doug ; Rosenburg, Bryan ; Roth, Tova: *Visualization of Dynamics in Real World Software Systems.* MIT Press, 1998, pp. 293–314

[105] Klint, Paul: How Understanding and Restructuring Differ from Compiling - A Rewriting Perspective. In: *Proceedings of the 11th IEEE International Workshop on Program Comprehension.* IEEE Computer Society, 2003, pp. 2–12

[106] Ko, Andrew J. ; DeLine, Robert ; Venolia, Gina: Information Needs in Collocated Software Development Teams. In: *Proceedings of the 29th International Conference on Software Engineering*, IEEE Computer Society, 2007, pp. 344–353

[107] Ko, Andrew J. ; Myers, Brad A.: Debugging reinvented: asking and answering why and why not questions about program behavior. In: *Proceedings of the 30th international conference on Software engineering.* ACM, 2008, pp. 301–310

[108] Koeleman, Martin: *Detection, Processing and Visualization of Execution Information in Complex Software Systems*, Hasso-Plattner-Institute at the University of Potsdam, Germany, Master Thesis, March 2009

[109] Kosara, Robert ; Healey, Christopher G. ; Interrante, Victoria ; Laidlaw, David H. ; Ware, Colin: User Studies: Why, How, and When? In: *IEEE Computer Graphics and Applications* 23 (2003), No. 4, pp. 20–25

[110] Koschke, Rainer: Software Visualization for Reverse Engineering. In: *Software Visualization*, 2001, pp. 138–150

[111] Koskimies, Kai ; Mössenböck, Hanspeter: Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs. In: *In Proceedings of the 18th international conference on software engineering*, 1996, pp. 366–375

[112] Kouznetsova, Svetlana: Using BlueJ and Blackjack to teach object-oriented design concepts in CS1. In: *Journal of Computing Sciences in Colleges* 22 (2007), No. 4, pp. 49–55

[113] *Chapter* Visualizing Program Behavior with the Event Graph. In: Kranzlmüller, Dieter: *Software Visualization - From Theory to Practice.* Kluwer Academic Publishers, 2003, pp. 29–57

[114] Kruchten, Philippe: *The Rational Unified Process: An Introduction.* Addison-Wesley Longman Publishing Co., Inc., 2003

[115] Lange, Carola ; Winter, Andreas ; Sneed, Harry M.: Comparing Graph-Based Program Comprehension Tools to Relational Database-Based Tools. In: *Proceedings of the 9th International Workshop on Program Comprehension.* IEEE Computer Society, 2001, pp. 209–220

[116] Lange, Danny B. ; Nakamura, Y.: Object-oriented program tracing and visualization. In: *Computer* 30 (1997), No. 5, pp. 63–70

[117] Lanza, Michele ; Ducasse, Stéphane: A categorization of classes based on the visualization of their internal structure: the class blueprint. In: *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* ACM, 2001, pp. 300–311

[118] Lanza, Michele ; Marinescu, Radu: *Object-Oriented Metrics in Practice.* 2006

[119] Letovsky, Stanley: Cognitive processes in program comprehension. In: *Journal Systems Software* 7 (1987), No. 4, pp. 325–339

[120] Lewis, Bil: Debugging Backwards in Time. In: *Proceedings of the 5th International Workshop on Automated Debugging*, 2003

[121] Lienhard, Adrian ; Gîrba, Tudor ; Nierstrasz, Oscar: Practical Object-Oriented Back-in-Time Debugging. In: *Proceedings of the 22nd European conference on Object-Oriented Programming.* Springer-Verlag, 2008, pp. 592–615

[122] Lorenz, Mark ; Kidd, Jeff: *Object-Oriented Software Metrics.* Prentice Hall, 1994

[123] Löwe, Welf ; Panas, Thomas: Rapid Construction of Software Comprehension Tools. In: *International Journal of Software Engineering and Knowledge Engineering* 15 (2005), No. 6, pp. 995–1026

[124] Luk, Chi-Keung ; Cohn, Robert ; Muth, Robert ; Patil, Harish ; Klauser, Artur ; Lowney, Geoff ; Wallace, Steven ; Reddi, Vijay J. ; Hazelwood, Kim: Pin: building customized program analysis tools with dynamic instrumentation. In: *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation.* ACM, 2005, pp. 190–200

[125] Maass, Stefan: *Techniken zur automatisierten Annotation interaktiver geovirtueller 3D-Umgebungen*, Hasso-Plattner-Institut at the University of Potsdam, Diss., 2009

[126] Maass, Stefan ; Döllner, Jürgen: Efficient View Management for Dynamic Annotation Placement in Virtual Landscapes. In: *Smart Graphics*, 2006, pp. 1–12

[127] Madison, A. W. ; Batson, Alan P.: Characteristics of program localities. In: *Communications of the ACM* 19 (1976), No. 5, pp. 285–294

[128] Maimon, Oded ; Rokach, Lior: *Data Mining and Knowledge Discovery Handbook.* Springer-Verlag New York, Inc., 2005

[129] Maletic, Jonathan I. ; Marcus, Andrian ; Collard, Michael L.: A Task Oriented View of Software Visualization. In: *Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis.* IEEE Computer Society, 2002, pp. 32–40

[130] Mamone, Salvatore: The IEEE standard for software maintenance. In: *SIGSOFT Software Engineering Notes* 19 (1994), No. 1, pp. 75–76

[131] Maqbool, Onaiza ; Babri, Haroon: Hierarchical Clustering for Software Architecture Recovery. In: *IEEE Transactions on Software Engineering* 33 (2007), No. 11, pp. 759–780

[132] Marcus, Andrian ; Feng, Louis ; Maletic, Jonathan I.: 3D representations for software visualization. In: *Proceedings of the ACM symposium on Software visualization.* ACM, 2003, pp. 27–36

[133] Marcus, Andrian ; Feng, Louis ; Maletic, Jonathan I.: Comprehension of Software Analysis Data Using 3D Visualization. In: *Proceedings of the 11th IEEE International Workshop on Program Comprehension.* IEEE Computer Society, 2003, pp. 105–114

[134] Mayrhauser, Anneliese von ; Vans, A. M.: Program Comprehension During Software Maintenance and Evolution. In: *IEEE Computer* Bd. August 1995 (Vol. 28, No. 8), 1995, pp. 44–55

[135] Mayrhauser, Anneliese von ; Vans, A. M.: Program Understanding: Models and Experiments. In: *Advances in Computers* 40 (1995), pp. 1–38

[136] Mehner, Katharina: JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs. In: *Software Visualization*, 2001, pp. 163–175

[137] Mehta, Alok ; Heineman, George T.: Evolving legacy systems features using regression test cases and components. In: *Proceedings of the 4th International Workshop on Principles of Software Evolution.* ACM, 2001, pp. 190–193

[138] Michaud, Jeff ; Storey, Margaret-Anne ; Muller, Hausi: Integrating Information Sources for Visualizing Java Programs. In: *Proceedings of the International Conference on Software Maintenance.* IEEE Computer Society, 2001, pp. 250–259

[139] Microsoft Corporation: *Enable _penter Hook Function with the /Gh compiler option.* `http://msdn.microsoft.com/en-us/library/c63a9b7h%28VS.100%29.aspx`, retrieved 13. December 2009

[140] Microsoft Corporation: *Debugging Tools for Windows.* `http://www.microsoft.com/whdc/DevTools/Debugging/default.mspx`, retrieved November 30

[141] Mili, Rym ; Steiner, Renee: Software Engineering - Introduction. In: *Software Visualization*, 2001, pp. 129–137

[142] Müller, H. A. ; Klashinsky, K.: Rigi—A system for programming-in-the-large. In: *Proceedings of the 10th international conference on Software engineering.* IEEE Computer Society Press, 1988, pp. 80–86

[143] Müller, Hausi A. ; Tilley, Scott R. ; Wong, Kenny: Understanding software systems using reverse engineering technology perspectives from the Rigi project. In: *Proceedings of the conference of the Centre for Advanced Studies on Collaborative research*, IBM Press, 1993, pp. 217–226

[144] Moonen, Leon: *Exploring Software Systems*, Faculty of Natural Sciences, Mathematics, and Computer Science, University of Amsterdam, Diss., December 2002

[145] Moret, P. ; Binder, W. ; Ansaloni, D. ; Villazon, A.: Visualizing Calling Context Profiles with Ring Charts. In: *Proceedings of the 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 2009, pp. 33–36

[146] Murphy, Gail C. ; Notkin, David ; Sullivan, Kevin J.: Software Reflexion Models: Bridging the Gap between Design and Implementation. In: *IEEE Transactions on Software Engineering* 27 (2001), No. 4, pp. 364–380

[147] Mutzel, Petra ; Eades, Peter: Graphs in Software Visualization - Introduction. In: *Software Visualization*, 2001, pp. 285–294

[148] Myers, B. A.: Visual programming, programming by example, and program visualization: a taxonomy. In: *Proceedings of the SIGCHI conference on Human factors in computing systems* 17 (1986), No. 4, pp. 59–66

[149] Myers, B.A.: Taxonomies of visual programming and program visualization. In: *Journal of Visual languages and Computing* 1 (1990), pp. 97–123

[150] Nagpurkar, Priya ; Krintz, Chandra ; Hind, Michael ; Sweeney, Peter F. ; Rajan, V. T.: Online Phase Detection Algorithms. In: *Proceedings of the International Symposium on Code Generation and Optimization.* IEEE Computer Society, 2006, pp. 111–123

[151] Nassi, I. ; Shneiderman, B.: Flowchart techniques for structured programming. In: *SIGPLAN Notes* 8 (1973), No. 8, pp. 12–26

[152] Nethercote, Nicholas: *Dynamic Binary Analysis and Instrumentation*, University of Cambridge, United Kingdom, Diss., November 2004

[153] Nevill-Manning, Craig G. ; Witten, Ian H.: Identifying hierarchical structure in sequences: a linear-time algorithm. In: *Journal of Artificial Intelligence Research* 7 (1997), pp. 67–82

[154] Noack, Andreas: Energy Models for Graph Clustering. In: *Journal Graph Algorithms Applications* 11 (2007), No. 2, pp. 453–480

[155] Object Management Group: *The Unified Modeling Language UML.* `http://www.uml.org`, retrieved 26. December 2009

[156] Open Source: *Source Navigator.* `http://sourcenav.sourceforge.net`, retrieved 13. March 2010

[157] Open Source: *Doxygen - Source code documentation generator tool.* `http://www.stack.nl/~dimitri/doxygen`, retrieved 15. April 2010

[158] Open Source: *Chronomancer.* `http://code.google.com/p/chronomancer`, retrieved 25. March 2010

[159] oreas GmbH: *Open Graph Drawing Framework.* http://www.ogdf.net, retrieved 13. March 2010

[160] oreas GmbH: *GoVisual.* http://www.oreas.com, retrieved 2. January 2010

[161] Orso, Alessandro ; Jones, James A. ; Harrold, Mary J.: GAMMATELLA: visualizing program-execution data for deployed software. In: *Information Visualization* 3 (2004), No. 3, pp. 173–188

[162] Ottenstein, Karl J. ; Ottenstein, Linda M.: The program dependence graph in a software development environment. In: *Proceedings of the first SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments.* ACM, 1984, pp. 177–184

[163] Panas, Thomas ; Lundberg, Jonas ; Löwe, Welf: Reuse in Reverse Engineering. In: *International Conference on Program Comprehension* (2004), pp. 52–61

[164] Panchenko, Oleksandr ; Koglin, Alexander ; Bohnet, Johannes ; Zeier, Alexander: XPath-Based Query Language for Trace Analysis. In: *Proceedings of the 5th International Workshop on Program Comprehension through Dynamic Analysis*, 2010, pp. to appear

[165] Parnas, D. L.: On the criteria to be used in decomposing systems into modules. In: *Communications of the ACM* 15 (1972), No. 12, pp. 1053–1058

[166] Parnas, David L.: Software aging. In: *International Conference on Software Engineering*, 1994, pp. 279–287

[167] Passing, Johannes ; Schmidt, Alexander ; Löwis, Martin von ; Polze, Andreas: NTrace: Function Boundary Tracing for Windows on IA-32. In: *Working Conference on Reverse Engineering*, IEEE Computer Society, 2009, pp. 43–52

[168] Pennington, Nancy: Comprehension strategies in programming. (1987), pp. 100–113

[169] Pennington, Nancy: Stimulus structures and mental representations in expert comprehension of computer programs. In: *Cognitive Psychology* 19 (1987), No. 3, pp. 295–341

[170] Pilato, C. M. ; Collins-Sussman, Ben ; Fitzpatrick, Brian W.: *Version Control with Subversion.* 2nd. O'Reilly Media, 2008

[171] Pinzger, M. ; Fischer, M. ; Gall, H. ; Jazayeri, M.: Revealer: A Lexical Pattern Matcher for Architecture Recovery. In: *Proceedings of the 9th Working Conference on Reverse Engineering.* IEEE Computer Society, 2002, pp. 170–180

[172] Pothier, Guillaume ; Tanter, Eric: Back to the Future: Omniscient Debugging. In: *IEEE Software* 26 (2009), pp. 78–85

[173] Price, Blaine A. ; Baecker, Ronald ; Small, Ian S.: A Principled Taxonomy of Software Visualization. In: *Journal of Visual languages and Computing* 4 (1993), No. 3, pp. 211–266

[174] Raza, Aoun ; Vogel, Gunther ; Plödereder, Erhard: Bauhaus - A Tool Suite for Program Analysis and Reverse Engineering. In: *Ada-Europe*, 2006, pp. 71–82

[175] Reiss, Steven P.: Visualizing Java in action. In: *Proceedings of the ACM symposium on Software visualization.* ACM, 2003, pp. 57–66

[176] Reiss, Steven P.: Dynamic detection and visualization of software phases. In: *SIGSOFT Software Engineering Notes* 30 (2005), No. 4, pp. 1–6

[177] Ren, Xiaoxia ; Chesley, Ophelia C. ; ; Ryder, Barbara G.: Identifying Failure Causes in Java Programs: An Application of Change Impact Analysis. In: *IEEE Transactions on Software Engineering* 32 (2006), pp. 718–732

[178] Renieris, M. ; Reiss, S.: Fault localization with nearest neighbor queries. In: *In Proceedings of the 18th Conference on Automated Software Engineering*, 2003, 30–39

[179] Renieris, Manos ; Reiss, Steven P.: Almost: Exploring Program Traces. In: *Workshop on New Paradigms in Information Visualization and Manipulation*, 1999, pp. 70–77

[180] Richner, Tamar ; Ducasse, Stephane: Using Dynamic Information for the Iterative Recovery of Collaborations and Roles. In: *Proceedings of the International Conference on Software Maintenance*, 2002, pp. 34–43

[181] Roman, Gruia-Catalin ; Cox, Kenneth C.: A Taxonomy of Program Visualization Systems. In: *Computer* 26 (1993), No. 12, pp. 11–24

[182] Röthlisberger, David ; Härry, Marcel ; Villazon, Alex ; Ansaloni, Danilo ; Binder, Walter ; Nierstrasz, Oscar ; More, Philippe: Augmenting Static Source Views in IDEs with Dynamic Metrics. In: *Proceedings of the 25th International Conference on Software Maintenance*, 2009, pp. 253–262

[183] Rumbaugh, J. ; Jacobson, I. ; Booch, G.: *The Unified Modeling Language Reference Manual.* Addsion Wesley, 1999

[184] Salah, Maher M.: *An environment for comprehending the behavior of software systems.* Philadelphia, PA, USA, Diss., 2005

[185] Saleh, Kassem A.: *Software Engineering.* J. Ross Publishing Inc., 2009

[186] Schumann, Heidrun ; Müller, Wolfgang: *Visualisierung - Grundlagen und allgemeine Methoden.* Springer Verlag Berlin Heidelberg, 2000

[187] Scientific Toolworks Inc.: *Understand.* `http://www.scitools.com`, retrieved 13. March 2010

[188] Sherwood, Timothy ; Perelman, Erez ; Hamerly, Greg ; Calder, Brad: Automatically characterizing large scale program behavior. In: *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems.* ACM, 2002, pp. 45–57

[189] Shneiderman, Ben: Tree visualization with tree-maps: 2-d space-filling approach. In: *ACM Transactions on Graphics* 11 (1992), No. 1, pp. 92–99

[190] Shneiderman, Ben ; Mayer, Richard: Syntactic/semantic interactions in programmer behavior: A model and experimental results. In: *International Journal of Parallel Programming* 8 (1979), June, No. 3, pp. 219–238

[191] Shneiderman, Ben ; Wattenberg, Martin: Ordered Treemap Layouts. In: *Proceedings of the IEEE Symposium on Information Visualization.* IEEE Computer Society, 2001, pp. 73–78

[192] Sim, S. E. ; Clarke, C. L. A. ; Holt, R. C.: Archetypal Source Code Searches: A Survey of Software Developers and Maintainers. In: *Proceedings of the 6th International Workshop on Program Comprehension.* IEEE Computer Society, 1998, pp. 180–189

[193] Smith, Michael ; Munro, Malcolm: Providing a User Customisable Tool for Software Visualisation at Runtime. In: *Proceedings of the International Conference on Visualization, Imaging and Image Processing*, 2004

[194] Smith, M.P. ; Munro, M.: Runtime visualisation of object oriented software. In: *Proc. First International Workshop on Visualizing Software for Understanding and Analysis*, 2002, pp. 81–89

[195] Soloway, Elliot ; Ehrlich, Kate: Empirical Studies of Programming Knowledge. In: *IEEE Transactions on Software Engineering* 10 (1984), No. 5, pp. 595–609

[196] Spence, Robert: *Information visualization.* Addison-Wesley, 2001 (ACM Press books)

[197] Stasko, John ; Zhang, Eugene: In: *Proceedings of the IEEE Symposium on Information Vizualization.* IEEE Computer Society, 2000, pp. 57–65

[198] Stasko, John T.: Tango: A Framework and System for Algorithm Animation. In: *Computer* 23 (1990), pp. 27–39

[199] Stasko, John T. ; Wehrli, Joseph F.: Three-Dimensional Computation Visualization. In: *Proceedings of the IEEE Symposium on Visual Languages*, 1993, pp. 100–107

[200] Storey, M.-A. D. ; Fracchia, F. D. ; Mueller, H. A.: Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualization. In: *Proceedings of the 5th International Workshop on Program Comprehension.* IEEE Computer Society, 1997, pp. 17–28

[201] Storey, M.-A. D. ; Muller, H. A.: Manipulating and documenting software structures using SHriMP views. In: *Proceedings of the International Conference on Software Maintenance.* IEEE Computer Society, 1995, pp. 275–284

[202] *Chapter* Designing a Software Exploration Tool Using a Cognitive Framework. In: Storey, Margaret-Anne: *Software Visualization - From Theory to Practice.* Kluwer Academic Publishers, 2003, pp. 113–147

[203] Storey, Margaret-Anne ; Best, Casey ; Michaud, Jeff ; Rayside, Derek ; Litoiu, Marin ; Musen, Mark: SHriMP views: an interactive environment for information visualization and navigation. In: *Extended abstracts on Human factors in computing systems.* ACM, 2002, pp. 520–521

[204] Sugiyama, K. ; Misue, K.: Visualization of structural information: automatic drawing of compound digraphs. In: *Systems, Man and Cybernetics, IEEE Transactions on* 21 (1991), Jul/Aug, No. 4, pp. 876–892

[205] Systä, Tarja: Dynamic Reverse Engineering of Java Software. In: *Proceedings of the Workshop on Object-Oriented Technology.* Springer-Verlag, 1999, pp. 174–175

[206] Systä, Tarja: *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*, University of Tampere, Diss., 2000

[207] Systä, Tarja: Understanding the Behavior of Java Programs. In: *Proceedings of the Working Conference on Reverse Engineering*, 2000, 214–223

[208] Systä, Tarja ; Koskimies, Kai ; Müller, Hausi: Shimba—an environment for reverse engineering Java software systems. In: *Software—Practice & Experience* 31 (2001), No. 4, pp. 371–394

[209] Telea, Alexandru ; Maccari, Alessandro ; Riva, Claudio: An open toolkit for prototyping reverse engineering visualizations. In: *Proceedings of the symposium on Data Visualisation.* Eurographics Association, 2002, pp. 241–250

[210] Teyseyre, Alfredo R. ; Campo, Marcelo R.: An Overview of 3D Software Visualization. In: *IEEE Transactions on Visualization and Computer Graphics* 15 (2009), pp. 87–105

[211] Thiemann, Peter: Higher-Order Code Splicing. In: *Proceedings of the 8th European Symposium on Programming Languages and Systems.* Springer-Verlag, 1999, pp. 243–257

[212] Tilley, Scott R.: The canonical activities of reverse engineering. In: *Annals of Software Engineering* 9 (2000), No. 1-4, pp. 249–271

[213] Trümper, J. ; Bohnet, J. ; Döllner, J.: Understanding Complex Multithreaded Software Systems by Using Trace Visualization. In: *Proceedings of the ACM Symposium on Software Visualization*, 2010, pp. to appear

[214] Trümper, J. ; Bohnet, J. ; Voigt, S. ; Döllner, J.: Visualization of Multithreaded Behavior to Facilitate Maintenance of Complex Software Systems. In: *Proceedings of the International Conference on the Quality of Information and Communications Technology*, 2010, pp. to appear

[215] Trümper, Jonas: *Localization and Visualization of Faulty Code Changes*, Hasso-Plattner-Institut für Softwaresystemtechnik an der Universität Potsdam, Master Thesis, 2009

[216] Tufte, Edward R.: *The Visual Display of Quantitative Information.* Graphics Press, 2001

[217] Undo Ltd.: *UndoDB.* `http://undo-software.com`, retrieved 25. March 2010

[218] Van Wijk, Jarke J. ; Wetering, Huub van d.: Cushion Treemaps: Visualization of Hierarchical Information. In: *Proceedings of the IEEE Symposium on Information Visualization.* IEEE Computer Society, 1999, pp. 73–78

[219] Vidacs, Laszlo ; Beszedes, Arpad ; Ferenc, Rudolf: Macro Impact Analysis Using Macro Slicing. In: *Proceedings of the 2nd International Conference on Software and Data Technologies*, INSTICC Press, 2007, pp. 230–235

[220] Voigt, Stefan: *Visualisierung von Dynamik komplexer Softwaresysteme*, Hasso-Plattner-Institute at the University of Potsdam, Germany, Master Thesis, April 2008

[221] Voigt, Stefan ; Bohnet, Johannes ; Döllner, Jürgen: Enhancing structural views of software systems by dynamic information. In: *Proceedings of 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 2009, pp. 47–50

[222] Voigt, Stefan ; Bohnet, Johannes ; Döllner, Jürgen: Object aware execution trace exploration. In: *Proceedings of the 25th IEEE International Conference on Software Maintenance* (2009), pp. 201–210

[223] Voinea, Lucian ; Lukkien, Johan ; Telea, Alexandru: Visual assessment of software evolution. In: *Science of Computer Programming* 65 (2007), No. 3, pp. 222–248

[224] Walker, Robert J. ; Murphy, Gail C. ; Freeman-Benson, Bjorn ; Wright, Darin ; Swanson, Darin ; Isaak, Jeremy: Visualizing dynamic software system information through high-level models. In: *Proceedings of the 13th SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* ACM, 1998, pp. 271–283

[225] WARE, Colin: *Information Visualization: Perception for Design*. 2nd. Morgan Kaufmann Publishers, 2004

[226] WARE, Colin: Visual Queries: The Foundation of Visual Thinking. In: *Knowledge and Information Visualization*, 2005, pp. 27–35

[227] WETTEL, Richard ; LANZA, Michele: Program Comprehension through Software Habitability. In: *Proceedings 15th IEEE International Conference on Program Comprehension*, 2007, pp. 231–240

[228] WILDE, Norman ; HUITT, Ross: Maintenance Support for Object-Oriented Programs. In: *IEEE Transaction on Software Engineering* 18 (1992), No. 12, pp. 1038–1044

[229] WILDE, Norman ; SCULLY, Michael C.: Software reconnaissance: mapping program features to code. In: *Journal of Software Maintenance* 7 (1995), No. 1, pp. 49–62

[230] WITTEN, Ian H. ; FRANK, Eibe: *Data mining: practical machine learning tools and techniques*. 2. ed. Elsevier, Morgan Kaufman, 2005

[231] XIE, Tao ; THUMMALAPENTA, Suresh ; LO, David ; LIU, Chao: Data Mining for Software Engineering. In: *Computer* 42 (2009), pp. 55–62

[232] ZAIDMAN, Andy: *Scalability Solutions for Program Comprehension Through Dynamic Analysis*, Unviversiteit Antwerpan, Diss., 2006

[233] ZELLER, Andreas ; LÜTKEHAUS, Dorothea: DDD—a free graphical front-end for UNIX debuggers. In: *SIGPLAN Notes* 31 (1996), No. 1, pp. 22–27

[234] ZHANG, Kang (Hrsg.): *Software Visualization: From Theory to Practice*. Kluwer Academic Publishers, 2003

[235] ZIMMERMANN, Thomas ; ZELLER, Andreas: Visualizing Memory Graphs. In: *Revised Lectures on Software Visualization, International Seminar*. Springer-Verlag, 2002, pp. 191–204

List of Figures

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate deutlich kenntlich gemacht zu haben.

Potsdam, den 13.10.2010 Johannes Bohnet