HASSO-PLATTNER-INSTITUT

Fachgebiet Computergrafische Systeme

Visualization Techniques for the Analysis of Software Behavior and Related Structures

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

Jonas Trümper

Potsdam, Germany
29. Januar 2014

Abstract

Software maintenance encompasses any changes made to a software system after its initial deployment and is thereby one of the key phases in the typical software-engineering lifecycle. In software maintenance, we primarily need to understand structural and behavioral aspects, which are difficult to obtain, e.g., by code reading. Software analysis is therefore a vital tool for maintaining these systems: It provides – the preferably automated – means to extract and evaluate information from their artifacts such as software structure, runtime behavior, and related processes.

However, such analysis typically results in massive raw data, so that even experienced engineers face difficulties directly examining, assessing, and understanding these data. Among other things, they require tools with which to explore the data if no clear question can be formulated beforehand. For this, software analysis and visualization provide its users with powerful interactive means. These enable the automation of tasks and, particularly, the acquisition of valuable and actionable insights into the raw data. For instance, one means for exploring runtime behavior is trace visualization.

This thesis aims at extending and improving the tool set for visual software analysis by concentrating on several open challenges in the fields of dynamic and static analysis of software systems. This work develops a series of concepts and tools for the exploratory visualization of the respective data to support users in finding and retrieving information on the system artifacts concerned. This is a difficult task, due to the lack of appropriate visualization metaphors; in particular, the visualization of complex runtime behavior poses various questions and challenges of both a technical and conceptual nature.

This work focuses on a set of visualization techniques for visually representing control-flow related aspects of software traces from shared-memory software systems: A trace-visualization concept based on icicle plots aids in understanding both single-threaded as well as multi-threaded runtime behavior on the function level. The concept's extensibility further allows the visualization and analysis of specific aspects of multi-threading such as synchronization, the correlation of such traces with data from static software analysis, and a comparison between traces. Moreover, complementary techniques for simultaneously analyzing system structures and the evolution of related attributes are proposed. These aim at facilitating long-term planning of software architecture and supporting management decisions in software projects by extensions to the circular-bundle-view technique: An extension to 3-dimensional space allows for the use of additional variables simultaneously; interaction techniques allow for the modification of structures in a visual manner.

The concepts and techniques presented here are generic and, as such, can be applied beyond software analysis for the visualization of similarly structured data. The techniques' practicability is demonstrated by several qualitative studies using subject data from industry-scale software systems. The studies provide initial evidence that the techniques' application yields useful insights into the subject data and its interrelationships in several scenarios.

Kurzfassung

Die Softwarewartung umfasst alle Änderungen an einem Softwaresystem nach dessen initialer Bereitstellung und stellt damit eine der wesentlichen Phasen im typischen Softwarelebenszyklus dar. In der Softwarewartung müssen wir insbesondere strukturelle und verhaltensbezogene Aspekte verstehen, welche z.B. alleine durch Lesen von Quelltext schwer herzuleiten sind. Die Softwareanalyse ist daher ein unverzichtbares Werkzeug zur Wartung solcher Systeme: Sie bietet – vorzugsweise automatisierte – Mittel, um Informationen über deren Artefakte, wie Softwarestruktur, Laufzeitverhalten und verwandte Prozesse, zu extrahieren und zu evaluieren.

Eine solche Analyse resultiert jedoch typischerweise in großen und größten Rohdaten, die selbst erfahrene Softwareingenieure direkt nur schwer untersuchen, bewerten und verstehen können. Unter Anderem dann, wenn vorab keine klare Frage formulierbar ist, benötigen sie Werkzeuge, um diese Daten zu erforschen. Hierfür bietet die Softwareanalyse und Visualisierung ihren Nutzern leistungsstarke, interaktive Mittel. Diese ermöglichen es Aufgaben zu automatisieren und insbesondere wertvolle und belastbare Einsichten aus den Rohdaten zu erlangen. Beispielsweise ist die Visualisierung von Software-Traces ein Mittel, um das Laufzeitverhalten eines Systems zu ergründen.

Diese Arbeit zielt darauf ab, den „Werkzeugkasten“ der visuellen Softwareanalyse zu erweitern und zu verbessern, indem sie sich auf bestimmte, offene Herausforderungen in den Bereichen der dynamischen und statischen Analyse von Softwaresystemen konzentriert. Die Arbeit entwickelt eine Reihe von Konzepten und Werkzeugen für die explorative Visualisierung der entsprechenden Daten, um Nutzer darin zu unterstützen, Informationen über betroffene Systemartefakte zu lokalisieren und zu verstehen. Da es insbesondere an geeigneten Visualisierungsmetaphern mangelt, ist dies eine schwierige Aufgabe. Es bestehen, insbesondere bei komplexen Softwaresystemen, verschiedenste offene technische sowie konzeptionelle Fragestellungen und Herausforderungen.

Diese Arbeit konzentriert sich auf Techniken zur visuellen Darstellung kontrollflussbezogener Aspekte aus Software-Traces von Shared-Memory Softwaresystemen: Ein Trace-Visualisierungskonzept, basierend auf Icicle Plots, unterstützt das Verstehen von single- und multi-threaded Laufzeitverhalten auf Funktionsebene. Die Erweiterbarkeit des Konzepts ermöglicht es zudem spezifische Aspekte des Multi-Threading, wie Synchronisation, zu visualisieren und zu analysieren, derartige Traces mit Daten aus der statischen Softwareanalyse zu korrelieren sowie Traces mit einander zu vergleichen. Darüber hinaus werden komplementäre Techniken für die kombinierte Analyse von Systemstrukturen und der Evolution zugehöriger Eigenschaften vorgestellt. Diese zielen darauf ab, die Langzeitplanung von Softwarearchitekturen und Management-Entscheidungen in Softwareprojekten mittels Erweiterungen an der Circular-Bundle-View-Technik zu unterstützen: Eine Erweiterung auf den 3-dimensionalen Raum ermöglicht es zusätzliche visuelle Variablen zu nutzen; Strukturen können mithilfe von Interaktionstechniken visuell bearbeitet werden.

Die gezeigten Techniken und Konzepte sind allgemein verwendbar und lassen sich daher auch jenseits der Softwareanalyse einsetzen, um ähnlich strukturierte Daten zu visualisieren. Mehrere qualitative Studien an Softwaresystemen in industriellem Maßstab stellen die Praktikabilität der Techniken dar. Die Ergebnisse sind erste Belege dafür, dass die Anwendung der Techniken in verschiedenen Szenarien nützliche Einsichten in die untersuchten Daten und deren Zusammenhänge liefert.

Acknowledgments

This thesis presents the results of my research in software visualization. First, and foremost, this thesis was made possible by my supervisor Prof. Dr. Jürgen Döllner who encouraged me to continue working in this field while I was finishing my Master's thesis. I want to thank him for offering me this possibility and his continuous support during my research; I learned a lot from him. I also want to thank Prof. Dr. Alexandru C. Telea from the University of Groningen and Prof. Dr. Guido Wirtz from the University of Bamberg for their agreement to review this thesis. Moreover, I want to thank Prof. Dr. Alexandru C. Telea for his commitment to several collaborative projects.

I am grateful to all persons who supported me during my research work; this includes giving feedback on my research, discussing ideas for improvements, and proofreading this thesis.

I very much enjoyed my time at the Computer Graphics Systems group. I thank all my colleagues and the group's always helpful and cheerful secretary Sabine Biewendt. I particularly want to thank my colleagues Dr. Johannes Bohnet, Stefan Voigt, Martin Beck, Sebastian Hahn, Daniel Limberger, Sebastian Kay Belle, Christine Lehmann, and the student Benjamin Karran for the research collaboration in various projects.

Last, but not least, I am deeply grateful to my family and friends for their loving support, friendship, and belief in my abilities which all in all gave me the energy for this work.

Several companies (alphabetically ordered) supported this work by generously providing access to their software systems:

- Francotyp-Postalia GmbH (`http://www.francotyp.com`)
- SAP AG and SAP Innovation Center Potsdam (`http://www.sap.com`)
- Software Diagnostics Technology GmbH (`http://www.softwarediagnostics.com`)
- virtualcitySYSTEMS GmbH (`http://www.virtualcitysystems.de`)

This work has been funded by the Research School on "Service-Oriented Systems Engineering" of the Hasso-Plattner-Institute, the ZIM program of the Federal Goverment of Germany (BMWi), and the InnoProfile Transfer research group "4DnD-Vis" of the Federal Ministry of Education and Research (BMBF), Germany.

Potsdam, Germany, January 29, 2014 *Jonas Trümper*

Contents

Chapter 1

Introduction

"There is an unstoppable trend toward increasing scale in many [software] *systems important to our society. Scale changes everything. These changes undermine the assumptions we routinely make in traditional software engineering approaches."* — Linda Northrop, 2013 [198]

The contributions of this thesis are mainly located in the field known as *software maintenance* that is one of the key elements in the typical software engineering lifecycle [96, 202]. The IEEE Computer Society defines it as *"the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment"* [1]. *Software analysis* is a method that operates on existing systems and can be used to extract information from such systems' artifacts. In practice, such systems, mainly legacy systems, form important building blocks of existing software infrastructures. They are often irreplaceable for several reasons and thus are typically in operation for decades. Their software maintenance, therefore, has to cope with this challenging continuous adaptation process and resulting negative side effects such as *software aging*. Results of software aging include violations of design principles, code duplication, and undocumented code [187, 202]. As a matter of fact, the maintenance of complex software systems (e.g., comprising more than 10^5 lines of code (LOC)) is thus responsible for a large portion of such software projects' total lifecycle costs. In a study, Erlikh [87] reports estimates of 85-90%.

In large software projects, there exist not only engineering perspectives, but also architectural and management perspectives. From the engineering perspective, *program comprehension*[1] denotes tasks performed to *understand* existing code, including its structure and behavior (i.e., its internal runtime dynamics) [159, 178]. So, engineers are concerned with numerous software artifacts that, as a whole, constitute a system's implementation. For various reasons, even understanding only parts of a large system's implementation is a difficult and tedious task [18, 64, 247, 280]. For example, it is typically hard for engineers to (1) establish the relationships between *dynamic* (e.g., control flow) and *static* aspects (e.g., structure) of a system [146, 194], and to (2) understand the effects of changed execution contexts on software behavior [158, 218]. The management perspective, on the other hand, sets a project's strategic direction. This includes bringing together end-user requirements, budget plans, and development activity. A prerequisite for this is assessment of a project's current status, which is known to be a difficult and erroneous task in practice [33, 263]. Management here requires transparency at a level which allows the assessment of cost

[1]The term 'program' in 'program comprehension' is used synonymously with 'software system' in literature and, where appropriate, in this thesis, too.

effectiveness and long-term development perspectives but omits any unnecessary technical details. For the decision-making process, management also needs to assess this information in the context of alternatives. Thus, management benefits from a view of development artifacts and the corresponding development process that is condensed to a level relevant to their decisions.

Software Analysis and Visualization in Software Maintenance

From the engineering as well as the architectural and management perspectives, we need to understand the structural and behavioral aspects of software systems. *Software analysis and visualization* aim to cope with these challenges. They provide both engineers and managers with techniques that simplify their work and enable insights into software artifacts and processes that are difficult to establish otherwise (e.g., by code reading). For this, software analysis commonly gathers and evaluates structural, behavioral, and evolutionary data using static (at compile-time) and dynamic (at runtime) analysis approaches, which are outlined next.

Software Analysis

Dynamic software analysis is primarily used to extract behavioral information [255], but can also extract related structural information. It can expose the interaction of software artifacts at runtime, including aspects such as late binding and data-dependent execution paths, which are generally hard to find via static analysis [69, 286]. In this thesis, *tracing* refers to techniques for dynamic analysis that capture control flow from user-space applications by means of *software traces* (or short: *traces*). *"The advantage of using event traces to understand software behavior is their ubiquity, and their (sometimes hardware assisted) ability to capture millions of incidents, state changes, and other artifacts"* [74]. Tracing generally captures a finite interval of a software system's execution on a certain level of abstraction and commonly yields only an incomplete picture of its structure and behavior: It can only capture those parts that contribute to the traced execution. Moreover, trace data is typically only valid for a specific execution scenario or even only a single execution of such a scenario.

In contrast, static analysis extracts information from a system by inspecting its artifacts (e.g., source code) at compile-time [196]. It can even provide a complete picture of a system's implementation, which also includes all possible control-flow paths in a code base, and "*holds for all executions of the program*" [78]; in this sense, static analysis is typically *sound*, while dynamic analysis is rarely sound [88]. Hence, correlating trace data with static-analysis data can serve both to complement the partial picture of dynamic analysis with the missing bits and pieces, and, vice versa, to reduce the complete picture of static analysis to a subset relevant to specific behavior [88].

Software Visualization

Software analysis typically generates massive raw data, so that even experienced engineers face difficulties inspecting these data [158, 202, 271]. They require tools to preprocess it: Either to find concise answers to specific questions, or to *explore* the data if there is no clear question that can be formulated beforehand. For example, *trace visualization* enables the exploratory visual analysis of software traces. In such exploratory use cases, visualization that presents the data in an interactive manner and permits the application of top-down and bottom-up exploration strategies can be very effective [25]. As Wirtz and Zhang point out, this is particularly true for concurrency phenomena: *“graphical representations are more adequate to describe parallel aspects than one-dimensional textual languages”* [283]. For this, visualization processes the data, generates abstractions, and finally maps the data to geometric representations, which then can be rendered. This is illustrated in the *visualization pipeline* [47] as shown in Fig. 1.1. For interactive visualization, such pipelines typically support partial executions where users can adjust parameters of individual pipeline stages and only re-run affected (i.e., subsequent) stages. This thesis’ contributions are mainly located in the fields of the preprocessing, mapping & abstraction, and rendering stages of the visualization pipeline.

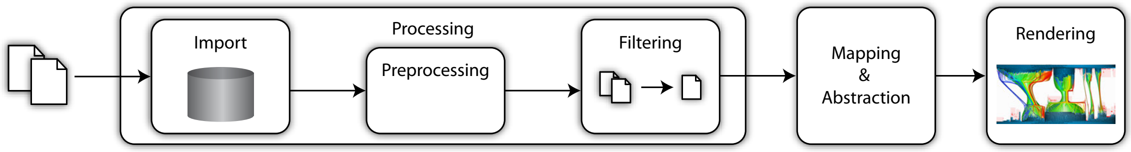


Figure 1.1: *Conceptual model of a visualization pipeline inspired by Card et al. [47]. Raw data is processed, which includes importing and converting, preprocessing to optimize it for further processing, and filtering. Subsequently, it is mapped to geometric representations, which can also include generating abstraction levels of the data. In the final step, the geometric representations are rendered by applying appropriate transformations and a potential rasterization for their display on pixel-based displays.*

1.1 Problem Statement and Aim of this Thesis

Parallelization has become a key means to increase execution speed of software [257] and is used on various scales, from small-scale microprocessors to large-scale distributed systems [144, 229]. It typically requires software-based solutions, which constitutes a challenge in software development and software analysis [257]. With this ever more widespread use of parallel systems, there is also a growing importance and need for suitable tools for behavioral analysis of such systems through tracing. For this, runtime data has to be captured and analyzed, which is a difficult task. This is particularly true for multi-threaded software systems, which are the focus of this thesis. They exhibit a characteristically close coupling of the parallel executions: Besides concurrency, thread interaction plays a major role in their execution and thus often is also crucial in understanding their behavior by means of software traces.

Moreover, trace analysis often not only requires a representation of the runtime events (such as function[2] calls), but also needs to correlate this behavior with other software artifacts such as software structure. For example, this can provide insight into which components are actually used in an execution. Such structural analysis needs to at least represent the software artifacts and their (hierarchical) composition; often, their inter-artifact dependencies are relevant as well. Both aspects, the behavioral analysis and the structural analysis, require a concise and accurate representation of the massive subject data. This poses a number of challenges, out of which seven are outlined next.

For better support of visual analysis of software behavior from the engineering perspective, one of the challenges involves (C_1) the visualization of multiple large traces from single-threaded or multi-threaded software systems. This needs to show multiple threads’ runtime behavior individually (each comprising hundreds of thousands of function calls), as well as concurrency of function calls. It should compactly represent traces to reduce the

[2]The term ‘function’ is used interchangeably with the terms ‘method’, ‘procedure’, ‘routine’, etc.

amount of user interaction needed. (C_2) To accommodate for the additional depiction of correlations between threads in terms of synchronization relationships, the visualization layout needs to be screen-space efficient. Another set of challenges is the visual correlation of traces with other software artifacts. In particular, the correspondence visualization needs to account for the circumstance that *time* is typically used as a primary spatial dimension in trace visualization. (C_3) To correlate traces with system structures that also comprise thousands of entities, a visualization needs to display software behavior, software structure, and the bidirectional relationships between the two. (C_4) Moreover, to visually correlate traces with other traces, it needs to show where and how software behavior (captured as traces) is similar and different, and to explain these (dis)similarities. The correspondence visualization needs to be able to represent changes in duration and order of function calls, swapped function calls, repetitions, and insertions/deletions.

Similarly, static analysis to support architectural and management perspectives needs to show graphs comprising many entities and relationships as well as entity attributes, and support visual aggregation to provide high-level representations of the graphs. It is thus faced with challenges such as (C_5) the visualization of multivariate compound graphs emerging from attributed system structures. Furthermore, another challenge is visually manipulating these large and complex compound graphs in what-if scenarios. (C_6) One needs to provide interaction mechanisms and (C_7) direct feedback for such manipulation with respect to change impact as well as change effort.

This thesis aims to extend and improve the tool set for visual software analysis of software systems written in *imperative* programming languages by addressing the challenges C_1-C_7 in dynamic and static analysis. It develops concepts and tools for the *exploratory visualization* of the respective data to support users in finding and retrieving information on the system artifacts concerned. In particular, trace-visualization techniques for visually correlating control-flow related aspects of traces with data from both dynamic and static analysis are presented. Moreover, complementary techniques for simultaneously analyzing system structures and evolution of related attributes are proposed. These aim at facilitating long-term planning of software architecture and supporting management decisions in software projects. The techniques' practicability is demonstrated by several qualitative studies in which the techniques are applied to industry-scale software systems.

The research questions of this thesis can be summarized as follows:

RQ1 *How can multi-threaded behavior of complex software systems be compactly represented in a visual manner? In particular, how can traces that capture function calls as well as synchronization between threads be visualized?*

RQ2 *How can traces of software behavior be visually correlated with other software artifacts such as software structure or other software behavior?*

RQ3 *How can software structure be analyzed and manipulated visually such that both the hierarchical composition of software artifacts and the relationships between those artifacts become visible?*

1.2 Techniques Presented in this Thesis

To address the aforementioned challenges, several visualization techniques are presented in this thesis (see Fig. 1.2). It starts by describing techniques for dynamic analysis of concurrent software behavior. In addition, related techniques are presented that target the visual correlation of software behavior with other software artifacts, which includes dynamic-static analysis. Second, techniques that target static analysis are outlined: They aim at analyzing and manipulating hierarchical structures and relationships therein.

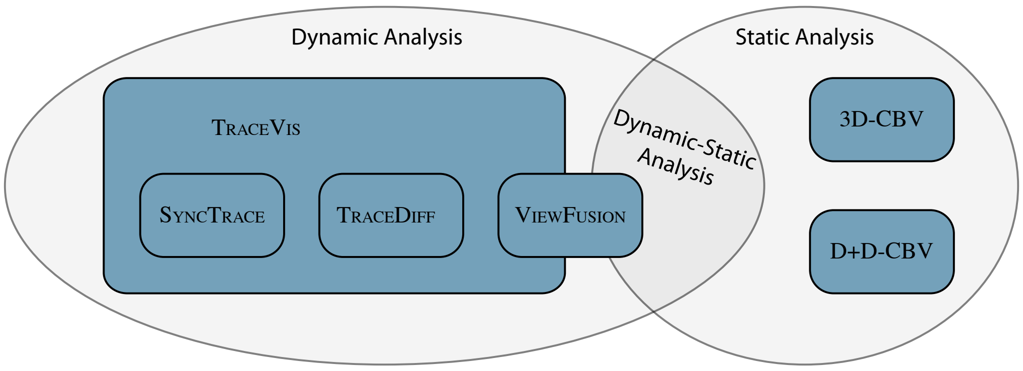


Figure 1.2: *A map of the techniques presented in this thesis.*

1.2.1 Analysis of Software Behavior

The techniques for behavior analysis, based on traces, are targeted towards software engineers and architects. With their design, the techniques follow the common understanding of exploration strategies applied in program comprehension: Users applying top-down and bottom-up exploration (with frequent switches between both) and cross-referencing [253, 272] (see also Section 2.3.1). The well-known information-seeking mantra *"overview, zoom and filter, and details on demand"* [237] is a common approach in supporting these strategies and is also used in the techniques presented. The following sections outline how these techniques relate to each other.

Analysis of Multi-Threaded Software Behavior

TraceVis is a basic concept and technique for compactly visualizing function-call sequences from multi-threaded traces ($\rightarrow C_1$). It serves as a fundamental visual design as well as a preprocessing stage for software traces, and constitutes a framework for all other trace-visualization techniques presented in this thesis.

SyncTrace is an extension of TraceVis to further support the analysis of correspondences between threads ($\rightarrow C_2$). It extends TraceVis by (1) providing improved visual scalability with respect to the number of threads that can be depicted simultaneously, and (2) it provides visualization metaphors to also show thread interactions in terms of blocking system calls such as shared lock usage, and synchronized input/output (IO) operations. The two contributions are described in Chapter 4.

Analysis of Correlated Behavior-Structure or Behavior-Behavior

ViewFusion, building upon TraceVis, is a method for correlating behavior-structure in *one* spatial area ($\rightarrow C_3$). It uses screen space arguably more efficiently than the common linked-views technique. Furthermore, ViewFusion causes less disruption during interaction, as the two views' average spatial distance is lower than with the linked-views technique.

TraceDiff, also based on TraceVis, targets the comparison of execution traces ($\rightarrow C_4$). Through a multi-scale design, it supports comparison of both coarse-grained (aggregated to high-level functionality) and fine-grained correspondences, defined on low-level function calls. Chapter 5 explains the contributions in detail.

1.2.2 Analysis of Software Structure and Related Artifacts

3D-CBV is a 3-dimensional extension of the circular bundle view (CBV) technique for analyzing multivariate compound graphs ($\rightarrow C_5$). It enables the additional depiction of visual variables and can thus be applied, for example, in perfective maintenance to assess software structure, dependencies and attributes of software entities. Drag&Drop-CBV (D+D-CBV) is a set of interaction techniques for the CBV to enable manipulation of the depicted structure in various ways ($\rightarrow C_6$). They provide direct feedback on the change impact of modifications ($\rightarrow C_7$) and enable manipulation without media disruption. These techniques are discussed in Chapter 6.

Chapter 2

Key Aspects of Software Maintenance and Software Visualization

---

This section is based on parts of the following publication(s):

- Martin Beck, Jonas Trümper, and Jürgen Döllner. A Visual Analysis and Design Tool for Planning Software Reengineerings. In Proceedings of the International Workshop on Visualizing Software for Understanding and Analysis, pages 54-61. IEEE, 2011. [294]
- Jonas Trümper and Jürgen Döllner. Extending Recommendation Systems with Software Maps. In Proceedings of the International ICSE Workshop on Recommendation Systems for Software Engineering (RSSE), pages 92-96. IEEE, 2012. [299]
- Jonas Trümper, Stefan Voigt, and Jürgen Döllner. Maintenance of Embedded Systems: Supporting Program Comprehension Using Dynamic Analysis. In Proceedings of the International ICSE Workshop on Software Engineering for Embedded Systems, pages 4396–4402. IEEE, 2012. [304]
- Benjamin Karran, Jonas Trümper, and Jürgen Döllner. SyncTrace: Visual Thread-Interplay Analysis. In Proceedings (electronic) of the Working Conference on Software Visualization, pages 1-8. IEEE, 2013. [296]

Software maintenance, concerned with *“adapting* [software systems] *to the needs of a changing environment”* [162], aims at keeping a system operational after its initial deployment. This includes changing both a system’s functional aspects (to keep it useful) and its non-functional aspects (to keep it maintainable) [1, 39, 162]. Functional aspects typically refer to externally visible *features* such as the ability to read a certain file format. Non-functional aspects, for example, denote a system’s performance characteristics, scalability, and *maintainability.*

Changing a complex system’s functional aspects often leads to negative side effects, such as non-functional anomalies or defect regressions, that in turn can impair its maintainability. Reasons for this ‘automatic’ decrease in maintainability can be manifold: Prominent ones are, first, the size of the code base that makes it hard, if not impossible, for humans to memorize a system as a whole and to assess the ‘big picture’ [64]. Instead, certain individuals often understand some parts of a code base well, while others are experts in other parts of the same code base. Second, engineers trying to understand code previously unknown to them may misunderstand the code or its documentation [202]. Third, documentation is often outdated, for example because tight project schedules do not allow enough time for changing code *and* updating affected parts of the documentation [94]. Fourth, with constantly changing teams of engineers, there is a probability that knowledge of undocumented code ultimately gets 'lost'.

As a result, large systems are often insufficiently understood over the long term. By using or changing existing code, engineers then 'risk' introducing design anomalies: They likely use or change code in a way that was not intended in its initial design [86, 202]. Moreover, Lim et al. emphasize that practitioners often perceive *"intentional decisions to trade off competing concerns during development"* [169] as a reason for impaired maintainability, i.e., even though they appear to understand the respective software systems well, it is hard for them to ensure good maintainability while still accounting for all project constraints.

For these and other reasons, it is thus generally hard to understand and maintain large software systems. In the long run, the fact that large *and* aged systems are even harder to understand only adds to this problem. The term *software aging* denotes the frequent effect that software, being in operation for longer periods of time, is becoming harder and harder to maintain. A software project is then likely to end up in a vicious cycle in which software aging itself leads to even more software aging [86, 277].

Low maintainability is commonly considered to increase a project's overall costs: Hard-to-maintain code typically requires additional effort for its understanding and modification [277]. From a business perspective, this effect of software aging is similar to that of financial debt. Consequently, Cunningham coined the term *technical debt* [70]. Brown et al. likewise state that *"[...] like financial debt, technical debt incurs interest payments in the form of increased future costs owing to earlier quick and dirty design and implementation choices. Like financial debt, sometimes technical debt can be necessary. One can continue paying interest, or pay down the principal by re-architecting and refactoring to reduce future interest payments"* [41]. In this sense, technical debt refers to the circumstance that the effects of software aging often do not become evident immediately, but will inevitably inflict *"charges against future revenue"* [63]. The term 'technical debt' sets the *"focus on the tradeoffs between expedient short-term decisions and the resulting, potentially crippling, long-term costs"* [169]. Or, as one participant of a study on technical debt put it, "[s]*ometimes, when dropping requirements wasn't an option, they provided solutions that fulfilled the requirements only enough for customer acceptance, knowing that those solutions' implementation still required work"* [169].

As hinted above, trading technical debt for business values is not generally negative and can make sense in numerous cases. Therefore, Lim et al. instead refer to this 'tradeoff' as a "balancing act" [169]. *"So, although the participants recognized that taking on technical debt always affects software quality in some way, numerous scenarios existed in which the technical debt's market benefits clearly outweighed the hit on quality"* [169]: For instance, it offers the chance to gain early feedback on prototypes or get to market before other competitors do.

The concept of of technical debt and the awareness of it have gained traction in research and industry in the last decade [41, 135, 169]. Some even argue that technical debt, which potentially reduces future venue, should be included in a company's books just as financial debt is already included [63].

If software maintenance and the balancing act involving technical debt actually fail, we know that software systems cannot be upgraded, and are re-implemented, or in the worst case, are abandoned. The following examples are only three out of many:

- *"A* [German] *electronic health insurance card, which should store all information relevant to a patient's health, was supposed to make treatments more efficient and cheaper, as well as facilitate the physicians' work. The decree is 10 years old. The project to date* [(2013)] *has cost more than €700 million in insurance premiums. The only visible result: Insurance cards now bear a photo* [of the patient]*."* [172] (Translation by the author)
- *"According to information from FOCUS, the* [German] *Federal Ministers and State Ministers of Finance aim to declare the joint software development* [of Fiscus]*, which started in 1991, as failed. The programming attempts to date have cost approximately €250 to 900 million."* [2] (Translation by the author)
- *"The U.S. Federal Aviation Administration spent $2.6 billion trying to upgrade its air-traffic-control system, only to cancel the project in 1994."* [53]

So, despite trading maintainability for other project goals, it is essential to keep a software system useful and maintainable in the long run [39, 162]. The IEEE defines the activities of *perfective* and *preventive* software maintenance [1] in order to counterbalance this problem. Among other aims, these activities both serve to ensure sufficient code quality, e.g., by software analysis.

2.1 Software Analysis

Software analysis *"aims to obtain insightful and actionable information from software artifacts that help practitioners accomplish tasks related to software development, systems, and users"* [288]. This section outlines selected aspects in this respect and discusses several key techniques used in software analysis: Feature location; reengineering and refactoring, forward engineering and reverse engineering, and behavioral and structural analysis.

2.1.1 Feature Location

Feature location is the task concerned with *"identifying the parts of the source code that correspond to a specific functionality"* [208] (i.e., feature), such as reading input data, and which *"is needed whenever a change is to be made"* [213]. Identifying the particular software artifacts is often non-trivial, especially in cases of incomplete or outdated documentation. One main reason is that software is commonly built upon the principles for software reuse such as *modularity* [201]: Its goal is to build software artifacts as modular and self-contained as possible – i.e., it aims for a low coupling between modules – to better support the reuse of individual software artifacts as independent building blocks. As a result, though, the implementation of a – especially higher-level – feature is typically not only contained in a single software artifact but is scattered across many software artifacts [213].

There exist a variety of techniques for feature location such as 'simple' text search in source-code artifacts. More complicated techniques include tracing the execution of the corresponding features and subsequently identifying which components were active during execution of the feature [280]. For this technique, Sato et al. highlight the importance of correctly identifying correspondences between traced behavior and feature [231], which can be hard if the traced behavior potentially corresponds to a series of features.

2.1.2 Reengineering and Refactoring

As part of perfective and preventive maintenance, *reengineering* and *refactoring* are commonly used methods to address maintainability issues in large-scale software systems. By changing a system's representation *without* changing its behavior – 'only' *restructuring* it –, these methods aim at improving a system's architectural and code quality [56, 96, 207]. In fact, numerous newer, particularly agile, software development methods include refactoring or reengineering phases as regular activity [21, 61, 96]. Refactoring addresses small-scale changes, such as splitting up larger methods into smaller ones, and is typically applied as a drive-by action by engineers. [191]. In contrast, reengineering targets larger-scale and large-scale changes that can affect a set of classes, subsystems or even an entire system – including major revisions of a software system's design – and is mostly planned by software architects [191]. Murphy-Hill and Black distinguish these two types of restructuring using a dental metaphor [191]: (*Floss*) refactoring and *Root-canal* refactoring (reengineering).

2.1.3 Forward Engineering versus Reverse Engineering

Forward engineering is a process in which higher-level descriptions and concepts regarding a system's implementation are iteratively refined to finally yield executable code [153]. Hoc and Nguyen similarly define 'programming' as *"the process of transforming a mental plan of desired actions for a computer into a representation that can be understood by the computer"* [120]. Conversely, *reverse engineering* can be thought of as the inverse process: Recovering information about an existing software system, including its structure, dependencies, and behavior [56, 277]. The goal of this recovery is to create higher-level abstractions of a system's implementation, e.g., translate source code into models, typically to reduce the amount or the complexity of the original information.

Reverse engineering does not change the subject system, while forward engineering creates or changes it [264]. Due to the fact that in practice only very few software projects do start from scratch, forward engineering can rarely be seen in isolation. Reverse engineering and feature location, then, are a prerequisite for any form of forward engineering [56]. To build upon existing source code or binary libraries that are not well documented, reverse engineering is needed to understand their purpose and usage. For example, reengineering or refactoring an existing code base (forward engineering) is only possible if its structure is known/reconstructed (reverse engineering) [56, 151].

Tilley identifies *"data gathering, knowledge management, and information exploration"* [264] as canonical activities in reverse engineering. Data gathering (A1) describes the process of extracting information (the raw data) from a system's implementation. In knowledge management (A2), the raw data is abstracted to form higher-level representations of a system. Finally, information exploration (A3) is the process of analyzing, correlating, and understanding these higher-level representations. The techniques presented in this thesis mainly address activities A2 and A3.

2.1.4 Behavioral Analysis

Behavioral analysis is classified as a dynamic technique concerned with a software system's *runtime* behavior. In this thesis, the term 'software behavior' refers to the time-dependent process that is observable when an instance of a software system is executed. In a more general fashion, 'runtime behavior' refers to any time-dependent process observable in this context, for example the behavior of a single thread or of a distributed system that runs multiple instances of a software system.

There are many techniques for this kind of dynamic analysis such as conventional debuggers and profilers. These debuggers, however, only yield snapshots of a system's state and do not capture and present data about its behavior as a whole [168]. Likewise, profilers only sample system states at specific points in time [108]. Therefore, this thesis concentrates on tracing as a more powerful technique that captures a stream of runtime events. In tracing, *instrumentation* denotes the process of augmenting a *target system's* execution using *probes* that record trace data. Software-based solutions include instrumentation of a system's executable code [32] or its execution environment [233]. They allow for tracing in a nontransparent manner only. That is, the injected probes may interfere with the target system's execution. Hardware-based probes yield transparent trace recordings, but are less commonly used in practice due to implementation challenges and limited versatility [304]. In both cases, the target system is then executed within a given scenario, which includes possible input data, and configuration of the system and its execution environment. This has the benefit of reducing the union of all possible outcomes (accessible via static analysis) to a single outcome.

Interference with the Target System

The results of nontransparent behavioral analysis have to be interpreted with care: For many possible reasons, the results of such runs are typically not generalizable to future runs of the same scenario, not to mention different scenarios. Most prominently, this is due to non-deterministic execution [26, 160, 225], which can significantly alter the outcome of future runs of even the same scenario. Non-determinism is almost ubiquitous in today's computers and operating systems [26, 225] that support multi-tasking (time sharing among multiple processes) and multi-threaded execution (time sharing among multiple pieces of code within a single process), variable processor clock speeds [42], etc. Moreover, even the slightest variation in the execution scenario can have a significant impact on the resulting behavior of a software system [26, 225].

In general, tracing a software system by means of software-based probes includes the introduction of a runtime performance-overhead, which alters its timing behavior [170]. In the context of tracing, this alteration is often referred to as the *probe effect* [100]. In particular, if timing is critical, the probe effect may alter a system's runtime behavior unintentionally. This phenomenon is similar to the *uncertainty principle* in quantum mechanics put forth by Heisenberg [118]: Simultaneously observing so-called “complementary variables” of a particle suffers certain non-technical precision limits. That is, we cannot be sure of their precise values if we observe them at the same time. Likewise, there can be uncertainty in software-based tracing of software behavior, since observing its behavior can at the same time influence this very behavior.

Multi-threaded software systems are thus also prone to interference issues. Deadlocks or race conditions, for instance, may be caused or prevented if the timing is changed. Moreover, full instrumentation generates massive trace data, of which typically only smaller portions are relevant for a given task. For instance, frequently executed utility functions, e.g., string concatenation, typically contribute to a large extent to a trace's size, but not to a better comprehensibility of the data [112].

Selective Instrumentation for Reducing Runtime Overhead and Trace Size

To reduce these overheads and the likelihood of such interferences, we can instrument only those parts of the system's implementation (*selective instrumentation*) that are relevant for the functionality to be debugged [91, 182]. This approach make possible low

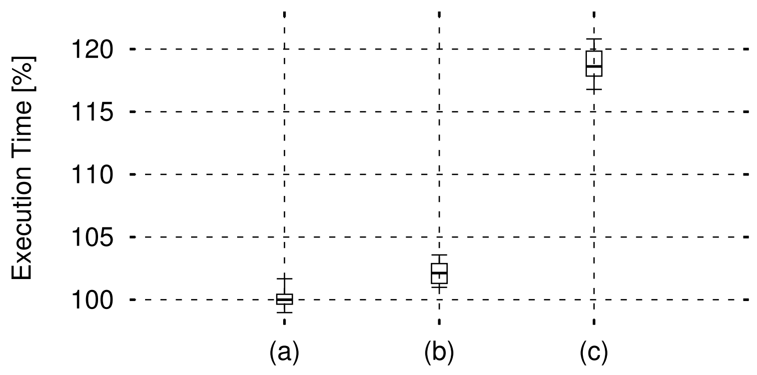


Figure 2.1: *Performance overhead comparison for a tracing implementation on an embedded system of Francotyp-Postalia GmbH [304]: Normal run of the system with no instrumentation (a), instrumented, but disabled (b), and selectively instrumented (c). The 100% mark is the mean execution time of the original binary.*

instrumentation overheads – despite costly event registration mechanisms (for an example, see Fig. 2.1): The methods with shorter execution times are excluded from tracing; hence, the performance overhead is only added to more costly calls (longer execution times) which results in a lower *relative* and therefore lower *absolute* overhead. Similarly, trace compression [217] is a means to reduce the size of recorded data.

Methods for selective instrumentation range from manual to semi-automatic. Especially in application scenarios where the target system has only low processing power (e.g., embedded systems), a manual method is preferable since it does not include any runtime processing. A possible workflow for this can contain the following four steps [304]: (1) An input set of functions for selective instrumentation can typically be gathered from the engineers' knowledge. If not, however, a similar approach to the aforementioned utility-function classification is possible: Based on execution frequencies, functions are iteratively de-selected from instrumentation until sufficiently low overhead is achieved. For this, engineers initially run the system with full instrumentation and record a trace. (2) Next, the engineers explore this first trace and gain an understanding of which parts of the system's implementation are relevant for the given task, and discard the first trace. Particularly, they identify low-level methods that have short execution times and are frequently executed (thus often contained in the trace), but do not contribute to the process of understanding (Fig. 2.2). (3) Then, engineers subjectively decide which parts of the implementation will be excluded from further tracing. (4) Finally, they exercise the functionality again with only selective instrumentation.

Similar to the manual one, a semi-automatic method [32] does not require recording an initial trace. Instead, it automatically disables tracing at runtime for frequently executed parts based on a manually defined execution-frequency threshold. As such, this heuristic may accidentally exclude parts from the tracing that would be relevant for the subsequent analysis. A combination of both the heuristic and the manual method would allow for automatic generation of an initial exclusion set, which is then manually edited to ensure relevant functions are included in the instrumentation.

In our experience, these approaches work well for target systems used in this thesis. A limitation of the semi-automatic approach, however, is its reduced applicability for small-sized or highly time critical systems that cannot additionally perform the execution-frequency analysis at runtime. One example of such systems involves specific embedded real-time systems in the automotive domain.

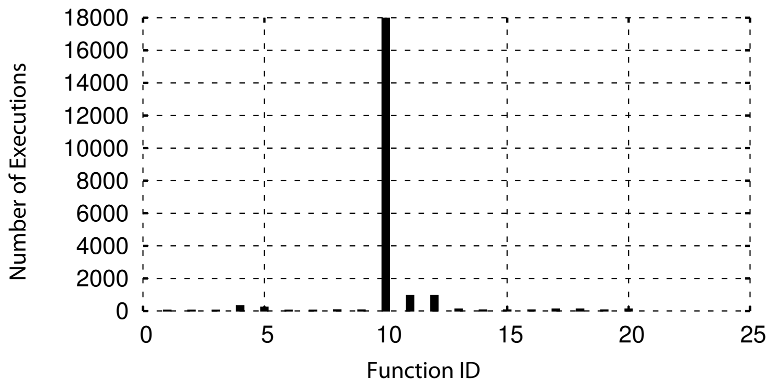


Figure 2.2: *Execution frequencies from an example trace of an embedded system [304]: In this respect, function with ID 10 is clearly an outlier. Under the assumption that it does not contribute essential information to analyzed runtime behavior, the trace overhead could be massively reduced by not instrumenting this function.*

Alternative approaches compute an optimal instrumentation prior to execution [91, 204], e.g., to guarantee worst-case execution times with injected tracing probes. Such an approach could simplify the aforementioned selection of functions to be instrumented (cf. Section 2.1.4). Similarly, sampling can be used [167] to keep runtime overhead low. That, however, reduces precision. For instance, in debugging scenarios, the execution of functions of specific interest may be missed.

A Formal Model of Software Traces

A software trace describes the behavior of a software system. In this thesis, this trace is generally defined on the level of function calls, hence also the name *function-boundary trace* (FBT), since it describes software behavior as a sequence of 'function entry'/'function exit' events. A software trace $\mathbb{T}$ can be modeled as a set of *thread traces* T_m in which each thread trace describes a single thread's runtime behavior and $m \in \mathbb{N}$ is a numerical thread identifier (*thread ID*).

It can thus be modeled as

$$\mathbb{T} = \{T_m\}. \tag{2.1}$$

A thread trace can further be defined as a tree $T = \{f\}$ of function calls. Such datasets can also emerge in different scenarios, e.g., state machine simulations. A function call can be modeled as

$$f = (F, t^s \in \mathbb{R}^+, t^e \in \mathbb{R}^+, p \in T, C = \{c_i \in T\}). \tag{2.2}$$

Here, F represents the declaration, or identity, of the called function. Depending on the application and data availability, this can be a full syntax tree description of the function declaration or just the fully qualified function name. The values t^s and t^e, where $t^s < t^e$, represent the 'start' and 'end' moments of the call, and $\mathrm{dur}(f) = t^e - t^s$ its duration. The parent, or caller, of f is denoted by p, and p is undefined for the root of T_m ($p = \texttt{undef}$). The set C holds the children, or callees, of f, ordered by call times, i.e., $\forall c_i \in C, c_j \in C, i < j | t^e(c_i) < t^s(c_j)$. Further, we denote the call stack rooted at f by $S(f)$ and its depth as

$$\mathrm{depth}(f) = \begin{cases} 0 & \text{if } p(f) = \texttt{undef} \\ \mathrm{depth}(p(f)) + 1 & \text{otherwise} \end{cases}. \tag{2.3}$$

Similarly, maxDepth(T) returns the maximum depth of all function calls in a thread trace.

Although the definition above is given in the context of software traces, it is applicable to other domains, such as network communication, where hierarchical event sequences exist. A similar definition for thread traces is given by Bohnet [32].

2.1.5 Structural Analysis, Software Metrics, and Software Structure

Software typically consists of hierarchically structured entities. In this hierarchy, leaf nodes correspond to low-level software entities, e.g., files or classes, and non-leaf nodes represent architectural entities such as modules or packages. This hierarchical organization supports the separation of concerns by grouping similar functionalities into distinct software entities. Additionally, usage or call relationships typically interconnect these software entities.

Structural analysis yields information on this *hierarchy* of modules and *relationships* within a system's implementation. Together, the hierarchy and relationships will be referred to as the *structure* in what follows. Its analysis is typically done at compile-time (static), e.g., by parsing source code, binaries, and related artifacts such as design documents.

A Formal Model of Software Structures

To cover a broad range of distinct software systems, in particular aged and legacy systems, a generic model of a software's structure is required. Such a model supports multiple programming paradigms and intermixed software systems, as it abstracts from specific programming languages.

Hence, we model a higher-level representation of a system's structure as a directed acyclic graph (DAG), or compound directed graph $\mathbb{S}$ as a tuple of tree $H = \{n\}$ and relationships R:

$$\mathbb{S} = (H, R). \tag{2.4}$$

A node $n \in H$ can then be modeled as

$$n = (N, p \in H). \tag{2.5}$$

N represents the node's identity such as the full path of a source-code file. As the graph is hierarchical, each non-root node has a parent p; p is undefined for the root node ($p = \texttt{undef}$). Similarly, each non-leaf node n has a set of children $C(n) = \{n_i\}$, where any $n_i \in C(n)$ cannot be a parent of n (acyclic graph). For leaf nodes $C(n) = \emptyset$. We call $S(n)$ the subtree rooted at node n. Relationships R (directed edges between *leaf*-nodes) is given by a set of tuples as follows:

$$R = \{(n_i, n_j) | n_i, n_j \in H, C(n_i) = \emptyset, C(n_j) = \emptyset\}. \tag{2.6}$$

This concept of representing a software's modular hierarchy and dependencies between its entities as a compound directed graph is not limited to common programming languages such as C++ or Java. For example, an SAP ABAP software system or a legacy COBOL system consists of a similar modular hierarchy, although different types of entities and dependencies may be used.

Moreover, compound graphs exist in various other domains. For instance, relationships between persons can be grouped hierarchically by country, essentially forming this kind of compound graph. A similar model of a compound graph is given by Burch [43].

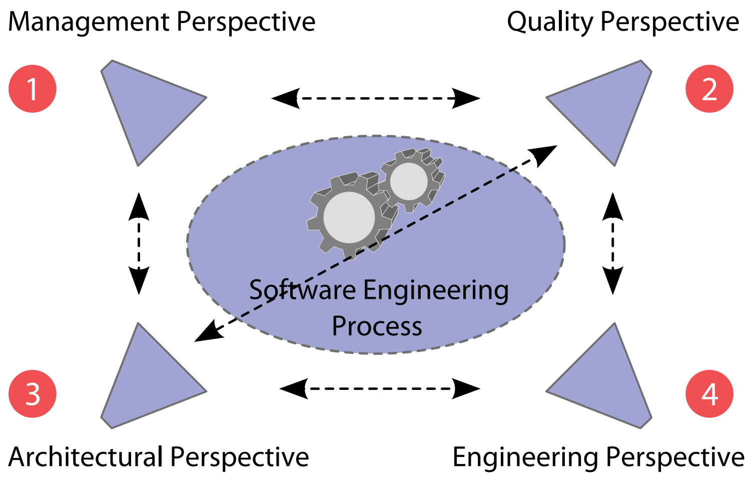


Figure 2.3: *Typical perspectives in larger software engineering projects [299]. Links between perspectives represent communication channels.*

Software Metrics

Structural analysis also forms the base for software metrics. Here, several 'atomic' figures, such as the measure of LOCs, as well as (linear) combinations of atomic figures as a 'complex' number, are used to measure properties of a code base [223]. A body of metrics addresses the maintainability [62] and complexity [177] of source code and is often used to implement monitoring facilities for software quality.

2.2 Perspectives on Software Projects and their Challenges

Orthogonal to the previously presented techniques that are applied in software engineering, various methodologies exist for describing and formalizing software engineering processes. This includes the Spiral model [31], the Rational Unified Process [153], and agile methods such as Extreme Programming (XP) [21] and Scrum [61]. Unsurprisingly, this literature does not employ a consistent terminology to denote the various stakeholders present in larger software development projects: While some of them explicitly state *roles* to be represented by humans, others do not (e.g., the Spiral model), and instead concentrate on the process itself. Some methodologies define roles orthogonal to other methodologies (e.g., RUP vs. Scrum).

Regardless of the methodology used for software development and despite these differences in the literature, in many larger software development projects there exist a number of key *perspectives* on the underlying process and its deliverables [299] (Fig. 2.3): First, these perspectives differ in terms of abstraction level and, second, in information needs and communication needs. They are different from the aforementioned 'roles' in the sense that they are more general and usually include many natural persons.

There does exist software-engineering literature that extensively discusses the information needs of the engineering perspective [147, 158, 159]. The literature, though, generally does not focus on *interconnected views* of these perspectives and their needs. Thus, this thesis includes a brief discussion of these perspectives (management, quality, architecture, and engineering) and their key information and communication needs based on observations we have made in large industrial software projects.

Management Perspective

Personnel in this perspective (1 in Fig. 2.3) are responsible for ongoing work and future directions, and ultimately decide on resources. They are typically non-technical staff and thus it is essential for them to obtain condensed, high-level information as overview of a project's state (M1). However, during subsequent decision-making processes, they may need on-demand access to additional, detailed information (M2). Some of this information can be retrieved from up-to-date fact sheets of projects (M3). In all remaining cases, personnel in this perspective also communicate with other representatives of the management perspective as well as the quality perspective and the architectural perspective (M4). A representation of the subject information under discussion that is understood by all participating personnel is essential then (M5).

Quality Perspective

The quality perspective's (2 in Fig. 2.3) main objective is to balance internal software quality (e.g., maintainable code) and external software quality (functionality visible to users). With respect to external software quality, requirements, their representation as tests, and their realization are important subjects when representatives of this perspective communicate with personnel coming from the management perspective. In addition, they need to constantly determine how well each sub-module's functionality is secured by tests and where to invest effort in future testing (Q1). For managing internal software quality, up-to-date information on system structure (Q2), on internal structure of modules (Q3), and assessments of maintainability and change impact (Q4) are all important factors [147].

Architectural Perspective

Architects (3 in Fig. 2.3) are in charge of a system's architecture and coordinate development efforts, so they need to be aware of a system's entire structure (A1) and current development activity (A2). In addition, working out how to meet user requirements with a software-based solution to implement the 'big picture' [153], they are in contact with both the management perspective and the engineering perspective. Therefore, they constantly need to translate between both worlds, which in turn requires suitable multi-level representations of systems (A3). Further, their main concerns on a system-wide level include (future) maintainability and long-term risks. Essential methods in this context include refactoring and restructuring. Both methods require information on how a system's structure can be changed (A4), the potential change impact [159] (A5), and change effort (A6).

Engineering Perspective

An engineer (4 in Fig. 2.3) is mainly concerned with a system's representation as source code. In cooperation with architects, they work out how to realize features and how to integrate them into an existing code base [64, 159]. Over time, they acquire expert knowledge on specific parts of a system's implementation on which they work. However, when it comes to the different parts, they first need to acquire in-depth knowledge of these (program comprehension, E1) [271]. In a more detailed, technical way than architects, they require information on how a system's behavior and structure can be changed (E2), the potential change impact [159] (E3), and change effort (E4). In addition, they typically have to advocate the engineer's view of the code base, i.e., argue for favoring maintainability actions rather than constantly introducing new features [33]. For this, tools and representations that make maintainability visible to non-technical staff are needed (E5).

Perspectives' Information Needs: An Overview

By analogy, with their individual viewing angles as shown in Fig. 2.3, each perspective has its own view and abstraction of a software engineering process. This is reflected in their different requirements with respect to information needed and information to be communicated (Table 2.1). As a general rule, the more abstract a perspective's view on software-engineering processes, the more multi-dimensional and holistic (i.e., 'strategic') are its information and communication needs. On the other hand, the more detailed the view from one perspective is, the more concrete and analytical (i.e., 'technical') are its needs.

Perspective	Id	Description of Needs
Management	M1	Condensed, high-level information
	M2	On-demand context information
	M3	Up-to-date overview fact-sheets
	M4	Discussions with other perspectives
	M5	Representation understood by all parties
Quality	Q1	Information on where to invest testing efforts
	Q2	Up-to-date information on system structure
	Q3	Up-to-date information on internal module structure
	Q4	Information on maintainability and change impact
Architecture	A1	Up-to-date information on system structure
	A2	Up-to-date information on development activity
	A3	Multi-level representation of the system or facts
	A4	Possibilities of restructuring
	A5	Change impact assessments
	A6	Change effort assessments
Engineering	E1	Up-to-date information on system behavior and structure
	E2	Possible modifications to behavior and structure
	E3	Change impact assessments
	E4	Change effort assessments
	E5	Representation understood by all parties

Table 2.1: *Needs overview by perspective.*

2.3 In Depth: Engineering and Management Perspectives

This section takes a closer look at both the engineering and management perspectives. As the former is the primary focus of this thesis, it is also outlined in more detail than the latter.

2.3.1 Program Comprehension: An Engineering Perspective

Program comprehension (or *program understanding*) is typically a difficult and time-consuming task. Major reasons for this include the size of a system's implementation, outdated documentation, technical debt and software aging, and constantly changing engineer teams [202]. According to several sources, program comprehension is reported to account for 30-60% of the overall maintenance costs of large software systems [64, 247, 280]. Not only due to the constant fluctuations in personnel, insufficiently understood code is a major pain here: Knowledge of the original rationale for and the purpose of pieces of code are (temporarily) lost. LaToza further reports in a study of 371 questions that 179 engineers from Microsoft perceive in their everyday software-engineering work as hard-to-answer [158].

One of the main objectives of program comprehension is thus to recover and/or gain sufficiently detailed knowledge on a system's code base – including its syntactic as well as semantic meanings [238] – to counteract software aging. Use-cases involve understanding a software's structure (e.g., its hierarchy of artifacts) and behavior (e.g., order and duration of function calls), and correlating structure and behavior. Hence, program comprehension is a largely exploratory task, in which engineers typically apply *top-down*, *bottom-up*, and *cross-referencing* exploration strategies [253, 272] and regularly switch between them [272].

Biggerstaff et al. define what constitutes human understanding of a program as follows:

Definition 1 (Program Understanding as a Conditional State (Biggerstaff et al., 1993))**.** *"A person understands a program when they are able to explain the program, its structure, its behavior, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program."* [28]

For this, engineers read and/or execute code, and draw conclusions from available documentation to effectively form hypotheses about the system's actual implementation, which they subsequently verify or falsify [159, 164, 178]. From verified hypotheses, they aim to build a mental model [132, 145, 253] that they often require to navigate and modify a system's source code and related artifacts. This model, then, represents a system's structure and behavior on different scales – or, as Mayrhauser and Vans put it: This model is *"one plan composed of many subplans representing different levels of abstraction"* [273].

Tilley defines this process of gaining program understanding as follows:

Definition 2 (Program Understanding as a Process (Tilley, 2000))**.** *"The goal of program understanding is to acquire sufficient knowledge about a software system so that it can evolve in a disciplined manner. The essence of program understanding is identifying artifacts and understanding their relationships; this process is essentially pattern matching at various abstraction levels. It involves the identification, manipulation, and exploration of artifacts in a particular representation of a subject system via mental pattern recognition by the software engineer and the aggregation of these artifacts to form more abstract system representations."* [264]

Likewise, Biggerstaff et al. state that:

Definition 3 (Program Understanding as a Process (Biggerstaff et al., 1993))**.** *"If a person starts to build an understanding of a heretofore unknown program or portion of a program, he or she must create or reconstruct the informal, human oriented expression of computational intent through a process of analysis, experimentation, guessing and crossword puzzle-like assembly. Importantly, as the informal concepts are discovered and interrelated concept by concept, they are simultaneously associated with or assigned to the specific implementation structures within the program (and its operational context) that are the concrete instances of those concepts."* [28]

2.3.1.1 Common Strategies for Program Comprehension

Top-Down and Bottom-Up Strategies In top-down exploration, engineers navigate from higher-level constructs to lower-level constructs [272]. This strategy is typically applied in (partially) known systems, in which also the purpose of such higher-level constructs is known beforehand [244]. This is typically the case if a piece of code or its *type* (e.g., a linked-list implementation) is familiar to an engineer.

In contrast, bottom-up exploration is usually applied if code or its type is unknown [205]. Here, engineers start with *"beacons to identify elementary blocks of code (control primes) in the program"* [205]. Using these 'beacons', they navigate the code and iteratively aggregate multiple recognized lower-level constructs (or beacons) to higher-level constructs, effectively building a high-level representation of the code. Mayrhauser and Vans name this process of aggregation the *"chunking strategy"* [273]. Moreover, there is evidence in studies that insight into software behavior is often key for engineers to build their mental model, i.e., they start building up the model by analyzing control flow [205] and only subsequently assess the data involved and data flow.

In both top-down and bottom-up strategies, lower-level constructs are (de)composed to form higher-level constructs and effectively form a DAG. In this *abstraction graph*, edges represent parent-child relationships. Hence, higher-level constructs can in fact be understood as parents of lower-level constructs.

Cross-Referencing Strategies Orthogonal to the former two strategies, cross-referencing is used to express relationships between different levels of abstraction such as the level of control flow and the functional level. In contrast to the former two strategies, engineers here insert complementary edges into their abstraction graph that connect any nodes from different levels, not necessarily obeying the parent-child relationships of the original graph. Mayrhauser and Vans state that *"cross-referencing is thus an integral part of building a complete mental representation across all levels of abstraction"* [273].

2.3.2 Project Management: A Strategic Perspective

On the management side, personnel seek to assess a project's current status in order to determine its future (strategic) direction in terms of features and development activity. However, they often have problems in the assessment. Thayer et al. name the following two major reasons for this [263]: *"programmers did not tell the truth (or did not know the truth) about the status of their programs"* and *"the true status of the project was never known"*. In other words, it is hard for management to keep pace with development progress, i.e., where and how much development effort is being invested, and where and how much effort should be invested to circumvent future maintenance problems. Assuming engineers know and tell the truth, management still often cannot invest enough time for the assessment or, even if they could, they usually have no software-engineering background, which makes it hard for them to understand the technicalities [33].

Moreover, business factors are highly influential here: *"Chief among the business factors* [for project failures] *are competition and the need to cut costs. Increasingly, senior managers expect IT departments to do more with less and do it faster than before; they view software projects not as investments but as pure costs that must be controlled"* [53].

So, when both parties communicate, the absence of a common language, the presence of misunderstandings, and different goals imply unnecessary communication overhead and errors such as the unknown true status of a project. This phenomenon can be referred to as an *information gap* [33] or *communication gap* [299]; the latter will be used in this thesis. Thus, management faces difficulties fulfilling the two opposing goals: good *internal software quality* (perceived by engineers, software maintainability) and good *external software quality* (perceived by users, functional completeness) [33]. It tends to favor *visible* changes to external quality. One of the reasons for this is the 'invisibility' of internal quality to non-technical staff and non-measurable return on investment for efforts devoted to it. That is, the *future* value of investments in internal quality is hard – if not impossible – to quantify, but still important for a successful project. Thus, none of the two goals (internal vs. external quality) can be neglected, so engineers and management have to bridge the communication gap and agree on compromises. Consequently, it is important not only to find reasons why a number of actions should be undertaken first, but these reasons also have to be visible to and reasonable for all stakeholders.

2.4 Visualization, Software Visualization, Information Visualization, and Scientific Visualization

Most work presented in this thesis builds upon concepts, methods, and techniques from the field of visualization applied in the software-engineering context. This section, therefore, outlines the basic idea of visualization and its sub-disciplines, in particular software visualization.

Keller and Tergan state that, from the viewpoint of psychologists, the term (information) visualization is used *"to signify a representational mode (as opposed to verbal descriptions of subject-matter content* [i.e., data]*) used to illustrate in a visual-spatial manner, for example, objects, dynamic systems, events, processes, and procedures"* [139]. When defining software visualization, Price et al. also state that *"in the Oxford English Dictionary (p. 699) the first six of the seven definitions of 'visual' relate to information gained from the use of the human eye, while the seventh suggests the conveyance of a mental image"* [209, 210]. Likewise, 'visualization' is defined in the Oxford English Dictionary as:

Definition 4 (Visualization (Oxford English Dictionary))**.** Visualization is *"the power or process of forming a mental picture or vision of something not actually present to the sight"* [242, p. 700].

Therefore, Price et al. conclude that *"a visualization can result from input from any combination of the human senses"* [209]. According to Ware, a visualization is intended as *"an external artifact supporting decision making"* [276]. So, in literature there exist those two somehow conflicting viewpoints: First, that *"visualization is an activity which a human being engages in, and that it is a cognitive activity [...]; in other words, it goes on in the mind"* [246], and second, that a visualization is *"an external artifact"* [276]. Spence suggests discerning the terms 'visualization' as a cognitive activity that is internal to a human's mind and 'visualization tools' as a means to support this cognitive activity [246].

Nevertheless, both sides agree that the purpose of visualization is *"gaining insight and understanding"* [246], that it can facilitate with handling of massive amounts of data, finding unanticipated properties of data, finding patterns in data, *"understanding both large-scale and small-scale features of [...] data"* [276], and forming and verifying hypotheses [276]. Complementary to that, problems with the data can (unintentionally) become visible.

To summarize, visualization is often seen as the process of representing subject-matter content (or data) – such as models, objects or relationships – using graphical primitives. The goal of this process is generally to provide users with (new) insight into and understanding of the input data using visual means as output. *"Graphics reveal data. Indeed graphics can be more precise and revealing than conventional statistical computations"* [266]: Tufte's summary for one of the main benefits of graphical representations of data.

The former definitions, however, do not yet define *how* the visual representation is created. Such visualization could still be hand-drawn or even have a physical appearance such as a scale model. In the context of computer science, Card et al. give a more narrow definition in the sense that it specifies computers as the means to create such visual representations:

Definition 5 (Visualization (Card et al., 1999))**.** *"Visualization is the use of computer-supported, interactive, visual representations of data to amplify cognition"* [47].

Although computers internally represent the subject-matter content in intangible ways, its representation may again have a physical appearance. Moreover, this definition is also wider than the previous ones, since by including ‘interactive’ Card et al. allow for a feedback loop between users and the creator of the visualization, i.e., the computer. So, with the help of computers, visualization was able to become interactive, which enabled it to support a whole new set of tasks.

Further, it is important to distinguish between a visualization’s task (visually representing subject-matter content) and a user’s tasks if one wants to classify a visualization (Butler et al. [45]) or evaluate its effectiveness and efficiency (Kosara et al. [149]). That is, it is crucial for a visualization not only to fulfill the visualization task itself but also to do this in a way that supports users in performing their tasks. For this, Butler et al. distinguish three high-level categories of user tasks:

- Descriptive visualization: *“Is used when the phenomena* [...] *is known* [sic]*, but the user needs to present a clear visual verification of this phenomena* [sic] *(usually to others)”* [45].
- Analytical visualization (directed search): *“Is the process we follow when we know what we are looking for in the data; visualization helps to determine whether it is there”* [45].
- Exploratory visualization (undirected search): *“Is necessary when we do not know what we are looking for; visualization may help us understand the nature of the data by demonstrating patterns in that data”* [45].

A common phenomenon in visualization that can hinder users in performing their task or reduce their efficiency is *visual clutter.* Clutter often results from too many items being represented at once. The American Heritage College defines clutter as *“a confused or disordered state, caused by filling or covering with objects”* [11]. Apart from the number of items, their confusing organization can already cause visual clutter. Rosenholtz et al. therefore extend this definition as follows:

Definition 6 (Rosenholtz et al. (2005))**.** *“Clutter is the state in which an excess of items, or their representation or organization, lead to a degradation of performance at some task”* [227].

Tufte [266] further emphasizes that visual clutter can not only be caused by either too many items or their organization, it can also be more generally understood as anything that causes users to get confused by a visualization. For example, a confusing visual mapping can hereby be an instance of visual clutter.

2.4.1 Information Visualization versus Scientific Visualization

Being a sub-discipline of visualization, information visualization is about representing *abstract* data, i.e., data that has no inherent appearance or geometry. This includes high-dimensional data such as financial data or collections of documents [47]:

Definition 7 (Visualization (Card et al., 1999))**.** *“Visualization is the use of computer-supported, interactive, visual representations of* abstract *data to amplify cognition”* [47]*.*

Abstract data, or any similar subject-matter content, contrasts with physical data (and thereby scientific visualization) in that it *“does not have any obvious spatial mapping”* [47]. So, information visualization is first concerned with actually mapping data to *suitable and appropriate* visual forms which can then be used to represent properties of the data. However, as Diehl and Spence note [78, 246], the distinction of information visualization to scientific visualization is not always perfectly clear. Despite efforts to define more clear distinction criteria [54, 55], these two disciplines always share some commonalities.

Haroz and Whitney stress *"that the severe capacity limits of attention strongly modulate the effectiveness of information visualizations, particularly the ability to detect unexpected information"* [114]. Several examples illustrate well that small differences, for instance, in a visualization's appearance or layout can make a big difference in its effectiveness. Since information visualization is typically concerned with complex or massive amounts of data, Chen points out that from the beginnings of information visualization one of its aims has been *"to make the insights stand out from otherwise chaotic and noisy data"* [54], i.e., to *"reveal*[ing] *structures that would otherwise be invisible"* [54]. The second goal that he mentions *"is about growth, evolution, and development. It is about sudden changes as well as gradual changes"* [54]. Here, we see that 'data' can also refer to a number of individual data sets that have to be combined, correlated and compared to eventually yield meaningful insight such as the differences between two data sets.

2.4.2 Software Visualization

Software visualization is a part of information visualization and, as such, also concerned with abstract data: Data from the software-engineering domain. Hence, software visualization is faced with subject-matter content that *"is inherently intangible and invisible"* [78]. According to Diehl, the goal of software visualization *"is to help to comprehend software systems and to improve the productivity of the software development process"* [78].

In a rather wide definition of software visualization, Zhang includes many sources of subject-matter content:

Definition 8 (Software Visualization (Zhang, 2003))**.** Software visualization is the *"use of various visual means in addition to text in software development. The forms of the development means* (sic) *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"* [289]*.*

Price et al. similarly give the following definition:

Definition 9 (Software Visualization (Price et al., 1993))**.** Software visualization is *"the use of the crafts of typography, graphic design, animation and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software"* [209]*.*

Interestingly, this definition also includes facilitating the *usage* of computer software, hence it does also involve *end users* of software, not only the software engineers that build software. However, in a later version Price et al. give a quite narrow but pragmatic definition:

Definition 10 (Software Visualization (Price et al., 1998))**.** Software visualization is *"the visualization of computer programs and algorithms"* [210]*.*

Diehl gives a similarly compact definition as follows:

Definition 11 (Software Visualization (Diehl, 2007))**.** Software visualization is the *"visualization of artifacts related to software and its development process"* [78]*.*

He adds more general descriptions such as that *"software visualization is the art and science of generating visual representations of various aspects of software and its development process"* [78].

Although Diehl highlights that *"the goal of software visualization is not to produce neat computer images, but computer images which evoke mental images for comprehending software better"* [78], this is not represented in his and the later definition of software visualization by Price et al. (from 1998). That is, these definitions do not take into account that the purpose of images in (software) visualization is not their mere existence, but to facilitate a specific task.

Mili and Steiner further stress that it is insufficient to *"restrict the term software to mean computer programs"* [183] and that a visualization can be done in many ways: *"we do not impose any restrictions on the term visualization: it can be textual or graphical provided it meets the criteria established in the definition"* [183]. Hence, they define the term from a software-engineering perspective as follows:

Definition 12 (Software Visualization (Mili and Steiner, 2001))**.** *"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 a software engineer that organizes his or her concept of the operation of a computer system"* [183].

No matter which definition is used, researchers and practitioners agree that software visualization can be *"a useful and powerful technique for helping programmers understand large and complex programs"* [254]. Kranzmüller states that *"in practice, many research areas have already accepted (and demonstrated) that graphical means are the only adequate solution to express the complexity of software"* [152]. Likewise, Mili and Steiner highlight 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"* [183]. Zhang adds 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"* [289].

The definitions cited above, though partially different, feature a significant overlap. They are each defined in individual contexts, so they focus on different aspects of software visualization. To properly relate them to each other, the respective context thus needs to be considered too.

To account for the context of this thesis, the following definition of *software visualization* will be used:

Definition 13 (Software Visualization)**.** *Software visualization is the art and science of visualizing various aspects of software, related artifacts, and its development process to facilitate both engineering and management tasks in software-engineering projects.*

Chapter 3

Basics and Related Work

There are various ways to classify techniques in software visualization [224]. For example, one can differentiate techniques by displayed subject-matter content (e.g., code or data) [192, 193, 209], display type (e.g, static image or interactive graphics) [175, 192, 193, 209, 224]. Among other criteria, Diehl [78], Stasko and Patterson [250], as well as Bohnet [32] use the *aspects to be understood*, such as structure, to classify techniques. While this provides a higher-level segmentation of the field, it is also rather broad. Therefore, this thesis combines the former organization (Section 3.1) with an application-specific view: Basic techniques are presented first and, subsequently, selected tools are each grouped by *tasks* relevant in this thesis (Sections 3.2 to 3.6). This is inspired by the *task-oriented model* by Maletic et al. [175]. For instance, one task in behavioral analysis is to understand 'basic' concurrency (e.g., parallel function calls) and another task is to understand synchronization of threads. In the remainder of this thesis, the model by Maletic et al. will also be used to characterize the presented techniques.

Related work in the field of behavioral analysis of software is concerned with visualizing behavior (Section 3.2), correlating it with structural entities (Section 3.3), and correlating it with other behavior (Section 3.4). In the field of structural analysis of software, related work separates into visualizing correlations between attributes of multivariate structures (Section 3.5) as well as techniques for virtually modifying such structures (Section 3.6).

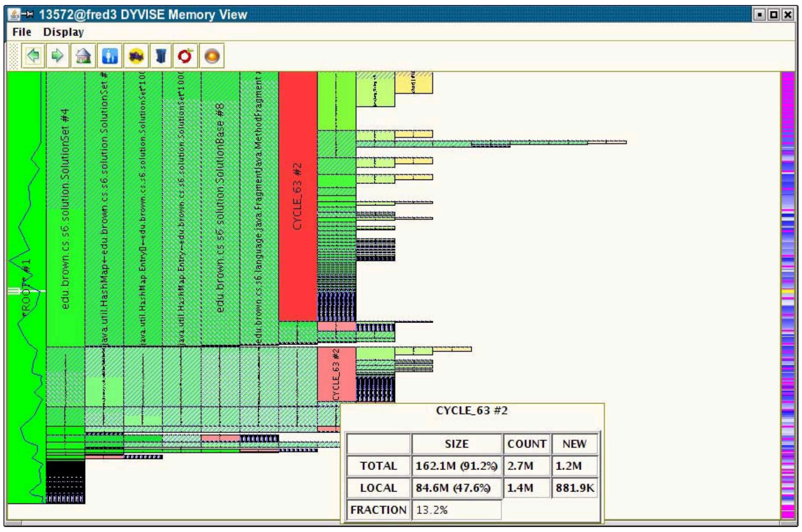
3.1 Basic Visualization Techniques for Structure and Activity

This section reviews techniques for visualizing structure and activity. *Activity* here refers to time-dependent signals and, as such, for example includes both data describing software behavior as well as software evolution. While the shown examples mainly originate from the software visualization domain, the techniques are not limited to this domain and generally serve as techniques in information visualization.

There is often no clear distinction between activity views and structure views in terms of the 'aspect to be understood'. Many 'structure views' can also be used to show activity and vice versa: For example, (nested) node-link diagrams can show software structure and call relationships [34] as well as call sequences using *animation* [268]. According to Stasko and Patterson, animation *"consists of the rapid sequential display of pictures or images, with the pictures changing gradually over time. These pictures are the frames of the animation"* [250].

3.1.1 Activity Views

Activity is commonly depicted using 2-dimensional (2D) timelines in which one axis explicitly encodes time [115]. Icicle plot [154] variants often visualize hierarchical event sequences, which then basically work as 2D timelines (e.g., Fig. 3.1). One axis is mapped


13572@fred3 DYVISE Memory View
File
Display
ROOT #1
edu.brown.cs.s6.solution.SolutionSet #4
CYCLE_63 #2

CYCLE_63 #2

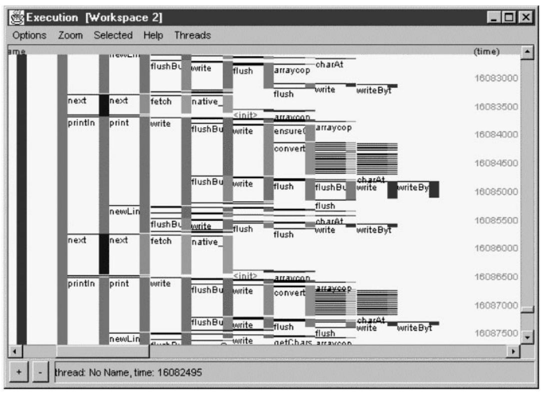
	SIZE	COUNT	NEW
TOTAL	162.1M (91.2%)	2.7M	1.2M
LOCAL	84.6M (47.6%)	1.4M	881.9K
FRACTION	13.2%		

Figure 3.1: *Example icicle plot in DYMEM [216].*

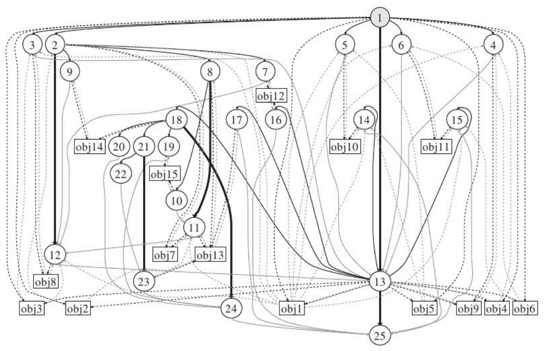
to time and the other axis is mapped to 'hierarchical composition' such as call stacks [76], memory block ranges [188], or object ownerships [216]. Stacked timelines can show multiple (correlated) time-dependent signals [221, 269], such as the evolution of source-code files.

In a similar way, node-link diagrams [19, 228] can, for instance, represent cause-effect relationships or data flow. Examples include UML sequence diagrams and UML timing diagrams [13, 75, 76, 93, 181, 284] (e.g., Fig. 3.2(a)) or extensions to the Business Process Modeling Notation [186]. These are, however, less efficient than icicle plots in terms of space usage. Such techniques have difficulties showing large graphs with many nodes and edges. Edge crossings become a major challenge there (compare Fig. 3.2(b)). Section 3.1.4 provides more details on this topic.

Seesoft [84] originally is a technique for depicting the evolution of source code. It shows miniaturized views on source code, which are then augmented with change information. Later, it was also applied successfully for visualizing runtime data such as fault-likelihood of statements [133] and profiling data [134]. It was further extended to 3-dimensional (3D) space for additional flexibility [176].



(a)



(b)

Figure 3.2: *(a) Example sequence diagram in Jinsight [76] and (b) example node-link diagram in ThreadScope [279].*

Animation is an orthogonal technique that enables other visualization techniques to show time-dependent signals and cause-effect relationships [136]. It typically generates transitions between 'key frames' that other visualization provide. For example, Voigt et al. visualize FBTs using node-link diagrams and animation [268], and after every frame highlight newly added nodes in the diagrams. Likewise, Sigovan et al. [239] depict communication types using scatter plots augmented with animation; Choudhury uses animation to show memory reference behavior of software systems [58].

For higher-level views on activity, techniques often aggregate the time-dependent data. For example, profilers, such as gprof [108], perform stack sampling to determine the accumulated duration of all calls to a function. Such aggregated information can be presented as bar charts [113, 117, 287], 1-dimensional timelines [195, 199, 287], node-link diagrams [199, 279, 292] as well as treemap variants [274, 285] (see Section 3.1.3 for more details). Treemaps and scatter plots, as multivariate visualization, help correlate items in high-dimensional data sets, such as peer-to-peer download metrics [270], software traces [113, 117, 184, 221, 235], and profiling data [163].

3.1.2 Activity Overviews

To aid navigation in a detail view, *overviews* provide miniaturized, zoomed-out representations of the depicted data. This idea seems to originate in the 1980s' video games. Literature is not clear in this respect, but for example, Cockburn et al. [60] mention Defender as an early example.

The visual representation of subject data in overviews typically bears a strong resemblance with the associated detail view [130, 131, 181, 243]. This resemblance essentially leverages a strength of the human visual system: Recognizing structural similarity and (recurring) patterns [276] between both views. The patterns thus act as 'landmarks' in both views during navigation. For it, overviews need to scale the detailed representation in a way that it is still readable in an overview.

Moreover, overviews can also present data in a different way [68, 81], essentially fulfilling two tasks then: Aiding navigation and emphasizing certain properties of the data. For example, the Massive Sequence View [68] (MSV) shows entire hierarchical sequences such as call sequences. It plots a hierarchy (e.g., namespaces, classes, functions) as icicle plots on the horizontal axis and time on the vertical axis. It shows events (e.g., function calls) occurring at certain points in time by connecting affected nodes (e.g., functions) with horizontal lines. This visually groups events by involved structural entities through horizontal alignment and simultaneously highlights outliers involving different entities.

Generally, for constructing miniaturized representations, several techniques exist. A straight-forward way is to compute a representation in the same way as the respective detail view and then scale it to fit the overview's screen area. More involved techniques include the use of anti-aliasing techniques to better cope with the frequent sub-sampling problem: Aliasing occurs if the visualization items to be scaled to the screen area are smaller than single pixels. This can be addressed by alpha-blending all items that would end up on the same screen pixel, effectively aggregating the original information. For example, the Information Mural [131] (Fig. 3.3) and the MSV apply this technique.

3.1.3 Structure Views

Structure views are commonly used to show hierarchical data such as organizational layers in companies, catalogs, directory structures, and containment of software artifacts. Treemaps (cf. Fig. 3.4), node-link diagrams, and icicle plots are used not only to show

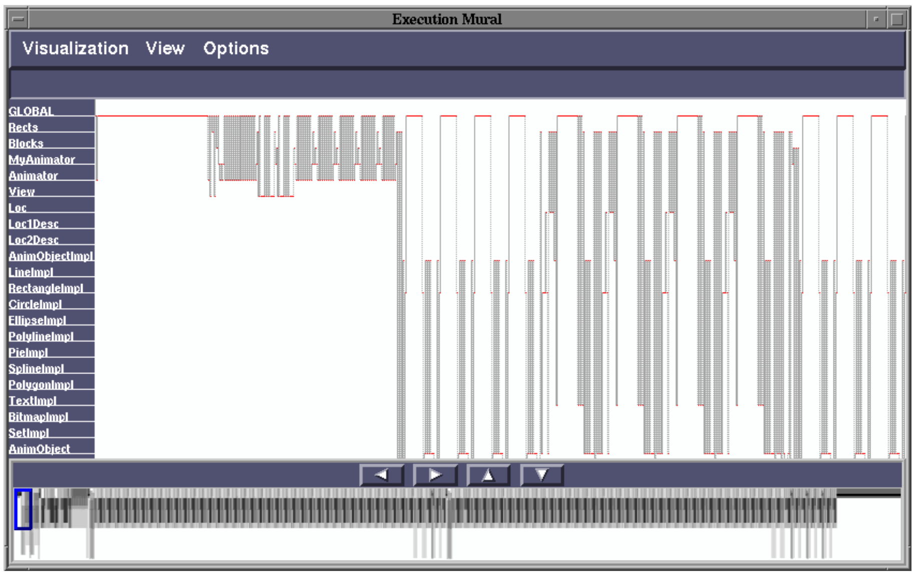


Figure 3.3: *The Information Mural as activity overview (bottom) [131].*

activity (see Section 3.1.1) but as well to show hierarchies H (cf. Section 2.1.5). They also support visually comparing substructures and correlating the distribution of metrics (mapped to visual attributes) with and within the visualized structure. The circular SunBurst technique [248] represents hierarchy levels as concentric rings surrounding an inner circle that represents the root node.

For node-link diagrams, various graph layouts [14, 15, 19, 165] focus on different aspects of the layout, such as minimized number of crossings, constrained angles in edges etc., to highlight certain aspects of the input structure. For hierarchies H, rooted-tree layouts and hyperbolic trees [155] are free of edge crossings, but in this 'bare' form do not include relationships R. Similarly, several layout algorithms for treemaps optimize the layouts with respect to certain criteria. One increasingly important criteria is the *layout stability* that refers to spatially consistent layouts for similar hierarchies. It can thus be defined as *"a layout algorithm's 'tolerance' against changes in varying input hierarchy-data with respect to the arrangement and layout of resulting visual representations"* [295]. For example, the Strip [22] layout optimizes towards square aspect ratio of the rectangular

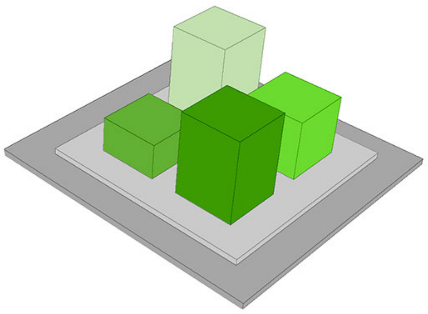


Figure 3.4: *Example (3D) treemap [298]. A hierarchy is represented as recursive subdivision of a rectangular area. Here, the gray rectangular area represents the hierarchy's root, level 0. While the light gray box represents the root's only child (hierarchy level 1), whereas the inner greenish boxes represent children of this node.*

treemap items and layout stability; the layout by Hilbert and Moore [258] improves on this layout stability.

Traditional treemaps use rectangles for representing entities in the input hierarchy. In contrast, 3D treemaps [29] use boxes, radial treemaps use circles [275], and Voronoi [17] treemaps use arbitrary polygons. Voronoi treemaps provide more flexibility with respect to item placement and, with deterministic distribution of items, also further improve layout stability [295]. While treemaps are generally space-filling visualization, software cities [251] intentionally work in a non-space-filling manner to improve layout stability compared to traditional treemaps. The layout is inspired by street networks in real cities and by how these cities expand: Expanding the ground area through adding streets and buildings in the surroundings. The layout algorithm for software cities thus accounts for future changes (additions, deletions, moves) in the hierarchy by incorporating a significant amount of unused space in the initial layout.

Node-link diagrams with multiple edge types allow for visualizing structures $\mathbb{S}$, i.e., a hierarchy H and relationships R [119]. Extensions for treemaps and other hierarchy visualization typically overlay the hierarchy H with relations R in the same screen space [50, 123], which creates hard-to-read images for larger structures due to a lot of overdraw and crossings (visual clutter). Edge bundling (cf. Section 3.1.4) can be leveraged to alleviate this problem, but some overdraw still exists. This is especially difficult to handle if the visualization shall also show labels for hierarchy items [297], which for readability reasons, should be geometrically close to the labeled items. CBVs [123] (cf. Section 6.1) address the hierarchy-relationships overdraw problem by using different geometric primitives and by geometrically separating H and R. A circular icicle plot represents H (similar to the SunBurst technique) and edge bundles depict R in the free area inside the icicle plot.

3.1.4 Representation of Relationships: Edge Bundling

Generally, the explicit visualization of relationships tends to be prone to edge crossing (compare Fig. 3.2(b)), which can significantly limit the readability – and thereby scalability – of a visualization [43, 85, 211]. Concerning this problem, Graham and Kennedy state that unbundled edges *"allow*[s] *exact and individual relationships to be traced between different trees* [, ...a] *drawback with this approach is if there are many lines displayed at once, then individual edges become impossible to distinguish from the mass of drawn lines, a common problem in the perception of graph drawings"* [106]. They add that there is *"a collection of methods* [such as edge bundling] *to alleviate traceability difficulties"* [106].

Conversely, McGee and Dingliana [179] found in experiments that, while bundling may reduce the users' accuracy at tracing individual edges, it improves their response time (at equal accuracy) with respect to the identification of higher-level connectivity of node clusters. According to their experiments, edge bundling appears to be suited for reading high-level connectivity information from large graphs and straight lines are more suited for reading individual edges. These results, though, relate to static images and thus possibly do not apply to interactive visualization tools which can highlight individual (bundled) edges upon user interaction.

3.2 Behavior Visualization of Multi-Threaded Software Systems

Multi-machine (distributed) and single-machine (shared-memory) systems share many concepts, but there are also major differences. For instance, distributed systems are commonly built such that individual nodes can operate autonomously, while many multi-threaded systems feature a coordinating thread that creates/destroys other threads and controls these threads' operation to a large extent: *"Many of these* [distributed] *applications support a high degree of concurrency, serving thousands or even millions of concurrent user requests. They support rich and frequent interactions with other systems, with no intervening human think time. Many server applications manipulate large data sets, requiring substantial network and disk infrastructure to support bandwidth requirements"* [9]. So, the means used for communication and synchronization is typically different: Distributed systems communicate via networks; shared-memory systems share memory ranges between threads, hence their name. As a result, visualization of shared-memory systems typically does not include network communication, but visualization for multi-machine systems does [73, 117, 121, 189, 195, 199, 287]. Also, for multi-machine systems, runtime behavior is often analyzed on a coarser scale; the runtime behavior of single processes or threads is shown only on higher levels of abstraction or omitted. Visualization techniques thus often focus on either of the two systems; a few hybrid variants capture both aspects. As this thesis focuses on shared-memory systems, only tools addressing their visualization are discussed in the following.

Selected Tools

Jinsight [76] uses several linked views to accomplish performance analysis and debugging tasks: Icicle plots, sequence diagrams (Fig. 3.2(a)), and call graphs (i.e., node-link diagrams). Code Bubbles [215] visualizes correspondences between threads and lock usages by stacking and alignment. Additionally, class-to-lock relationships are shown by node-link diagrams. Users can analyze which threads use which locks and how locks are inter-connected through their use in the same code locations. In terms of features and used visualization techniques, the former two techniques are closely related to TraceVis.

Program Dependence Graphs (PDG) [90] are variants of node-link diagrams and were first used for compiler optimizations. Extensions enable their use in analyzing concurrent software behavior [292]: Multi-threaded Dependence Graphs (MDG). ThreadScope [279] (Fig. 3.2(b)) and Atropos [174] similarly use dependence graphs for the analysis of data flow and control flow to enable locating synchronization bottlenecks. The OCoN modeling language [103, 282] likewise supports representing concurrency aspects and resource co-ordination in parallel and distributed systems; the lock-causality graph [142] which aims at uncovering potential deadlocks in multi-threaded systems includes specific symbols for lock/unlock and wait operations as well as causality edges. Similarly, SynchroVis [274] shows synchronization relationships between threads using a 3D-treemap metaphor extended by edges. Bottle Graph [80] is a stacked bar chart that shows each threads' (or groups of threads') running time and degree of parallelism for performance optimization. Similarly, WAIT [9] uses bar and pie charts to show various key performance indicators of multi-threaded systems.

Concurrency Visualizer [101] (Fig. 3.5), a tool integrated in Microsoft Visual Studio 2010, depicts thread states (running, waiting, synchronizing, etc.) along the recorded time span as stacked timelines. Exploration of a thread's execution context is restricted to selecting a thread's sample and then manually expanding its recorded stack in a list view. Using stacked bar charts, Jedule [127] shows scheduling decisions in MPI applications, but could also be used to show scheduling decisions on multi-core machines. The Eden Trace Viewer [27], Pajé [72], VPPB [37], and the Time Traveling Debugger [281] also use stacked timelines to show concurrent behavior on multiple scales such as machines, processes, and threads; colored sections in the timelines encode activity types. Messages between these entities are optionally shown as straight-line overlays. TraceVis [221] depicts concurrent executions on microprocessors by stacked timelines that each represent the execution of

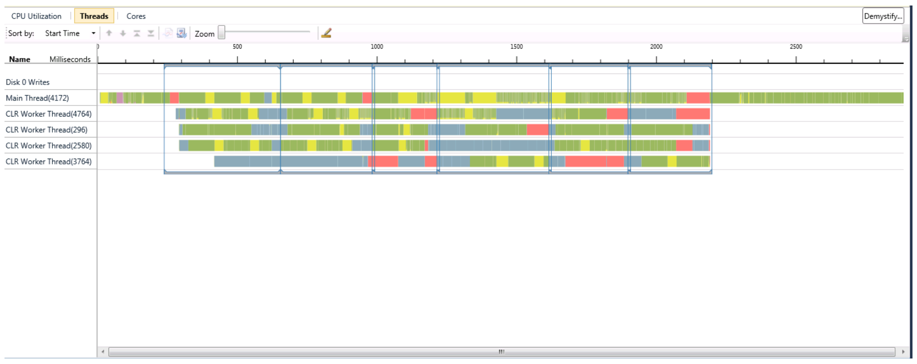


Figure 3.5: *Concurrency Visualizer in Microsoft Visual Studio 2010 [101].*

pipeline stages (fetch, decode, execute) by colored bars. In addition, TraceVis enables correlating the execution to higher-level static representations of the execution: Source code and assembly. Together 4 [181] (part of Javis [180]) presents multi-threaded software behavior using variants of UML sequence diagrams and an overview showing a miniaturized version of the diagram.

In addition, several tools for educational visualization target novices. For instance, they demonstrate concurrency principles by algorithm animation. Probably one of the first tools for animating concurrent software systems is the one by Zimmerman et al. [293]. Further examples are PARADE [249], TANGO [40] and a tool by Bedy et al. [23]. Kerren and Stasko give an elaborate overview of techniques and tools for algorithm animation [140].

Orthogonal to the examples in reverse engineering above, Visputer [290] is a tool for forward engineering parallel systems.

Differences to TraceVis and SyncTrace

TraceVis works as a fundamental concept for visualizing function-boundary traces captured either from single-threaded or multi-threaded systems. It was applied successfully in single-threaded contexts in Bohnet's thesis [32] by using a single activity view and overview. This thesis extends its application to multi-threaded contexts, particularly by providing methods for view compaction and synchronized exploration of such compacted views. This enables the *implicit* visualization of concurrent function calls while maintaining a compact representation of the massive subject data. While many existing techniques address performance optimization and understanding of multi-threaded behavior at the thread level, TraceVis aims at understanding runtime behavior at function-call level and at analyzing respective cause-effect relationships. Further novel extensions are interaction by adaptive brushing and approaches for addressing visual scalability issues: Rendering techniques for space-efficient representation of icicle-plot structure and manual as well as automatic aggregation strategies for icicle plots.

The view compaction essentially is a focus+context approach [98] using multiple foci. It compacts the underlying trace data per thread trace to harmonize highly variable event durations in the raw trace data. Similar approaches include TimeSeer [71], SignalLens [143], and Stack Zooming [130]. While the former two integrate focus and context representation within a single view, the latter is a method for stacking multiple views, each showing a focus of interest. In contrast to TraceVis, they scale all shown time-dependent signals equally, which makes it hard to emphasize features at different points in time in individual signals. Nonlinear mathematics are further used in a wide range of research domains. Examples include human behavior [95, 166] and self-adaptive systems [24, 122].

SyncTrace, which extends TraceVis, allows for visualizing multiple threads' activity in parallel while *explicitly* representing correspondences between those threads. It uses a combination of straight and bended icicle plots, which improves on visual scalability: First, the circular layout represents the threads' activity in a space-filling manner and thus theoretically allows for simultaneously showing an unlimited number of threads (or other hierarchical sequences) in a fixed-size area. In practice, the number of sequences that can be represented in a readable manner is still limited by available screen space. Second, the correspondence shapes allow for encoding the temporal extent of the wait durations connected to such relationships. Third, the circular layout of the sequence visualization allows for showing correspondences between any threads in the context view in a uniform manner.

3.3 Visualization of Correlated Behavior-Structure

This section outlines related work on visually correlating structure and behavior. While there may be no clear distinction between structure views and activity views, for correlating both data one typically uses two spatially separate views to depict these information spaces: One for structure, one for activity data. The two views are then commonly connected using the *linked-views* technique [48, 222]. Coordinating multiple views often *"makes the collection of views far more powerful than the sum of their individual strengths"* [76] and helps contrasting different conceptual views of the same data [222].

Selected Tools

Extravis [68] links a CBV and an MSV using selection and brushing to show which subsystems are active in certain execution phases. The MSV shows the activity (calls) over time and allows for selecting ranges for which aggregated call information is then shown as edges in the CBV. The outer rings (showing system structure) of the CBV enable correlating those calls with the system structure. Extravis is most similar to ViewFusion in terms of the supported task. TimeLineTrees [43] combine a rooted-tree visualization with timelines to show the contribution of leaf nodes or entire subtrees to an activity in a static image. Correlations between behavior and structure are thereby encoded implicitly by alignment. BusyBorg [89] uses augmented UML diagrams for live visualization: Overlayed edges show behavior in the context of the structure diagram.

Gammatella [134] correlates execution data on several levels: Statement-level (colored source-code representation), file-level (Seesoft-like miniaturized, colored source-code representation) and system-level (colored treemap). The views are linked similarly as in Extravis using context menus and highlighting. Tarantula [133] helps correlating runtime data in the form of outcomes of test cases (passed/failed) to localize faults in software systems. This is also done by means of the Seesoft technique and a bar chart showing test cases.

Differences to ViewFusion

Although easy to learn and use, views linked only by selection and brushing require a certain effort to use, in particular when the *spatial layout* of the views is different. More precisely, such split-attention setups are known to generate a significant amount of *context switches* that require mental effort, time, and short-term memory to assimilate these distinct views [60]. Especially mentally challenging tasks, such as program comprehension, though, also require users to concentrate and use their short-term memory to correlate pieces of information. Moreover, displaying such views side-by-side, such as when views share an axis, requires a non-negligible extra amount of screen space. In effect, separate visualization is not optimal for these tasks.

In contrast, the ViewFusion technique integrates an activity view and a structure view within a *single* view by 'fusing' the two information spaces using spatial overlay. In this sense, BusyBorg is similar to ViewFusion, but BusyBorg's edge overlays quickly render the underlying structure unreadable due to occlusion.

By the overlay in ViewFusion, the geometric distance between the two views is reduced. In effect, the results of user input become visible in this single view instead of being distributed across multiple separate views. Consequently, data in the two information spaces can be visually correlated with less disruption. This concept is exemplarily implemented in a prototype tool for correlating software behavior and software structure.

3.4 Visualization of Correlated Behavior-Behavior

Several techniques correlate single sequences [3, 20, 65]. The work presented in this thesis is concerned with correlating two sequences, which *"is significantly harder"* [110]. For it, various techniques exist that visually compare hierarchical, ordered sequences (e.g., thread traces, cf. Section 2.1.4): Graham and Kennedy present a survey of tree visualization and comparison techniques [106]. Such comparison is typically done by combining structure views or activity views (cf. Section 3.1.1, Section 3.1.3) with techniques for showing correspondences. These include scatter plots [3, 65]; parallel coordinate plots [20, 79]; color coding and interactive highlighting [190]; stacking and alignment [215, 269]; and edges – unbundled [291] as well as bundled [124, 260] – to explicitly represent correspondences.

Other than that, one can also compute a *union tree* (contains all nodes of all input trees) or a *consensus tree* (contains only nodes shared by all input trees) and then highlight structural uncertainty [99], i.e., distinct subtrees of the input trees. By this, shared subtrees are present only once in the visualization. Consensus and union trees thus works best for closely similar input trees.

Selected Tools

Guerra-Gómez et al. [110] discern five types of tree comparison problems as shown in Fig. 3.6. The comparison problems addressed in this thesis are of type 4 (TraceDiff), concerning the pair-wise comparison; and type 2 (SyncTrace), concerning the representation of pair-wise correspondences between multiple trees. The following tool examples are as well organized by these five comparison types.

Selected Tools: Topology Comparison (Type 0) To address type 0 problems, several techniques stack the trees for comparison and represent correspondences explicitly using edges. The work by Holten and van Wijk [124] exhibits a visual resemblance to TraceDiff. They also use juxtaposed icicle plots to show two trees (Fig. 3.7(b)) and show correspondences using Hierarchical Edge Bundles [123] (HEB). Color coding further highlights commonalities and differences in the two hierarchies. Similarly, Code Flows [260] (Fig. 3.8(b)) depicts multiple trees side-by-side as icicle plots and correspondences as 2-dimensional curves. Dynamic Indented Plots [44] also aligns indented hierarchies side-by-side, but in contrast to Code Flows, represents only changes using edges. Zhao et al. [291] compare object interactions across multiple test cases by stacking 2-dimensional sequence diagrams in 3D space (Fig. 3.8(a)); straight lines highlight similar nodes.

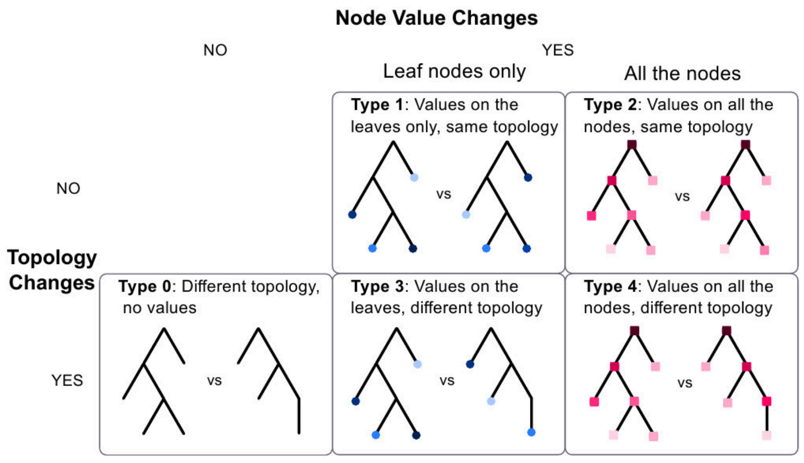

Node Value Changes
NO
YES
Leaf nodes only
All the nodes
Topology Changes
NO
YES
Type 0: Different topology, no values
Type 1: Values on the leaves only, same topology
Type 2: Values on all the nodes, same topology
Type 3: Values on the leaves, different topology
Type 4: Values on all the nodes, different topology
vs

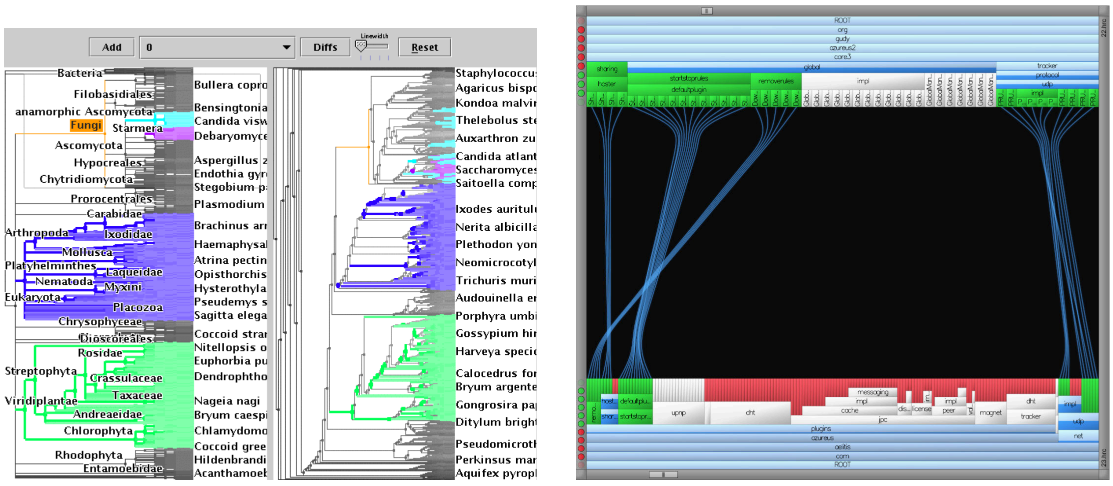
Figure 3.6: *Types of tree-comparison problems [110].*

Other tools, such as TreeJuxtaposer [190] (Fig. 3.7(a)), show correspondences by color linking [105] as well as by consensus trees and structural uncertainty [36, 99, 107, 161] (e.g, MultiTree [99]). MoireTree [185], a radial visualization, compares multiple trees that define orthogonal hierarchies on a fixed set of leaf nodes. TimeRadarTrees [43] can show the evolution of undirected relationships R in structures $\mathbb{S} = (R, H)$ (cf. Section 2.1.5) and edge weights using segmented concentric rings. TimeArcTrees [43] represents multiple versions of directed relationships R using several aligned node-link diagrams for the relationships and a separate rooted-tree representation of the hierarchy H. TimeArcTrees assume that H is either stable or a union/consensus tree.

To cope with large number of trees to compare, TreeSet [10] computes a 2-dimensional structural similarity per pair of trees. By drawing the result as a scatter plot in which the geometric distance represents similarity, users can subsequently select strongly matched trees for in-detail comparison based on their spatial proximity.

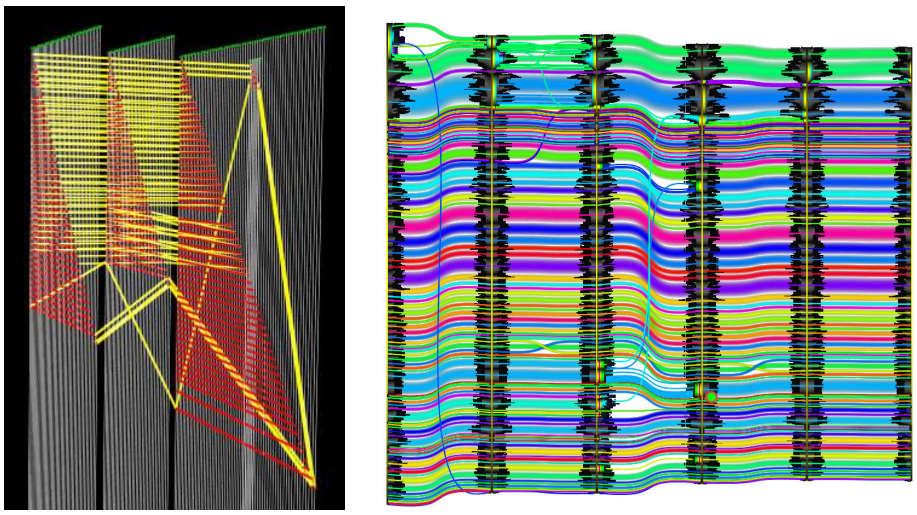
Selected Tools: Value Comparison and Topology Comparison (Type 1 to Type 3) Animated treemaps [102] provide transitions between treemap layouts and optimize layout stability. By this, changes to node sizes in consecutive layouts can be tracked better. Contrast treemaps [265] first compute a consensus tree of two input trees and then show the values of shared nodes using colors. To show the colors of two trees, each rectangular item is split into two distinctly colored triangles. As both techniques use treemaps, they are inherently restricted to showing aggregated value changes on non-leaf nodes (type 1). Parallel Coordinate Trees [38] can display multivariate node values on any hierarchy level using colored lines. Although it was not designed to show changes, it could be used to compare two versions of a value by differentiating the versions by line color (type 2).

Selected Tools: Value Comparison and Topology Comparison (Type 4) TreeVersity2 [110] builds upon the idea of the Contrast treemap and also visualizes a consensus tree, however, using icicle plots. Added and removed nodes are highlighted and both relative as well as absolute value changes are encoded by a vertical sub-partitioning of the plot's nodes and color, respectively. TreeVersity2 is most relevant to this work in terms of supported comparison types.



(a) (b)

Figure 3.7: *(a) TreeJuxtaposer [190] and (b) tree comparison by Holten and van Wijk [124].*



(a) (b)

Figure 3.8: *(a) Comparison of sequence diagrams in 3D space [291] and (b) Code Flows [260].*

Differences to TraceDiff

TraceDiff shows matches between two trees (here: thread traces) in a multiscale manner. Previous work often only addresses type 0 problems (topology changes). In contrast, TraceDiff supports type 4 problems by visualizing the time-extents (node values) of execution matches (similarities) and by explaining them using a multiscale nesting of the tube bundles (topology). In particular, it is different from previous techniques in that it can show *complex* topological changes, such as joins, split, and swaps, on *any* hierarchy level (by nested bundles) and value changes (by tube bundles) in parallel. Moreover, it addresses visual scalability and readability through the introduction of a modified layout for hierarchical edge bundles and their extension to shaded tube bundles.

Differences to SyncTrace

Though SyncTrace is not used to compare trees (traces) in this thesis, it does show correspondences on trees. As such, it is related to tree comparison techniques. The 'comparison' problem is not exactly matched by the classification of Guerra-Gómez et al.: The tree topologies are different (i.e., types 0, 3, and 4), but the differences are in fact only relevant for representing correspondences and their values (types 1 and 2). Moreover, there is no value per node but only a single value per correlated pair of nodes. As the correspondences concern nodes values on any hierarchy level, the best fit is a type 2 problem.

In contrast to many tree-comparison techniques, it arranges the individual tree representations in a space-filling manner. This enables it to show a large number of trees in parallel. Second, the 2D correspondence shapes partially support value comparisons: They can encode not only the relationship itself, but also the value(s) of *originating* node(s). Third, the circular layout of the trees allows for showing correspondences between any visualized trees in a uniform manner.

3.5 Visualization of Software Structure and Related Artifacts

For depicting multivariate structures of software systems, structure views are commonly combined with the linked-views technique. For it, *"the classic node-link diagram is inappropriate for tree representation* [of complex software structures]*, quickly becoming too large and making poor use of the available display space"* [49]; thus most tools use different structure views. Such structure views are further extended to show multiple attributes per node [29, 33]. Caserta and Zendra provide a survey of visualization techniques used for analyzing software structures [49].

Selected Tools

Most similar to the techniques presented in this thesis, the industrial Solid* tool suite (SolidSX, SolidFX, SolidSDD) of SolidSourceIT [220] uses CBVs and cushion treemaps to depict system structures: Hierarchy and relationships. EvoSpaces [5] (Fig. 3.9) uses a 2.5D grid or spiral layout to show a software structure. 3D-HEB [50] improves the edge routing of the former by applying the HEB to 3D space. GrammaTech CodeSonar[1] is another industrial tool that uses UML diagrams and reduces visual clutter by edge bundling.

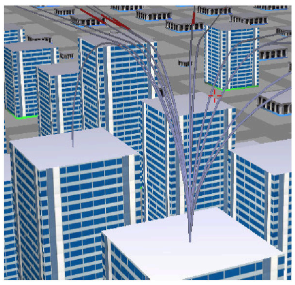


Figure 3.9: *Evospaces showing relationships between entities [5].*

Further, SeeSys [16] depicts the evolution of software metrics per software artifact in a treemap-like representation. Histograms are embedded in each treemap item's area to show its value evolution with respect to a certain metric. Similarly, Balzer and Deussen represent software structures as several visually linked treemaps, each treemap representing a subtree and edges representing relationships between subtrees. Software maps [33] and the similar software cities [156, 278], which basically are variants of 3D-treemaps, as well as Voronoi treemaps [17] depict snapshots of a system structure and can represent node attributes by visual variables size and color as well as height (in a 3D visualization). ExplorViz [92], although used for visualizing traces, similarly shows a hierarchy and relationships by adding edges to 3D-treemaps. EvoStreets [251] use a

[1]`http://www.grammatech.com/codesonar`, last accessed 11/27/2013

street-like layout on a 2.5D plane to show a hierarchy; node attributes are represented as color and size of the node items. Similarly to ExplorViz, relationships can be encoded in EvoStreets through edge overlays [148, 251] and bundling. The SArF tool [148] further focuses on features and architectural layers by a) feature-based clustering of items and b) using one dimension of the 3D space to represent layers.

MetricView [262] augments UML diagrams with symbols to add metric information. Polymetric views [157] are node-link diagrams with metrics mapped to visual variables of nodes to visualize object-oriented software systems; the nodes correspond to classes and the links typically represent inheritance relationships. CppDepend[2] is an industrial tool that uses node-link diagrams, treemaps and adjacency matrices (similar to IMMV [3]) for analyzing software metrics and dependencies between software artifacts. ADVIZOR [83] uses multiple linked views to depict software evolution: 2D and 3D matrix views, node-link diagrams, and bar charts.

Differences to 3D-CBV

The extension of the CBV to 3D space, 3D-CBV, extends the possibilities for analyzing multivariate structures. In contrast to previous work, it enables the combined, exploratory analysis of a software structure (hierarchy and relationships) and the values of three software metrics per node in a single view. Moreover, by using the HEB technique, it benefits from their reduced visual clutter. In this sense, it extends the software maps metaphor by additionally enabling the analysis of relationships between software artifacts. Its implementation as a prototype tool, PerfectMaintenance, demonstrates the applicability of the technique to industry-scale software systems.

3.6 Visual Modification of Software Structures

Refactoring tasks typically include fine-grained changes, they often modify only a few lines of code or a few methods. Reengineering, in contrast, often affects larger parts of a system (cf. Section 2.1.2). Techniques for reengineering software systems by visually (and virtually) modifying their structures therefore need to be able to depict these large structures in a readable manner. While refactoring operations *"vary greatly in their preconditions and mechanics"* [212], reengineering typically uses higher-level and less detailed operations, such as moving or merging software artifacts, which is not to say that these are simple operations on lower levels.

Selected Tools

Existing work on reengineering techniques can be separated into three categories: (1) Visual assessment tools focus on capturing and representing a system's design (see the tools presented in Section 3.5) while (2) recommendation systems use a system design as input to compute their output. For example, Hummel et al. [126] present a system that primarily aids during the design phase of software systems and during later reengineering tasks. It recommends similar (object-oriented) designs found in open-source repositories by matching 'footprints' of respective classes (class name and methods). Hutchens and Basili [128] use clustering algorithms to recommend a partitioning of a subject software system into modules; Schwanke and Hanson [234] propose a tool for improving a software system's modularization using neural networks. The tool solely focuses on moving modules and does not consider other actions for improving modularization.

[2] `http://www.cppdepend.com/`, last accessed 11/12/2013

Moreover, (3) general graph editing tools can be used in reengineering activities. Many of these tools represent structures using node-link diagrams such as UML variants. EDGE [203] is a customizable graph editor that has been used to visualize and edit software configurations. Several industrial tools support editing UML diagrams. This includes IBM Rational Rose[3], Microsoft Visio[4], the Fujaba Tool Suite[5], and Altova UModel[6]. TGraph [82] is a UML-variant, specifically targeted towards reengineering by merging structure and call dependencies in a single diagram. GraphWorks yEd[7] is a graph editor that uses node-link diagrams, but can also group nodes and use customized shapes to represent nodes.

Differences to D+D-CBV

Visual assessment tools take a rather passive role in reengineering tasks in that they solely represent the as-is structure of a software system. D+D-CBV, in contrast, takes an active role by primarily targeting activities that *modify* an as-is software structure. Recommendation systems are concerned with computing modifications, but typically do support users in assessing change impact.

In contrast to the graph editing tools, the D+D-CBV technique specifically extends the CBV technique, which enables editing structures using non-UML diagrams as visual representation. It thereby benefits from the fact that the CBV technique scales well for hierarchical structures with many relationships. The structure representation purposefully reduces details and groups relationships based on hierarchical containment to reduce visual clutter.

3.7 Summary and Overview of Closely Related Tools

As this chapter has shown, the techniques presented in this thesis address prominent challenges in software analysis and visualization. A variety of existing tools covers related questions in the domains structural analysis, behavioral analysis, and combined analysis of behavior-structure as well as behavior-behavior.

Table 3.1 provides an overview of the probably most closely-related tools in existing work, characterized along the dimensions 'application domain', 'used visualization techniques', and supported high-level 'tasks'. The tools are grouped by presented techniques that they are closely related to.

[3]`http://www-03.ibm.com/software/products/de/rosemod/`, last accessed 11/27/2013
[4]`http://office.microsoft.com//visio`, last accessed 11/13/2013
[5]`http://www.fujaba.de`, last accessed 11/13/2013
[6]`http://www.altova.com/de/umodel.html`, last accessed 11/13/2013
[7]`http://www.yworks.com/de/products_yed_about.html`, last accessed 11/13/2013

Tool or Author(s)	App. Domain	Used Vis. Techniques	Supported Tasks
TraceVis SyncTrace			
Jinsight [76]	BA	Linked views, icicle plot, sequence diagram, node-link diagram	Analysis of resource consumption, memory leaks, event sequences, sequence patterns, call trees
Code Bubbles [215]	BA	Linked views, icicle plot, stacked timelines, node-link diagram	Analysis of resource usage, thread dependencies, and locking patterns
Microsoft Concurrency Visualizer	BA	Stacked timelines, tables	Analysis of resource usage, thread scheduling, and call stacks
ViewFusion			
Extravis [68]	SA, BA	Linked views, CBV, MSV	Analysis of call relationships, structural composition, and event sequence
TimeLineTrees [43]	SA, BA	Linked views, node-link diagram, stacked timelines	Analysis of structural composition and event sequences
TraceDiff SyncTrace			
Holten and van Wijk [124]	SA	Icicle plots, HEB	Analysis of structural composition, comparison of tree topology
Code Flows [260]	SA	Icicle plot, shaded tube	Analysis of structural composition, comparison of tree topology
TreeVersity2 [110]	SA	Icicle plot, node-link diagram, table	Analysis of structural composition, comparison of tree topology and node values
3D-CBV D+D-CBV			
Solid* tool suite [220]	SA	CBV, Icicle plot, treemap, table	Analysis of structural composition, code clones, and call relationships
EvoSpaces [5]	SA	Grid/spiral, edges	Analysis of structural composition, and call relationships
IBM Rational Rose	SA	Node-link diagrams (UML)	Analysis and editing of structures
Microsoft Visio	SA	Node-link diagrams (UML)	Analysis and editing of structures

Table 3.1: *Overview of closely related tools in existing work; BA: Behavior Analysis; SA: Structural Analysis.*

Chapter 4

Analysis of Software Behavior

Since the early 1970s multi-threaded computing has evolved from a niche technology to a technology for the masses [52, 200]. Programming toolkits and libraries such as Qt [30], OpenMP [52], boost [138], or Intel's Threading Building Blocks [214] provide abstraction mechanisms to reduce the complexity that is inherent to writing code for multi-threaded systems. Despite these efforts, multi-threading adds a considerable number of new engineering challenges related to the runtime behavior of these concurrent software systems:

First, comprehending their behavior is difficult for several reasons. Engineers have to keep track of a lot of information per thread. Moreover, software traces capturing such behavior are typically massive data that pose several analysis and representation challenges [286].

Second, while debuggers and profilers are effective tools for debugging and performance optimization of single-threaded software systems, there are issues unique to multi-threaded software systems that are insufficiently supported by today's techniques and tools [69]. These issues include deadlocks, load imbalance, data-sharing patterns, race conditions, and contention. Timing and scheduling, which have to be considered in concurrent systems, are mostly ignored by existing tools. While, *"the execution of these* [single-threaded] *programs can be understood by a human sequentially stepping through the code"* [144], it is difficult for engineers to assess concurrent software behavior, which is typically not directly reflected by the source code [144]. That is, there is only a non-obvious mapping between control structures in the source code and a system's runtime behavior.

To overcome the lack of appropriate tool support for analyzing behavior of multi-threaded software systems, engineers have 'invented' workarounds to ease the forming of hypotheses. For instance, they often manually augment the source code with statements that print debug output to the console [125, 252]. Not only is this a time-consuming and tedious task, dumping output to the console is a resource intensive operation that may cause noticeable performance drop or even alter the system's timing significantly. Thus, engineers try to minimize usage of console output and apply guesswork instead. Consequently, mentally tracking the execution of a multi-threaded software system is a highly demanding and additionally error-prone task. With increasing importance and market share of multi-core machines and multi-threaded applications [97, 257], (visualization) tools that better support maintenance of multi-threaded software systems are more in demand.

This chapter first presents TRACEVIS (Section 4.1), a visualization concept that provides compacted, synchronized views on trace data of multi-threaded software systems. It enables the simultaneous analysis of multiple threads' behavior. Several advanced techniques for interaction and adaptive rendering, based on TRACEVIS, are subsequently presented in Section 4.2.

Second, SYNCTRACE (Section 4.3) builds upon the TRACEVIS concept and additionally enables the analysis of thread-interplay. SYNCTRACE further supports the exploratory nature of the underlying task and addresses *visual* scalability by a multiscale design that allows for trace analysis on several levels of abstraction, and complementary specialized interaction techniques.

Both sections describe techniques that target program-comprehension tasks related to multi-threaded behavior. Besides millions of runtime events triggered by such behavior, numerous threads may be spawned at runtime of which only a subset may be relevant for understanding specific aspects of its behavior. The selection of threads relevant to a task is hereby not the primary focus of the following sections. This selection can generally be driven by various criteria: Thread names (if available), execution of specific functionality, activity at specific points in time, access to certain variables or synchronization objects, etc. In the following, it is generally assumed that the analyzed software systems use threads in a non-massive way – as opposed to the massively parallel applications on graphic processing units (GPUs) and super computers – or that the number of threads to analyze can be reduced accordingly: Even for massively multi-threaded software systems, a representative set of threads can typically be determined, e.g., by reducing a set of worker threads, which execute the same code, to a single representative.

4.1 TraceVis: Visualization of Function-Call Sequences from Software Traces

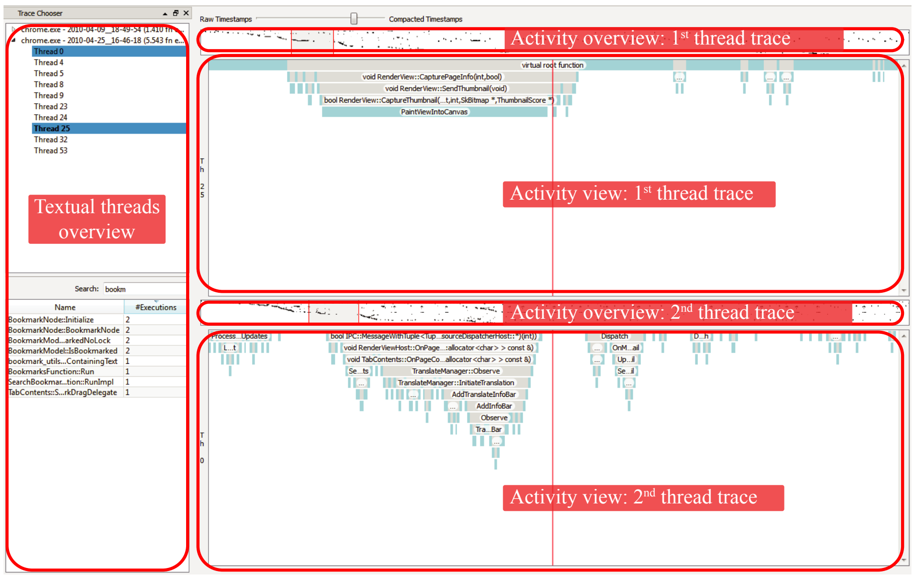
This section is based on the following publication(s):

- Jonas Trümper, Johannes Bohnet, Stefan Voigt, and Jürgen Döllner. 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, pages 325-330. IEEE, 2010. [301]
- Jonas Trümper, Johannes Bohnet, and Jürgen Döllner. Understanding Complex Multithreaded Software Systems by Using Trace Visualization. In Proceedings of the International Symposium on Software Visualization, pages 133-142. ACM, 2010. [300]

This section describes a visualization concept that is suitable for exploring large traces of multi-threaded software systems using multiple separate compacted views. In the preprocessing stage, the views' underlying trace data are compacted using a (nonlinear) time transformation that still enables synchronized exploration of the trace data. By this, the level of view compactness can be tuned continuously from (a) highly compact (strong nonlinearity) to make optimal use of available screen space to (b) linear time-to-screen-space mapping to reveal performance issues such as long waiting times.

4.1.1 Classification in a Task-Oriented Model

In the task-oriented model of Maletic et al. [175], the technique can be classified as follows: The *task* is to help users understand function-call sequences captured as large-scale traces in a space-efficient way, specifically to (a) analyze duration, order, and call relationships of function-call sequences; and (b) analyze concurrency of function calls across multiple thread traces. The *audience* includes software engineers who want to understand (concurrency-related) execution aspects of large software systems. The visualization *targets*


Trace Chooser
chrome.exe - 2010-04-09__18-49-54 (1.410 fn ...
chrome.exe - 2010-04-25__16-46-18 (5.543 fn e...
Thread 0
Thread 4
Thread 5
Thread 8
Thread 9
Thread 23
Thread 24
Thread 25
Thread 32
Thread 53
Textual threads overview
Search: bookm
Raw Timestamps
Compacted Timestamps
Activity overview: 1st thread trace
virtual root function
void RenderView::CapturePageInfo(int,bool)
void RenderView::SendThumbnail(void)
bool RenderView::CaptureThumbnail(...)
PaintViewIntoCanvas
Activity view: 1st thread trace
Activity overview: 2nd thread trace
Process...Updates
TranslateManager::Observe
TranslateManager::InitiateTranslation
AddTranslateInfoBar
AddInfoBar
Observe
Dispatch
Activity view: 2nd thread trace

Name	#Executions
BookmarkNode::Initialize	2
BookmarkNode::BookmarkNode	2
BookmarkMod...arkedNoLock	2
BookmarkModel::IsBookmarked	2
bookmark_utils...ContainingText	1
BookmarksFunction::Run	1
SearchBookmar...tion::RunImpl	1
TabContents::S...rkDragDelegate	1

Figure 4.1: TRACEVIS *user interface showing two thread traces.*

trace information (function calls and call durations) from multiple thread traces. These data are *represented* using a space-filling plot (for overviews) and icicle plots (for the call structure). Finally, the visualization *medium* consists of a standard screen with multiple linked views.

4.1.2 Visual Design

This section first describes the exploration workflow of the tool (Fig. 4.1) and the basic visualization design before the next sections discuss selected details of the tool implementation.

4.1.2.1 Exploration Workflow

Users start by selecting two or more thread traces $\{T_m\} \subset \mathbb{T}$ of a software trace $\mathbb{T}$ (see definitions in Section 2.1.4) to visualize. They then see an overview visualization of the trace data that enables them to find subsets of interest in the data. By clicking or panning in the overview, users can then adjust the subset shown in the detail view. Likewise, zooming and panning in the detail view allows for fine-grained adjustment of the shown subset. The detail view shows for any point in time in the current subset which functions were currently on the thread's stack. Any interactions with one view (detail view or overview) that adjust the currently shown subset of the trace data are automatically mirrored to all other views for all selected threads such that the views are always aligned. Users can thereby always visually correlate (simultaneous) activity in the selected threads by their relative alignment to each other.

4.1.2.2 Textual Threads Overview

For users to be able to choose a subset of all spawned threads, the analysis tool implements the aforementioned thread selection by executed functionality, which is a language-independent criterion. A *textual threads overview* summarizes the activities of selected threads, i.e., lists methods invoked in the context of a thread trace T_m together with their invocation count (Fig. 4.1). This way, users identify and select representative threads for subsequent visualization.

4.1.2.3 Trace Visualization

The visual design follows the well-known detail+overview and detail-on-demand principles in information visualization [237]. Each selected thread's trace data is depicted in an *activity overview* and an *activity view*, which shows a subset of the trace data. These views are synchronized to enable users to efficiently analyze the threads' behavior and jump between subsets of the trace data. The basic visualization concept for these two view types is as follows:

Activity views Each activity view is associated to a thread trace and shows its event sequence by a 2-dimensional graph using an icicle-plot metaphor [154]. In the following, this representation is synonymously referred to as 'activity view'. The event sequences are drawn such that a thread's call stack can be analyzed for every point in time. For this, time is mapped along the x axis and stack depth along the y axis (Fig. 4.2): An activity view displays function calls f of a thread trace T_m as rectangular cells (bars) in T_m-depth-order from top to bottom, horizontally positioned on their t^s, t^e start and end moments. Invocation relations between methods are thereby given implicitly, i.e., by the vertical stacking of the bars representing individual function calls. Zooming only affects the time axis (x) such that the graph is compressed (or stretched) in x direction, i.e., the aspect ratio of the sequence graph changes. Consequently, stack height (y axis) is preserved, which in turn also preserves the readability of method labels. This is useful when analyzing high-frequency traces that contain many short-duration events.

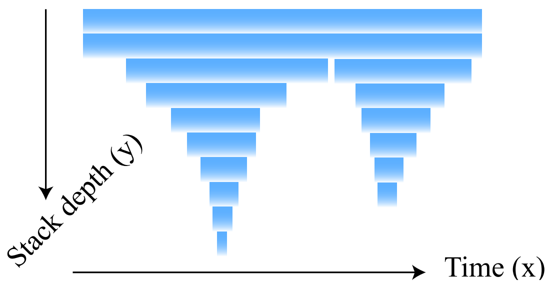


Figure 4.2: *Concept for the trace visualization in the activity views.*

Activity overviews Activity overviews facilitate quick orientation within the trace data. *Time span markers* denote the current subset that is shown in detail in the respective activity view, *activity markers* denote invocations of methods hovered in the activity view (Fig. 4.3). In the following, two different types of overviews are available:

- **Pattern-Based Overview:** As Ware [276] points out, "the human visual system is a pattern seeker of enormous power". To utilize this power, the pattern-based activity overview [300, 301] [32] (Fig. 4.4(a)) derives visual patterns from execution patterns. This overview computes a 2-dimensional grid in which methods are mapped along the grid's y axis and time is mapped along the x axis. When a method is executed at a specific point in time, the respective grid cell is colored black, white otherwise. This serves two purposes: First, repeated execution of the same (or similar) functionality causes repeating visual patterns in the overview (Fig. 4.5). Second, high-frequency calls to varying functions show up as high-frequency patterns in the overview.

  The computed grid is finally scaled to fit the view's size on the screen using anti-aliasing. By anti-aliasing, the visualization produces results in a similar way as the Information Mural [131] (cf. Section 3.1.2).

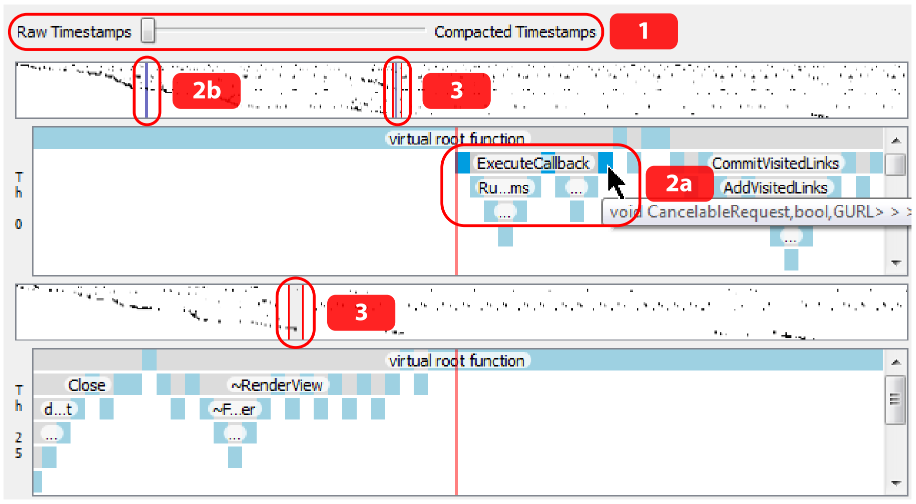
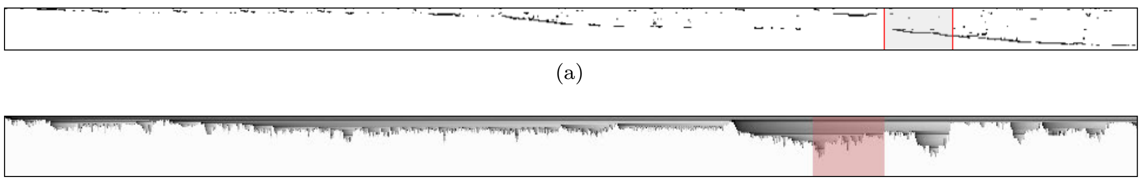


Figure 4.3: *Trace visualization shows two thread traces. (1) A slider enables configuring the timestamp scaling. (2a) Activity markers highlight invocations of methods hovered in the activity view within each (2b) activity overview . The detailed depicted subset (in the activity view) is denoted by (3) time span markers in the respective activity overview.*

- **Structure-Based Overview:** Call-stack structure and call-stack depth are other important information for gaining an overview of trace data. The pattern-based overviews, however, cannot show this. Hence, a complementary overview that shows this information is the space-filling depiction of the complete trace data as icicle plot. In other words, the structure-based overview (Fig. 4.4(b)) shows a zoomed-out version of the activity view. In this miniaturized representation, outliers such as deep and shallow call stacks or long-duration calls can quickly be spotted. Moreover, color coding can for instance be used to implement a lightweight way of highlighting the two forms of repetitive behavior on function level: Loops and recursion. By encoding a function's identity as color of the respective bar in the icicle plot, horizontal stripes of the same color indicate repetitions (Fig. 4.6(b)), vertical bands indicate recursion (Fig. 4.6(b)).

Due to the inherent scaling of these space-filling approaches, multiple function-call bars might be mapped to a single screen pixel. That is, the mapping of bars to screen pixels is not guaranteed to be injective. A limitation of these overviews is thus that in such cases, very small-scale repetitive behavior (representing short-lived function calls) may be invisible in the overviews.



(b)

Figure 4.4: *Comparison of activity overviews: (a) Pattern-based overview that highlights repetitive behavior and (b) structure-based overview that shows call stack structure and stack depth.*

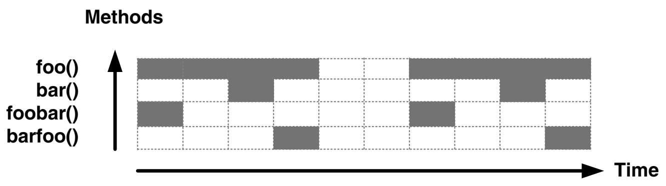


Figure 4.5: *Concept for the pattern-based activity overview: Multiple execution of the same (or similar) functionality is represented by repeating visual patterns.*

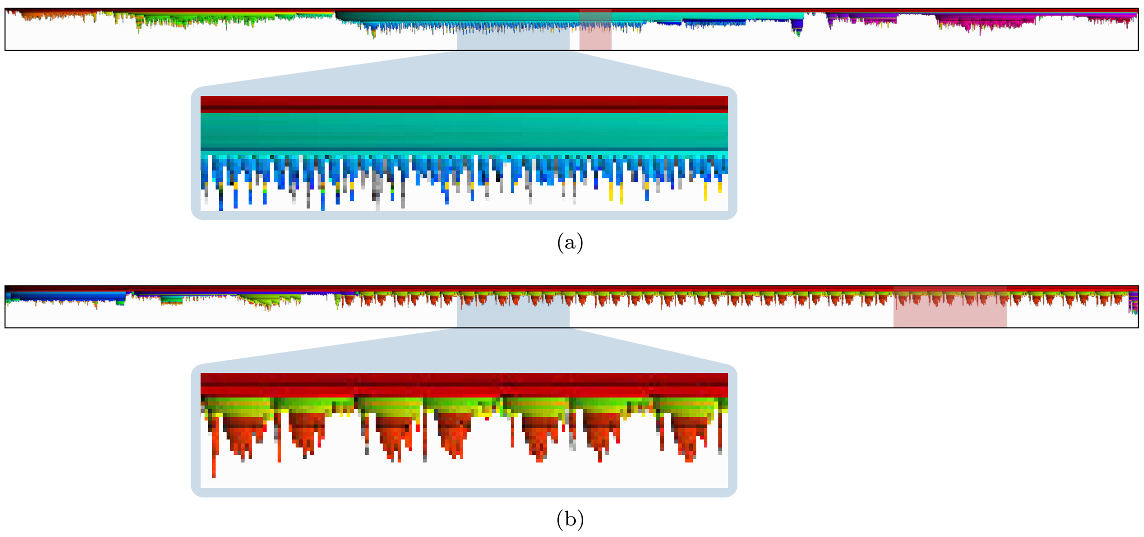


Figure 4.6: *Examples of higher-level repetitive behavior highlighted by color in structure-based activity overviews; Color encodes function identity: (a) Recursions (wide cyan-colored block at the top) and (b) Loop (vertical green-orange stripes at the bottom). The inlays each show a magnified version of a small fraction of the overview.*

4.1.3 Configurable Level of View Compactness

Execution durations of methods typically vary significantly: Whereas execution of method *foo* takes 20,000 milliseconds on average, the execution of method *bar* may take only a few nanoseconds on average. Even for depicting a single thread's activity, this poses a major challenge: When depicting these raw execution durations, lengthy method executions would span multiple screens while very short method executions might span only a single pixel (Fig. 4.7). Consequently, a time warping—a *scaling*—is required for efficient exploration.

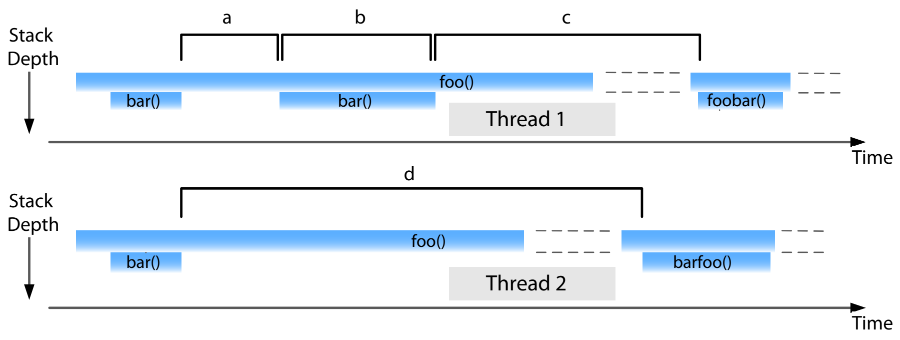


Figure 4.7: *View Compaction: Raw time stamps. Execution durations of methods, e.g.,* foo() *and* bar()*, vary significantly.*

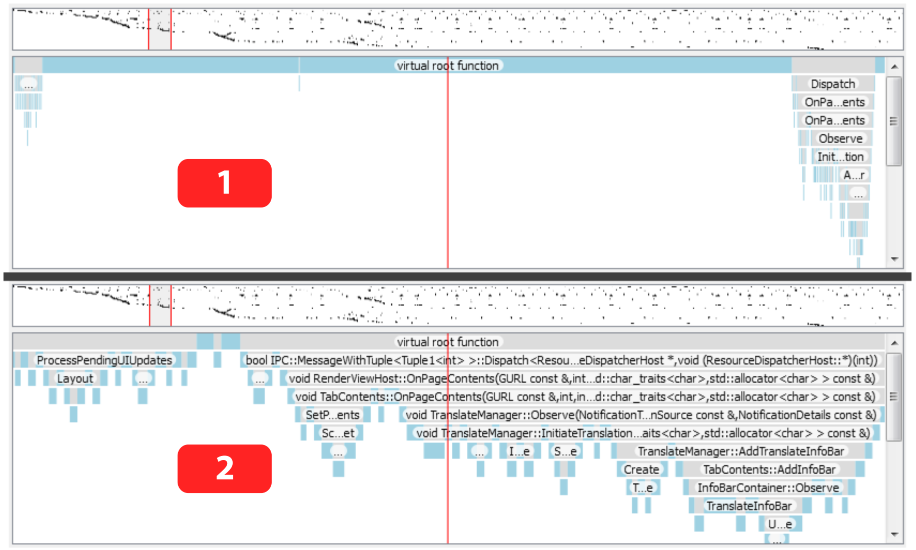

virtual root function
Dispatch
OnPa...ents
OnPa...ents
Observe
Init...tion
A...r
1
virtual root function
ProcessPendingUIUpdates
bool IPC::MessageWithTuple<Tuple1<int> >::Dispatch<Resou...eDispatcherHost *,void (ResourceDispatcherHost::*)(int))
Layout
void RenderViewHost::OnPageContents(GURL const &,int...d::char_traits<char>,std::allocator<char> > const &)
void TabContents::OnPageContents(GURL const &,int,in...d::char_traits<char>,std::allocator<char> > const &)
SetP...ents
void TranslateManager::Observe(NotificationT...nSource const &,NotificationDetails const &)
Sc...et
void TranslateManager::InitiateTranslation...aits<char>,std::allocator<char> > const &)
I...e
S...e
TranslateManager::AddTranslateInfoBar
Create
TabContents::AddInfoBar
T...e
InfoBarContainer::Observe
TranslateInfoBar
U...e
2

Figure 4.8: *Activity view depicting the same sequence subset using (1) linear and (2) logarithmic scaling. Linear scaling requires to zoom out (low detail) due to long thread idle time. The same set of method executions can be depicted in more detail using logarithm-based scaling.*

TraceVis provides configurable scaling for time spans Δt between successive timestamps t_i, t_j, which essentially is a focus+context approach. The scaling is either linear or logarithm-based. Linear scaling enables to analyze raw execution time stamps and as such is suited for performance analysis and optimization tasks. Linear scaling, however, hinders program-comprehension tasks. For example, engineers are here typically interested in the context surrounding specific function calls to understand the causality of events, and also require correlating those insights with higher-level information to understand which system components are involved (cf. Section 2.3.1). With linear scaling, they would have to explore execution traces at high detail (zoomed in) to analyze specific function calls, but use medium detail in order to grasp the context of the currently depicted function calls (Fig. 4.8), and low detail for correlating this with higher-level information.

In contrast, logarithm-based scaling is better suited for program-comprehension tasks, as short time spans are shrinked only slightly while long time spans are shrinked massively. Hence, idle times between method executions typically 'vanish' such that relevant parts of the trace can be explored in more detail while still maintaining the execution context. To achieve a configurable logarithmic scaling, the activity views do not use the plain natural logarithm. Instead, Δt is multiplied with a factor θ and the result of the logarithm is divided by the same factor to achieve configurable shrinking. This is motivated by the approximation of the natural logarithm by a Taylor series:

$$\ln(1 + \Delta t) = \sum_{k=1}^{\infty} (-1)^{k+1} \frac{\Delta t^k}{k} = \Delta t - \frac{\Delta t^2}{2} + \frac{\Delta t^3}{3} - \frac{\Delta t^4}{4} \pm \cdots$$

The scaling function then reads as:

$$
\begin{aligned}
logscale(\Delta t, \theta) &= \frac{\ln(1+\theta \cdot \Delta t)}{\theta} \\
&= \frac{\sum_{k=1}^{\infty}(-1)^{k+1}\frac{(\theta \cdot \Delta t)^k}{k}}{\theta} \\
&= \frac{\theta \cdot \Delta t}{\theta} - \frac{\theta^2 \cdot \Delta t^2}{2\theta} + \\
&\quad \frac{\theta^3 \cdot \Delta t^3}{3\theta} - \frac{\theta^4 \cdot \Delta t^4}{4\theta} \pm \cdots \\
&= \Delta t - \frac{\theta \cdot \Delta t^2}{2} + \\
&\quad \frac{\theta^2 \cdot \Delta t^3}{3} - \frac{\theta^3 \cdot \Delta t^4}{4} \pm \cdots
\end{aligned}
$$

Whereas θ cancels out in the first term of the power series, all subsequent terms contain θ. Hence, shrinking of Δt can essentially be adjusted continuously between linear *and* logarithmic (Fig. 4.9) as for $\theta \to 0$ all terms but the first are eliminated:

$$\lim_{\theta \to 0} \frac{\ln(1+\theta \cdot \Delta t)}{\theta} = \Delta t$$

In contrast to that, for $\theta > 0$, all subsequent terms account for the power series and scaling is logarithm-based.

4.1.4 Synchronization of Multiple Activity Views

Applying the nonlinear scaling *logscale* separately to each thread's events would hinder the visual comparability of concurrent method executions: Method executions or idle times in separate threads (time spans a, b, c and d in Fig. 4.7) that happen before the execution of *foobar* and/or *barfoo* may be scaled differently (Fig. 4.10(a)) such that method entry and exit timestamps of *foobar* and *barfoo* are warped and no longer visually aligned. In other words, the partial order of scaled timestamped events (across multiple selected threads) may differ from the order of unscaled timestamped events. Thus, the view synchronization would fail to provide usable visual results.

Consequently, a key challenge is the way to synchronize multiple views – each depicting a single thread's activity – such that collaboration between threads gets visible. In the absence of effective synchronization, users would need to compare event timestamps manually to assess whether a specific method execution in the context of thread i is actually concurrent to another method execution in the context of thread j.

To address this issue and to preserve visual comparability of concurrent method executions, TraceVis applies on-demand scaling for the selected threads and the depicted subset. That is, it merges all events of all selected threads into a single ordered queue (Fig. 4.10(b)). Concurrent events (having the same timestamp) are allowed to be enqueued at the same place within the queue. Thus, each Δt of successive events in the queue can safely be scaled whilst preserving partial event order across all selected threads (Fig. 4.10(c)).

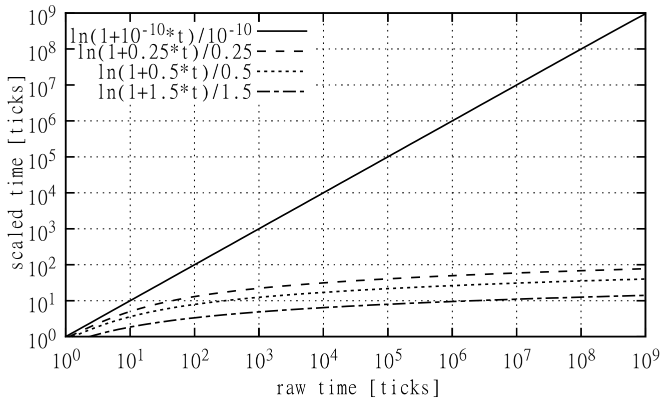
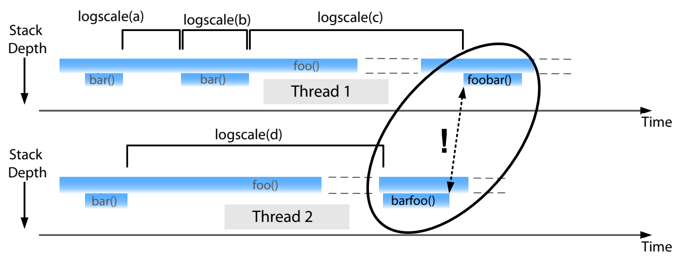
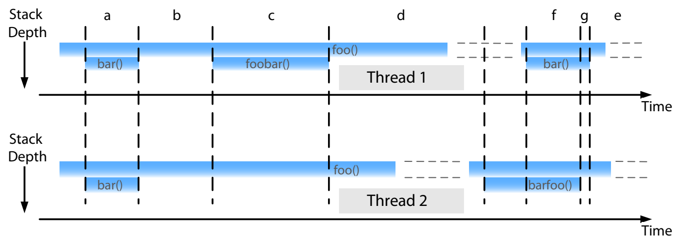


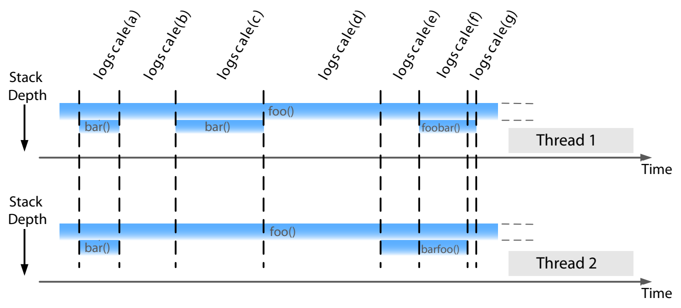
Figure 4.9: *Plots of logscale for different values of θ: The plot for $\theta = 1 \cdot 10^{-10}$ is already linear for Δt in $[1, 10^9]$. Larger values of θ, e.g., $\theta = \frac{1}{4}$ and $\theta = \frac{1}{2}$, allow for logarithm-based shrinking of Δt.*



(a) *Logarithmic scaling per thread. The execution of* foobar() *in thread 1 is no longer depicted as concurrent to the execution of* barfoo() *in thread 2.*

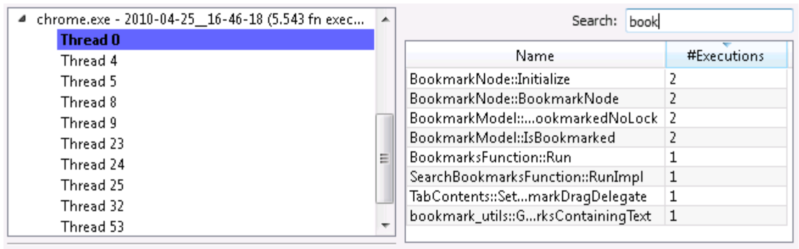


(b) *Partial order-preserving scaling: All selected threads' timestamped events are merged into a single queue. Timestamp differences (a, b, c, ...) are now calculated based on the merged queue.*



(c) *Logarithmic scaling based on the merged queue (logscale(a), logscale(b), ...): Visual comparability of* foobar() *and* barfoo() *is preserved.*

Figure 4.10: *Preserving visual comparability with logarithmic scaling.*

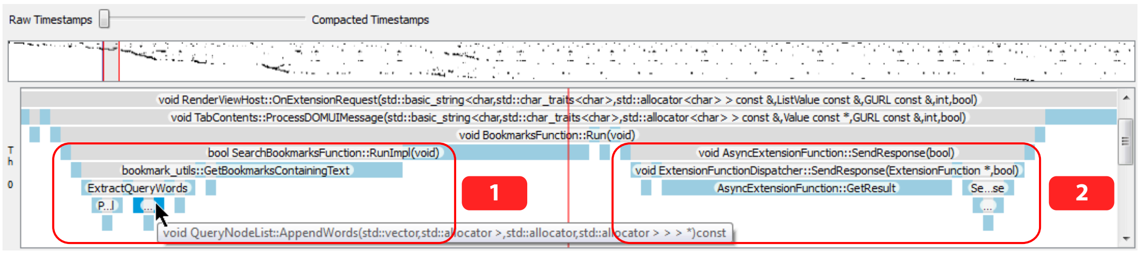

chrome.exe - 2010-04-25__16-46-18 (5.543 fn exec...

Thread 0
Thread 4
Thread 5
Thread 8
Thread 9
Thread 23
Thread 24
Thread 25
Thread 32
Thread 53

Search: book

Name	#Executions
BookmarkNode::Initialize	2
BookmarkNode::BookmarkNode	2
BookmarkModel::...ookmarkedNoLock	2
BookmarkModel::IsBookmarked	2
BookmarksFunction::Run	1
SearchBookmarksFunction::RunImpl	1
TabContents::Set...markDragDelegate	1
bookmark_utils::G...rksContainingText	1

Figure 4.11: *Textual threads overview: Identifying relevant threads in* Chromium.



void RenderViewHost::OnExtensionRequest(std::basic_string<char,std::char_traits<char>,std::allocator<char> > const &,ListValue const &,GURL const &,int,bool)
void TabContents::ProcessDOMUIMessage(std::basic_string<char,std::char_traits<char>,std::allocator<char> > const &,Value const *,GURL const &,int,bool)
void BookmarksFunction::Run(void)
bool SearchBookmarksFunction::RunImpl(void)
bookmark_utils::GetBookmarksContainingText
ExtractQueryWords
void QueryNodeList::AppendWords(std::vector,std::allocator >,std::allocator,std::allocator > > > *)const
void AsyncExtensionFunction::SendResponse(bool)
void ExtensionFunctionDispatcher::SendResponse(ExtensionFunction *,bool)
AsyncExtensionFunction::GetResult

Figure 4.12: Chromium *bookmark search in detail, scaling tuned to linear: (1) Database query and (2) result dispatching.*

This approach both scales time and synchronizes multiple views by focusing on a specific point in time in the trace and by updating all views such that the views are centered around this point in time.

4.1.5 Application in Industry-Scale Software Systems

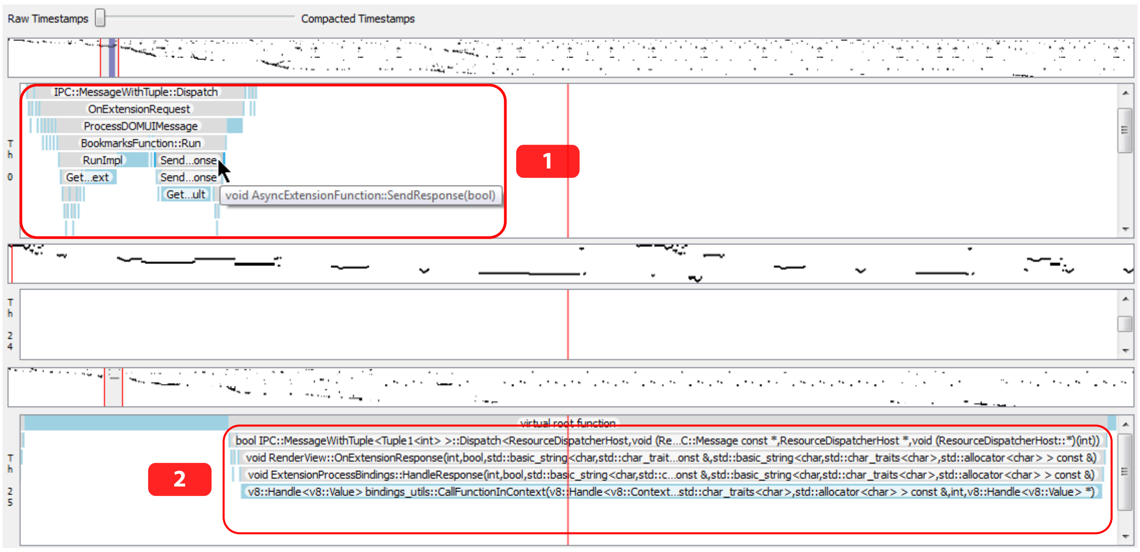
This section discusses two applications of TraceVis to industrially developed, multi-threaded software systems: (1) *Chromium*[1], the open-source code base of Google's web browser Chrome[2] and (2) *Building Reconstruction* (BRec), a tool for reconstructing 3D building models from laser scan data from virtualcitySYSTEMS GmbH[3].

4.1.5.1 Chromium: Bookmark Search

Chromium (version 6.0) consists of approximately 4 million source lines of code[4] (SLOC), thereof approximately 2.7 million SLOC in C and C++. According to the source code repository, more than 390 authors contributed to the code base. The implementation concepts of Chromium strongly rely on deferred processing, e.g., tasks are 'posted' by one thread and successively processed by another dedicated thread. Hence, engineers concerned with fixing performance weaknesses in Chromium likely face understanding problems that also occur during development of complex closed-source multi-threaded systems.

In this case study, we aim at identifying the main bottleneck in Chromium's bookmark search: Searching for popular terms takes considerably more time than expected. Our initial suspicion is that the database query could be responsible for the slowdown. To verify this, we identify relevant parts of Chromium's implementation (cf. Section 2.1.4) and re-run the scenario with selective instrumentation. We exercise the bookmark search scenario and import the resulting trace into our tool.

[1]`http://www.chromium.org`, last accessed 10/02/2012
[2]`http://chrome.google.de`, last accessed 10/02/2012
[3]`http://www.virtualcitysystems.de`, last accessed 10/02/2012
[4]`http://www.ohloh.net/p/chrome/analyses/latest`, last accessed 10/20/2010



```
bool IPC::MessageWithTuple<Tuple1<int> >::Dispatch<ResourceDispatcherHost,void (Re...C::Message const *,ResourceDispatcherHost *,void (ResourceDispatcherHost::*)(int))
void RenderView::OnExtensionResponse(int,bool,std::basic_string<char,std::char_trait...onst &,std::basic_string<char,std::char_traits<char>,std::allocator<char> > const &)
void ExtensionProcessBindings::HandleResponse(int,bool,std::basic_string<char,std::c...onst &,std::basic_string<char,std::char_traits<char>,std::allocator<char> > const &)
v8::Handle<v8::Value> bindings_utils::CallFunctionInContext(v8::Handle<v8::Context...std::char_traits<char>,std::allocator<char> > const &,int,v8::Handle<v8::Value> *)
```

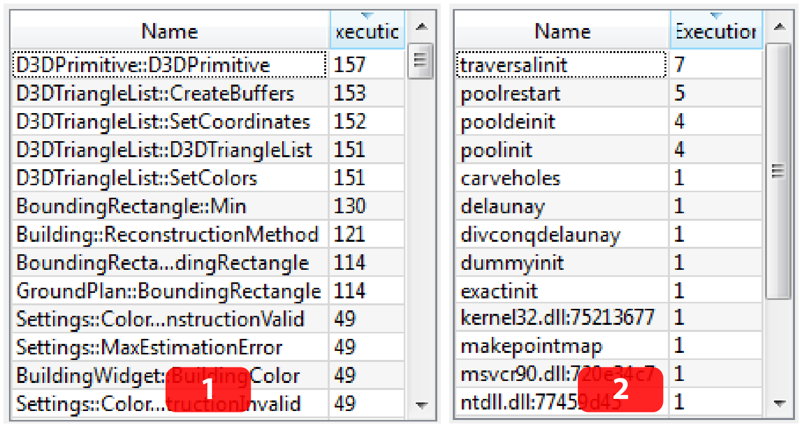
Figure 4.13: *(1) Chromium bookmark search and result dispatching (see Fig. 4.12) as well as (2) asynchronous result handling. Scaling is tuned to linear.*

Using the textual threads overview (Fig. 4.11), 3 out of 10 threads turn out to be relevant for our analysis task: Searching for 'bookmark' in the textual threads overview yields hits in one thread. Two other threads are identified as relevant as they seem to be concerned with deferred processing of jobs.

We initially seek through the trace data by using the zoom and pan facilities of the activity view with logarithm-based scaling. We identify key points for further inspection in the trace by remembering the respective patterns that are enclosed by time span markers in the activity overview. Subsequently, we issue a search for the term 'bookmark' in our tool and yield 9 hits. In contrast to that, a search for the same term in Chromium's source code results in approximately 19,000 hits in more than 800 files. One of the hits in the trace data is *BookmarksFunction::Run*. As that draws our attention, we inspect it first, tuning the mathematical scaling operation to linear.

It turns out that the respective method's invocation actually issues a query in the bookmark database (Fig. 4.12): (1) *BookmarksFunction::Run* calls lower-level functionality to process the database query. Among others, *ExtractQueryWords* is called. After the database query returns, (2) *BookmarksFunction::Run* invokes *SendResponse* in class *AsyncExtensionFunction* to enqueue the result-processing job. Processing of the search result is done asynchronously in another thread (Fig. 4.13): After the database query is finished and the job enqueued (1), *RenderView*'s method *OnExtensionResponse* is invoked via an IPC to process the query result. *OnExtensionResponse* hands result processing over to *ExtensionProcessBindings::HandleResponse* that subsequently calls some JavaScript functionality in Chromium's JavaScript engine V8.

In contrast to our initial expectations, the following facts were observed: First, the database query is not the performance bottleneck. Processing the result in *OnExtension-Response* is. It consumes more than thrice the time of the database query. Further, query execution and result processing, while being executed in separate threads, are actually executed sequentially. With true parallel execution, processor utilization on multicore systems could be increased and first search results could become visible in less time. Second, only 2 of 3 chosen threads were actually relevant for our comprehension task (threads 0 and 25). The third thread (24) did not execute any relevant functionality.



Name	xecutic
D3DPrimitive::D3DPrimitive	157
D3DTriangleList::CreateBuffers	153
D3DTriangleList::SetCoordinates	152
D3DTriangleList::D3DTriangleList	151
D3DTriangleList::SetColors	151
BoundingRectangle::Min	130
Building::ReconstructionMethod	121
BoundingRecta...dingRectangle	114
GroundPlan::BoundingRectangle	114
Settings::Color...nstructionValid	49
Settings::MaxEstimationError	49
BuildingWidget::BuildingColor	49
Settings::Color...tructionInvalid	49

Name	Executior
traversalinit	7
poolrestart	5
pooldeinit	4
poolinit	4
carveholes	1
delaunay	1
divconqdelaunay	1
dummyinit	1
exactinit	1
kernel32.dll:75213677	1
makepointmap	1
msvcr90.dll:720e34c7	1
ntdll.dll:77459d45	1

Figure 4.14: *Textual threads overview: (1)* BRec's *coordinator thread and (2) a worker thread.*

4.1.5.2 BRec: Rendering Process

The *BRec* software system of virtualcitySYSTEMS GmbH[5] reconstructs and visualizes 3D building models from raw laser scan data. Its rendering engine triangulates the building models in parallel with multiple threads. *BRec's* development started more than 10 years ago. The code base comprised approx. 100k LOC written in C/C++ with 15 engineers on average working on it.

In this case study, an engineer is concerned with speeding up the rendering process. The engineer starts with recording a trace of the complete rendering functionality. Using the textual threads overview, he identifies one coordinating thread and a vast amount of worker threads that all execute the same utility functionality: Delaunay triangulation. Identification of the coordinating thread is based on two facts: (1) It executes the main window's event loop and (2) it exhibits significantly higher invocation counts than the other threads (Fig. 4.14). The engineer selects the coordinating thread and two worker threads as the worker threads execute the same set of methods with the same invocation counts.

The engineer explores the trace using logarithm-based scaling to gain an initial overview of the rendering functionality and of how the coordinating thread handles worker threads. The first observation is that there is one coordinating method, which is responsible for the whole rendering process: *BuildingWidget::UpdateRenderer*. It creates instances of class *SolidRenderer* that each instantiate *Triangulation3D2*. The engineer notices that *Triangulation3D2* spawns two worker threads that calculate a triangulation (Fig. 4.15). Each worker threads' time span marker spans the thread's complete lifetime in the activity overview. This shows that the thread terminates immediately after finishing triangulation and complements the initial observation, which caused the engineer to classify them as worker threads. In addition, the two threads are executed sequentially, which can be caused by either a software-related problem (too much synchronization) or a hardware-related phenomenon (all CPU cores used).

As a worker thread's sole purpose is to calculate a single Delaunay triangulation, a significant number of threads is spawned during the rendering process. Although thread creation is considered to be cheap in comparison to process creation, its overhead cannot be neglected. Hence, the engineer decides to use a thread pool for triangulation jobs to speed up rendering by reducing thread creation costs.

[5]`http://www.virtualcitysystems.com`, last accessed 5/10/2011

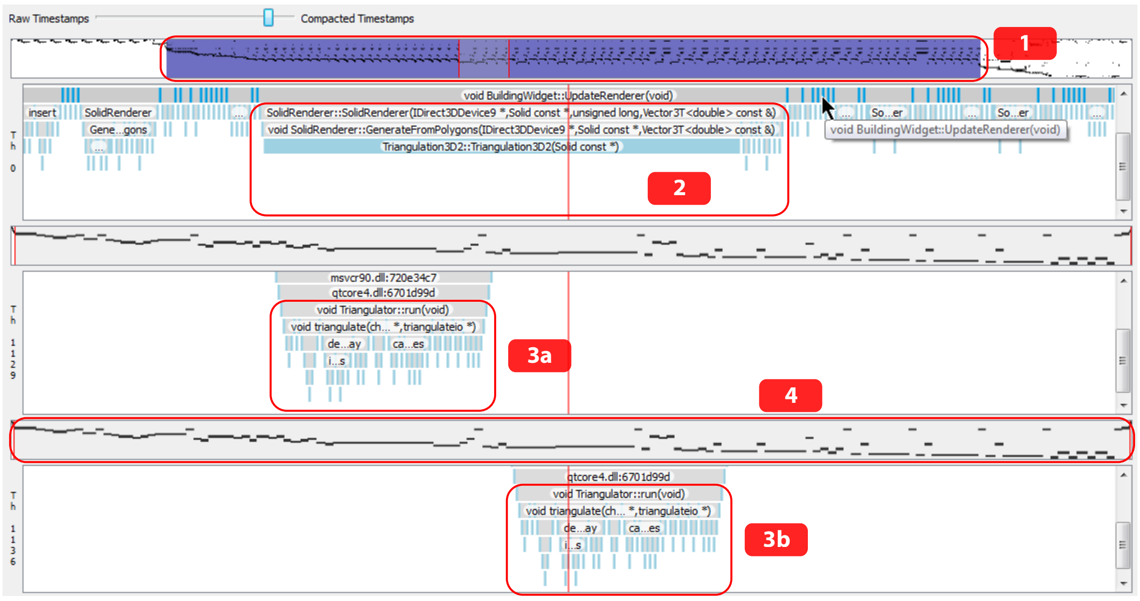

Raw Timestamps
Compacted Timestamps
1
void BuildingWidget::UpdateRenderer(void)
insert
SolidRenderer
SolidRenderer::SolidRenderer(IDirect3DDevice9 *,Solid const *,unsigned long,Vector3T<double> const &)
void SolidRenderer::GenerateFromPolygons(IDirect3DDevice9 *,Solid const *,Vector3T<double> const &)
Triangulation3D2::Triangulation3D2(Solid const *)
Gene...gons
void BuildingWidget::UpdateRenderer(void)
2
msvcr90.dll:720e34c7
qtcore4.dll:6701d99d
void Triangulator::run(void)
void triangulate(ch... *,triangulateio *)
de...ay
ca...es
3a
4
qtcore4.dll:6701d99d
void Triangulator::run(void)
void triangulate(ch... *,triangulateio *)
3b

Figure 4.15: BRec's *rendering engine: (1) Method* UpdateRenderer *of class* BuildingWidget *coordinates the rendering process. (2) The coordinator thread spawns two worker threads (3a, 3b), each calculating a triangulation. (4) Activity markers of threads 1136 and 1129 indicate that the activity views (below) depict the threads' complete activity.*

A surprising result of this case study was that two potential performance issues (thread creation, sequential execution of worker threads) could be located without needing to explore the trace with linear scaling, i.e., exploring the raw timestamps that show "real" waiting times. Different non-linearity grades of the logarithmic scaling were sufficient to understand the multi-threading behavior and to locate the main performance bottleneck in the rendering process.

4.1.6 Discussion

Generality The visualization, though presented for software traces, works on any hierarchical event sequence that is compatible with the given model definition (Section 2.1.4). The view compaction and synchronization technique can use any 1-dimensional function to implement the compaction step.

Visual Scalability The view compaction technique provides a way to implement an arbitrary prioritization function for weighted event sequences (e.g., duration as weight for function calls in software traces). Further, the presented technique is suitable for showing a relatively small number of thread traces in parallel due to the vertical stacking of the views. However, in the case of massive multi-threaded software traces, one can virtually increase the scalability of the technique by selecting representative threads.

Ease of Use and Flexibility The interaction techniques allow for easy user input; to explore the underlying trace data, users only need to learn how to point, click, zoom, and pan. By these inputs, users can adjust the analyzed subset of the activity data and adjust level of detail. Tuning the view compaction automatically updates all views while maintaining the views' focus and context.

Limitations Delimiting equal temporal distances in compacted views can be hard due to potential non-linearity in the compaction, but can be alleviated by temporarily tuning the compaction to linear. Examining very deep call stacks, which may occur due to recursion, requires vertical scrolling or aggregation just as examining subsets of a larger trace (many events) requires horizontal scrolling. The latter is addressed in Section 4.2.

Benefits The technique, as dynamic analysis technique, exploits that a user's search space to understand system behavior is reduced to those parts of the implementation that are relevant for a given execution. Thus, even engineers with little knowledge of a system's implementation are able to quickly identify relevant parts of its implementation and understand their interactions. The approach and its scalability was demonstrated by means of case studies on two large-scale and multi-threaded software systems; performance weaknesses in these systems and non-obvious system behavior have been revealed.

4.2 Advanced Interaction and Multiscale Rendering Techniques for TraceVis

This section is based on parts of the following publication(s):

- Jonas Trümper, Alexandru Telea, and Jürgen Döllner. ViewFusion: Correlating Structure and Activity Views for Execution Traces. In Proceedings of the Conference on Theory and Practice of Computer Graphics, pages 45–52. Eurographics, 2012 [303]
- Jonas Trümper, Jürgen Döllner, and Alexandru Telea. Multiscale Visual Comparison of Execution Traces. In Proceedings of International Conference on Program Comprehension, pages 53-62. IEEE, 2013. [305]
- Benjamin Karran, Jonas Trümper, and Jürgen Döllner. SyncTrace: Visual Thread-Interplay Analysis. In Proceedings (electronic) of the Working Conference on Software Visualization, pages 1-8. IEEE, 2013. [296]

Activity views for software traces display hierarchical time-dependent data that, in our case, is represented by an icicle-plot metaphor as described in Section 4.1.2.3. This section explains several basic techniques for user interaction with and rendering of activity views for software traces. The techniques are used later to develop application-specific visualization in Sections 4.3, 5.1, and 5.2.

Trace data in our software traces is tree-structured. Often not only single items but subtrees are of interest during the exploration, and, therefore users also need means to interact with such sets of focused items. Suitable definitions of the focus item and focus subset are designed by following the visual information-seeking mantra: Overview, zoom-and-filter, and details on demand [237], via three interaction modes. Furthermore, the data is typically massive, so a) space-efficient display is important to achieve high information density, and b) means for automatic aggregation help reduce visual clutter and rendering overhead.

4.2.1 Interaction Techniques

Apart from basic zoom and pan, this section describes a set of advanced interaction techniques for selecting sets of items in activity views and for semantic aggregation of activity views.

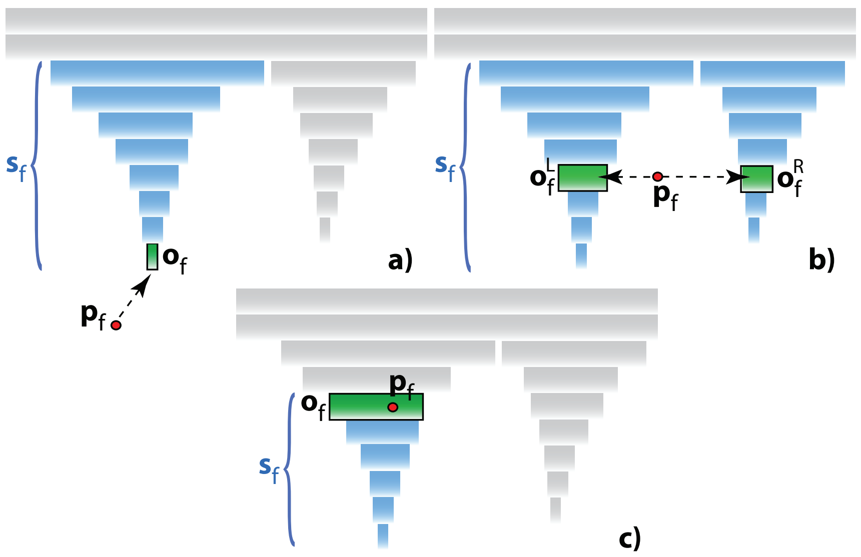


Figure 4.16: *Interaction modes: (a) Overview, (b) approach, and (c) detail. Focus item is green and focus subset is blue.*

4.2.1.1 Adaptive Brushing

The brushing technique enables users to implicitly select sets of items simply by moving the mouse. Based on the geometrical distance between mouse and items, the technique automatically adapts by selecting appropriate brushing modes.

We first introduce the three key elements of its interaction design: Focus point, focus item, and focus subset. The *focus point* $\mathbf{p}_f = (x_f, y_f) \in \mathbf{R}^2$ is the current mouse position. The *focus item* is the object $o_f \in V_m$ closest to the focus point in the view V_m that visualizes a thread trace T_m. A view V_m is selected as *input target* by placing the focus point inside the view's bounding box. The *focus subset* $s_f \subset V_m$ is a set of elements in the view V_m; s_f contains the focus item o_f and also other objects that are semantically and spatially close to o_f in V_m.

Overview Mode In this mode (Fig. 4.16 a), one typically visualizes a zoomed-out activity view, looking for 'salient' icicles, e.g., deep call stacks surrounded by shallow execution areas. One enters overview mode when the focus point $\mathbf{p}_f$ is outside *and* below the rendered icicle plot. We next define the focus item o_f as the closest (in Euclidean distance sense) icicle-plot bar to $\mathbf{p}_f$. We also define the focus subset s_f as the path starting at the root of T_m and ending at o_f, whose elements are visible in the activity view at the current zoom and pan levels.

Approach Mode In this mode (Fig. 4.16 b), users moves the mouse closer to the icicle plot: The focus point $\mathbf{p}_f$ is now *between* two bars of the icicle plot rather than below all icicles as in the overview mode, but still outside the icicle-plot itself. This mode is useful when one has decided to focus on an area within a trace dataset, but is not sure which specific call stacks within that trace deserve further attention. We define *two* focus items o_f^L and o_f^R as the closest items on the x axis to the left, respectively right, of $\mathbf{p}_f$. The focus subset s_f contains now the visible paths in E that pass through o_f^L and o_f^R.

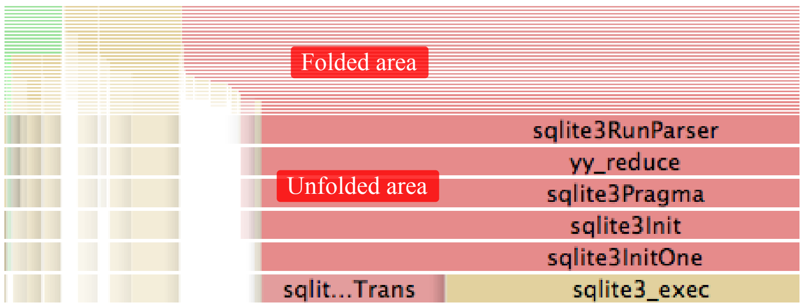


Figure 4.17: *Stack* folding*: Panning upwards successively shrinks upper stack levels one-by-one (folding); panning downwards successively resets the size of upper stack levels to normal one-by-one (unfolding).*

Detail Mode In this mode (Fig. 4.16 c), users move the mouse, thus $\mathbf{p}_f$, inside the icicle plot, e.g., decide to focus on the call stack below a given function call. We set o_f to the icicle-plot bar under $\mathbf{p}_f$, and s_f to the path from o_f downwards in T_m.

4.2.1.2 Semantic Aggregation for Activity Views: Stack Folding

In addition to the time compaction described in Section 4.1.3, this section presents an alternative to vertical panning: The *folding* technique, which provides an orthogonal approach for handling large traces that capture deep call stacks: Depth levels in an icicle plot are reduced in size (folded) by a vertical drag operation (Fig. 4.17) and *unfolded* by dragging in the opposite direction. For this, the activity view basically applies a graphical fisheye view [230] using a magnification function barheight(f) that is defined on a function call f. A simple two-step function that returns the height of a bar can be defined as follows:

$$\text{barheight}(f) = \begin{matrix} h_f & \text{if } \rho_l \geq \text{depth}(f) < \rho_u \\ h_u & \text{otherwise} \end{matrix} . \tag{4.1}$$

For a thread trace T, the lower and upper folding thresholds $\rho_l, \rho_u \in [0, \text{maxDepth}(T)]$ can then be modulated upon drag. Depending on the use case, the heights h_f, h_u for the folded and the unfolded bars can be modified accordingly as well. For example, to implement a folding behavior that successively shrinks bars starting at high-level function calls, one would set $\rho_l, \rho_u = 0$ and modulate ρ_u upon vertical drag. Vice versa, by defining barheight(f) as

$$\text{barheight}(f) = \begin{matrix} h_f & \text{if } \rho_l > \text{depth}(f) \leq \rho_u \\ h_u & \text{otherwise,} \end{matrix} \tag{4.2}$$

setting $\rho_l, \rho_u = \text{maxDepth}(T)$, and modulating ρ_l, one would achieve a folding that successively shrinks upwards, starting at low-level function calls.

More complex magnification functions using a continuous transition between the two thresholds, can of course be put in place. By this folding technique, users can make visible lower or higher depth-levels of very deep thread traces while maintaining the high-level context or low-level context, respectively: In contrast to traditional vertical panning, it does *not* hide any depth-levels outside the viewport.

4.2.2 Adaptive Rendering

Based on a selection defined with the adaptive brushing, this section outlines how items are rendered to show (a) the selection with respect to focus and out-of-focus items as well as the focus point (in time) and (b) the icicle-plot structure in a space-efficient manner. Moreover, an automatic multiscale aggregation technique for activity views is described.

4.2.2.1 Rendering of Items and Selection

Focus Items We render all items in s_f with full opacity and shaded cushions. We use cushions to convey both the structure of the selected focus subset s_f and the position of the focus point $\mathbf{p}_f$ within s_f. Consider an item $p \in s_f$ whose icicle-plot bar is a rectangle R spanned by (x_l, y_t) and (x_r, y_b) (Fig. 4.18). If $x_l < x_f < x_r$, we cut R in two rectangles $R_l = (x_l, y_t); (x_f, y_b)$ and $R_r = (x_f, y_t); (x_r, y_b)$ and texture these with two *luminance* textures ψ_l and ψ_r, based on Eqn. 4.3. We define the corresponding luminance profile $\psi_i : [0,1]^2 \to \mathbb{R}^+$ as

$$\psi_i(x,y) = \left[\left(1-(1-f_x x)^{d_i}\right)\left(1-(1-2f_y y)^{d_i}\right)\right]^k, \tag{4.3}$$

i.e., the product of two exponential profiles ψ_i^x and ψ_i^y (Fig. 4.19). The values $f_x, f_y \in [0,1]$ control the position of the highlight.

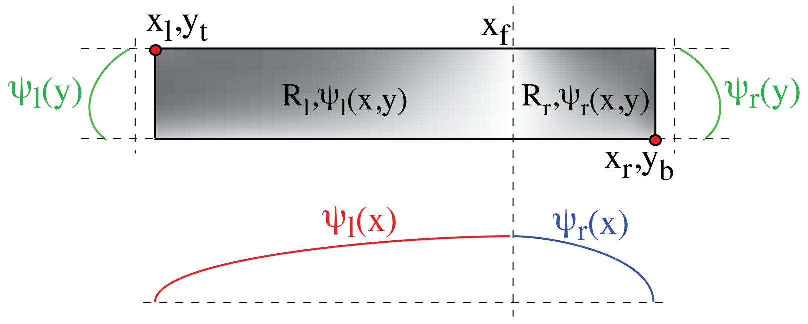


Figure 4.18: *Shaded cushions for activity view items.*

With $d_i = 1, f_x = 1, f_y = 0.8$, this yields a luminance profile that horizontally varies from dark ($x = x_l$) to fully bright ($x = x_f$) and then to dark again ($x = x_r$), and vertically shows the slightly convex profile in Fig. 4.19. If $x_f < x_l$ or $x_f > x_r$, we texture R as the R_r and R_l cases indicated above, respectively. As the user moves the mouse horizontally, the highlight at x_f moves along all items in s_f, like a 3D lighting which glides atop of the focus set.

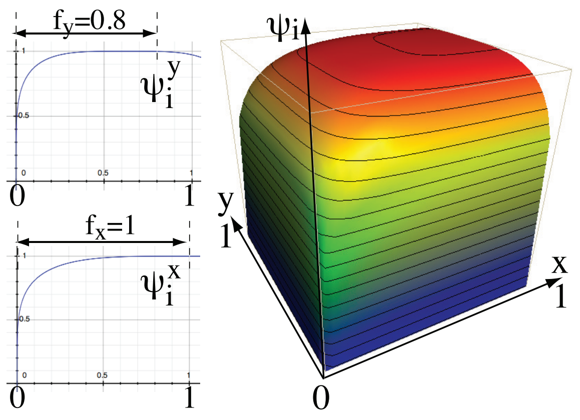


Figure 4.19: *Example luminance profile for the cushions.*

Items in s_f can further be color-mapped to show metrics of interest. For any such color mapping, we linearly decrease the saturation of colors in s_f upwards and downwards from o_f until the top-most and bottom-most items in s_f, respectively. As the user moves the mouse vertically within s_f, a saturation highlight follows the mouse to indicate the position of the focus item. Fig. 4.20 shows the rendering of an activity view with items in s_f colored in blue for illustration purposes. We see how items in the focus set change color close to the mouse. The horizontal shading gradient conveys a soft focus on items close to the mouse, and also emphasizes the icicle plot structure.

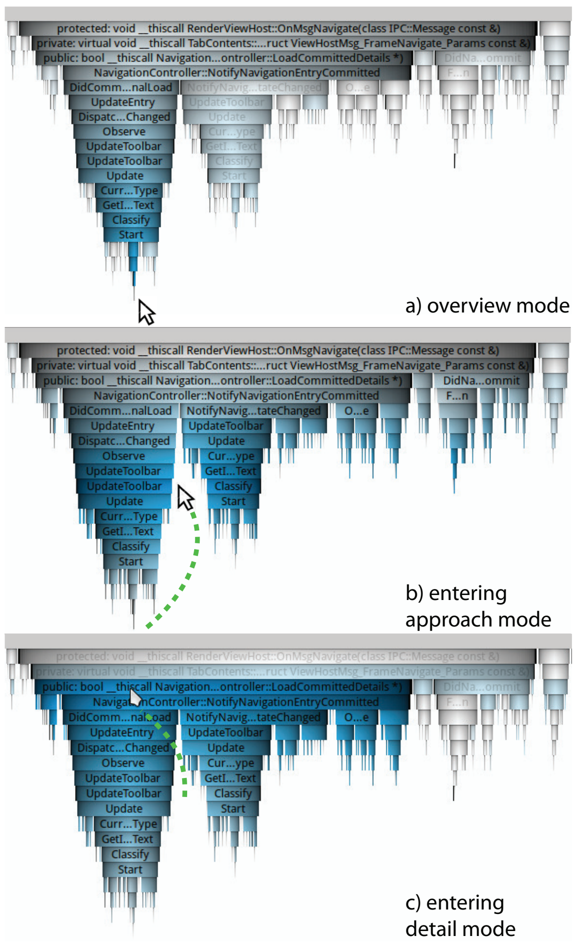

protected: void __thiscall RenderViewHost::OnMsgNavigate(class IPC::Message const &)
private: virtual void __thiscall TabContents::...ruct ViewHostMsg_FrameNavigate_Params const &)
public: bool __thiscall Navigation...ontroller::LoadCommittedDetails *)
NavigationController::NotifyNavigationEntryCommitted
DidComm...nalLoad
NotifyNavig...tateChanged
O...e
DidNa...ommit
F...n
UpdateEntry
Dispatc...Changed
Observe
UpdateToolbar
Update
Curr...Type
GetI...Text
Classify
Start
a) overview mode
b) entering approach mode
c) entering detail mode

Figure 4.20: *Transitions between interaction modes. Focus color is blue.*

Out-of-focus Items All items in $E \setminus s_f$ in the activity view are rendered with a high transparency and low saturation. For example, the right-most call stacks in Fig. 4.20, are desaturated *and* have a higher luminance (due to the transparent blending on a white background), which allows for visually separating s_f (blue) from its context (gray).

4.2.2.2 Multiscale Rendering for Activity Views

For large traces, especially in zoomed-out mode, many short-duration function calls would be too small to be rendered accurately as icicle plots in activity views (Fig. 4.21(a)). The bars would eventually have sub-pixel size, resulting in undesired high-frequency artifacts such as moiré effects. Moreover, rendering many such (sub-)pixel sized bars can have a

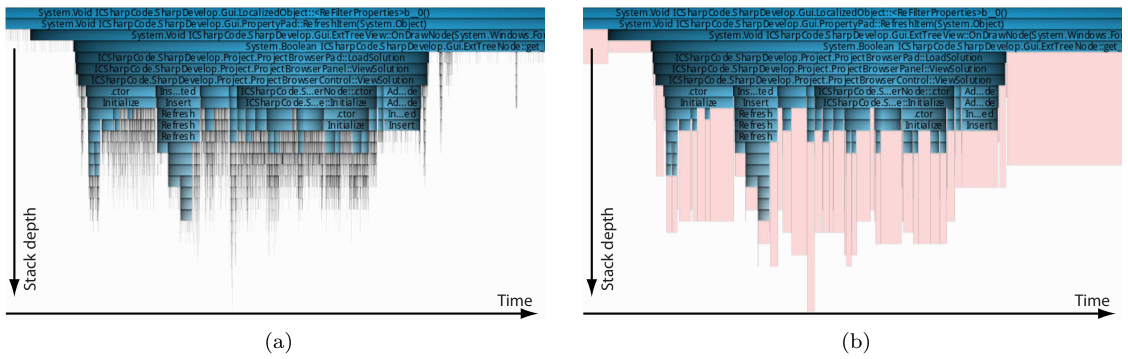


Figure 4.21: *(a) Activity view with many small bars causing a cluttered high-frequency image; bars are too small to show their labels. (b) Activity view with spatial aggregation; the high-frequency portions of the icicle plot are removed and replaced by aggregation bars representing their bounding box.*

significant performance impact without delivering added value. For this, the amount of bars to draw is reduced by aggregating smaller ones with less than a few pixels of horizontal screen extent. This upper-bounds the number of rendered shapes at a time, which allows for higher frame rates, and also makes the image less cluttered.

The aggregation stage first computes aggregation clusters A_i of very small adjacent bars (narrower than one pixel) rooted at calls f_i, i.e., a cluster contains all adjacent children $c_j, c_k \in C(f_i)$ of any call f_i with $\text{depth}(c_j) = \text{depth}(c_k)$ and $k = j + 1$. If a call is part of a cluster, the clustering does not process its children and skips to the next adjacent call: The bar representing a call's children can have only smaller horizontal extents and would thus be aggregated too. So, if a call is part of a cluster, its children are implicitly aggregated, too. The rendering of such clusters then happens either depth-wise or cluster-wise:

- Level-wise rendering: For each depth-level and for each adjacent calls in such level, a bounding box is computed, in which the box encloses all bars in the depth-level (Fig. 4.22(a)). Finally, each depth-level in the cluster is represented by a single bar. (Fig. 4.22(b)). These bars are rendered in a lighter color and without cushions to differentiate them from bars representing single function calls. In addition, with increasing stack depth, their color becomes even lighter.

- Cluster-wise rendering: For each cluster a bounding box is computed that encloses all calls c_j in a cluster and their children, i.e., the call stack $S(c_j)$ rooted at c_j. These bars (Fig. 4.22(c)) are rendered in a lighter slightly-transparent color and without cushions to differentiate them from bars representing single function calls.

Compared to several other techniques [66, 111, 197], this screen-space approach does not require any preprocessing of the trace data. As a result, the minimum screen size of the bars can be tuned interactively according to the users' needs.

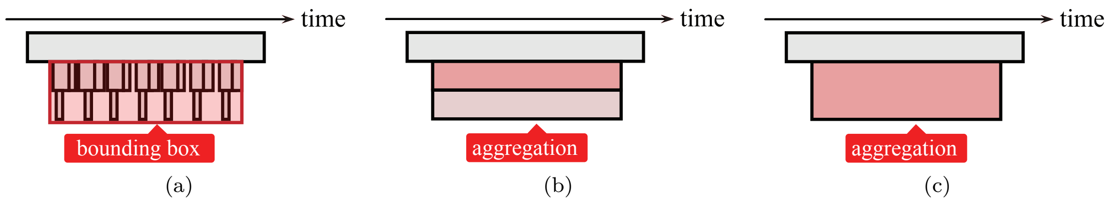


Figure 4.22: *Spatial aggregation strategy for icicle plots: (a) Bars which are too small to display, and their corresponding bounding box; (b) aggregation bars per depth-level; (c) aggregation bars per cluster.*

4.3 SyncTrace: Multiscale Visualization of Thread Interplay

This section is based on the following publication(s):

- Benjamin Karran, Jonas Trümper, and Jürgen Döllner. SyncTrace: Visual Thread-Interplay Analysis. In Proceedings (electronic) of the Working Conference on Software Visualization, pages 1-8. IEEE, 2013. [296]

Function boundary tracing generally records activity by means of entry/exit pairs per function call. As shown in the previous sections, visually analyzing such traces can help understanding sequential as well as concurrent software behavior. Information on the execution of threading-related system calls can further provide insights into the *interplay* of threads. For this, executions of not only non-blocking operations (e.g., user code) but also blocking operations (e.g., system calls for locking mutexes) are relevant. Tracing these two types of operations is a key strategy in program comprehension of concurrent programs [69]. For instance, such traces help explaining used concurrency patterns, thread responsibilities, and data sharing; locate performance bottlenecks as well as root causes of bugs caused by non-determinism; and incorrect synchronization. Moreover, they can be used to verify/falsify hypotheses about the (mis-)usage of blocking operations in a given program.

Not only due to massive data, analysis of such trace data and thereby of the interplay is hard, though. Analysis has to show much information per thread such as call relationships, timestamps, order of synchronization points as well as I/O operations, and related blocking system calls (including wait duration). Due to limited screen space and aspects of human cognition, it is hard to present and analyze this information for many threads in parallel.

This section presents a visualization technique for the interactive analysis of large traces captured from concurrent software systems. The visualization design enables the parallel analysis of many threads' runtime behavior and inter-relationships by a hybrid focus+context visualization approach: A combination of straight (focus) and bended (context) activity views depicts intra-thread call relationships, and edge bundles show inter-thread relationships caused by blocking operations such as synchronization. The technique further supports the exploratory nature of the underlying task; it addresses *visual* scalability by a multiscale design that allows for trace analysis on several levels of abstraction and by complementary interaction techniques.

4.3.1 Classification in a Task-Oriented Model

The visualization can be categorized in the task-oriented model of Maletic et al. as follows: The *task* is to help users analyze thread interplay captured in large-scale traces, specifically to (a) understand (concurrent) function calls per thread at several levels of detail; (b) analyze time stamps, order and durations of non-blocking function calls that explain intra-thread synchronization at several levels of detail; (c) explain wait durations caused by such calls; and (d) find concurrency patterns. The *audience* includes software engineers who want to understand execution aspects of large software systems. The visualization *targets* static software structure, information from a software trace (threads, function calls, call durations, and certain function parameters), and wait relationships between threads. These data are *represented* using a space-filling plot (for overviews), icicle plots (for the call structure), and a multiscale bundling metaphor (for the thread-to-thread wait relationships). Finally, the visualization *medium* consists of a standard screen with two linked views.

4.3.2 Extended Trace Model

This section extends the basic definition of a thread trace (cf. Section 2.1.4). Based on this model, the next section will describe the visualization design and related interaction techniques.

The definition of a function call is extended by the property 'objects' (O):

$$f = (F, t^s \in \mathbb{R}^+, t^e \in \mathbb{R}^+, p \in T, C, O \subseteq I). \tag{4.4}$$

with

$$I = \{I_j\}. \tag{4.5}$$

Further, four types of function calls can be distinguished:

1. S_w is the set of all calls to a synchronization function that waits for the objects in O.
2. S_r is the set of all calls to a synchronization function that releases the objects in O.
3. S_o is the set of all calls to a function that issues I/O requests, i.e., writes or reads a file.
4. S_c is the set of all calls to any other function.

For any f, with $S \notin S_c$, $O(f)$ is a set of objects on which the function call f operates. Hence, the set of all objects in a software trace $\mathbb{T}$ is called I. Such objects can be mutexes, semaphores, files, and other objects related to synchronization or I/O.

In addition to $\mathrm{dur}(f)$, the functions $\mathrm{dur}^x(f)$ are defined as follows:

$$\mathrm{dur}^x(f) = \begin{cases} \mathrm{dur}(f) & \text{if } f \in S_x \\ \sum_{c \in C(f)} \mathrm{dur}^x(c) & \text{otherwise} \end{cases} . \tag{4.6}$$

With x in $\{o, r, w\}$, $\mathrm{dur}^x(f)$ tell the duration that f spent doing synchronization ($\mathrm{dur}^w, \mathrm{dur}^r$) and I/O ($\mathrm{dur}^o$), respectively. Several threads can concurrently refer to an object used for such synchronization. Thus, there exists a set of potential correspondences $\mathrm{pcorr}(f_w) = \{f_r\}$ for any two $(f_w, f_r) \in S_w \times S_r$ if an object o exists that is released (or signaled) by a call f_r while f_w waited for o, i.e., if the following two expressions are true:

$$o \in O(f_w) \cap O(f_r) \text{ and} \tag{4.7}$$

$$\varnothing \neq [t^s(f_w), t^e(f_w)] \cap [t^s(f_r), t^e(f_r)] \tag{4.8}$$

A definitive *correspondence* $\mathrm{corr}(f_w, f_r)$ exists if f_r released (or signaled) o such that f_w can stop waiting. More precisely, a correspondence $\mathrm{corr}(f_w, f_r)$ exists if $\mathrm{pcorr}(f_w) \neq \varnothing$ and if the following holds for an $f_r \in \mathrm{pcorr}(f_w)$:

$$t^e(f_r) = max(\{t^e(f) | \, \mathrm{pcorr}(f_w)\}) \tag{4.9}$$

In other words, if we order the elements in $\mathrm{pcorr}(f_w)$ on their end moments t^e in ascending order, the last element is the respective release (or signaling) call f_r.

4.3.3 Visual Design

A multiscale representation of this trace data enables users, at a coarse scale, to identify synchronization patterns and outliers, as well as the concurrency patterns used in a software system such as farmer/worker, leader/followers, etc. [232]. At medium and fine scale, by contrast, detailed analysis of thread synchronization is possible.

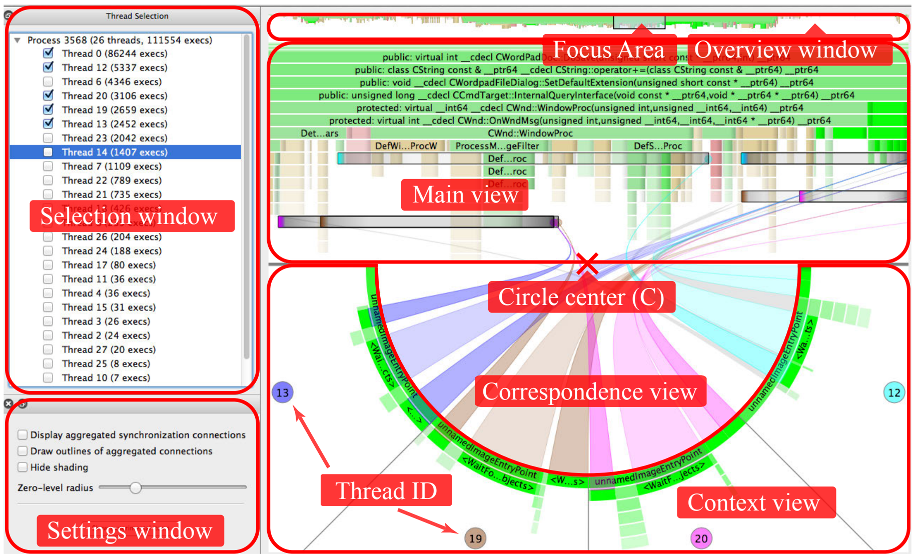
4.3.3.1 Exploration Workflow

Users start a top-down exploration by selecting a number of threads to visualize (selection window in Fig. 4.23). Next, they see an overview visualization of the trace data (top right in Fig. 4.23), which allows for identifying which are the main correspondences between which threads, which thread T_i waits mostly for which other thread T_j, and who waits longest. Subsequently, users can drill-down (top-down exploration) using zoom&pan to analyze subsets of the trace data in more detail using the main view and context view. In the correspondence view, they can now see where (stack level, function f) and when a thread waits for or releases objects by drawn overlays. On the waiting side, users can further investigate the amount of time spent waiting (and further details on demand), and which thread released objects waited for. On the release side, users can assess what happens while an object is being waited for. Visual attributes allow for discerning different wait types such as I/O cycles and synchronization. At any point in the analysis, users can switch to bottom-up exploration using the provided zoom facilities and directly jump to other subsets of the trace data using overview navigation.

4.3.3.2 Activity Views and Overviews

Both main view and context view visualize activity in the form of thread traces as icicle plots (based on TraceVis, cf. Section 4.1.2.3). The main view is located in the center (Fig. 4.23), with the context view below it. While the former shows only a single thread trace at a time and in more detail, the context view displays a variable number of thread traces. The righthand side of each bar in the icicle plot is augmented with a shaded texture (cf. Section 4.2.2). Through this, repetitive patterns show up as horizontal bands with equidistant vertical stripes.

For both the main view and the context view, a structure-based activity overview is drawn at the top of the main window. The activity overview for the context view shows an overview of one selected thread (see Section 4.3.4.2). Color coding in both the activity


Thread Selection
Focus Area
Overview window
Main view
Selection window
Circle center (C)
Correspondence view
Thread ID
Context view
Settings window
public: virtual int __cdecl CWordPad...
public: class CString const & __ptr64 __cdecl CString::operator+=(class CString const & __ptr64) __ptr64
public: void __cdecl CWordpadFileDialog::SetDefaultExtension(unsigned short const * __ptr64) __ptr64
public: unsigned long __cdecl CCmdTarget::InternalQueryInterface(void const * __ptr64,void * __ptr64) __ptr64
protected: virtual __int64 __cdecl CWnd::WindowProc(unsigned int,unsigned __int64,__int64) __ptr64
protected: virtual int __cdecl CWnd::OnWndMsg(unsigned int,unsigned __int64,__int64,__int64 * __ptr64) __ptr64
Det...ars
CWnd::WindowProc
DefWi...ProcW
ProcessM...geFilter
DefS...Proc
Def...roc
unnamedImageEntryPoint
<W...s>
<WaitFo...bjects>
13
12
19
20

Thread Selection

- Process 3568 (26 threads, 111554 execs)
  - [✓] Thread 0 (86244 execs)
  - [✓] Thread 12 (5337 execs)
  - [ ] Thread 6 (4346 execs)
  - [✓] Thread 20 (3106 execs)
  - [✓] Thread 19 (2659 execs)
  - [✓] Thread 13 (2452 execs)
  - [ ] Thread 23 (2042 execs)
  - [ ] Thread 14 (1407 execs)
  - [ ] Thread 7 (1109 execs)
  - [ ] Thread 22 (789 execs)
  - [ ] Thread 21 (735 execs)
  - [ ] Thread 18 (426 execs)
  - [ ] [illegible]
  - [ ] Thread 26 (204 execs)
  - [ ] Thread 24 (188 execs)
  - [ ] Thread 17 (80 execs)
  - [ ] Thread 11 (36 execs)
  - [ ] Thread 4 (36 execs)
  - [ ] Thread 15 (31 execs)
  - [ ] Thread 3 (26 execs)
  - [ ] Thread 2 (24 execs)
  - [ ] Thread 27 (20 execs)
  - [ ] Thread 25 (8 execs)
  - [ ] Thread 10 (7 execs)

- [ ] Display aggregated synchronization connections
- [ ] Draw outlines of aggregated connections
- [ ] Hide shading

Zero-level radius

Figure 4.23: *Overview of the* SyncTrace *main window.*

views as well as the activity overviews allows for finding high-activity ranges in thread traces with respect to synchronization and I/O, which is detailed later (Section 4.3.5).

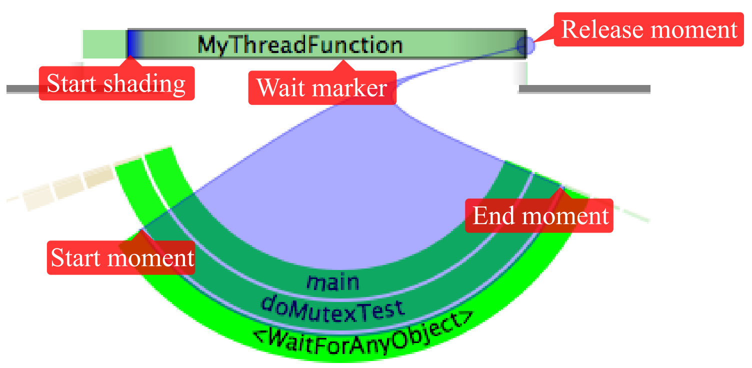
In the context view, all thread traces together form a semicircle in which each thread trace is assigned an equal section by default. To indicate which thread trace is visualized in a section, the section is labeled with the thread ID as colored circle, each thread T having one color $\mathrm{col}(T)$ out of a predefined set of colors. Although such assignment of colors is not bijective, the assignment is constrained such that no two adjacent threads can have the same color. By this, repetitions of thread colors can only occur for threads that are geometrically distant in terms of angle on the semicircle.

In contrast to the main view, the icicle plots are bended. The resulting circle segments provide only a relatively small screen space per thread. In a traditional icicle plot, this would make it particularly hard to read labels of low-level, short-duration function calls, which are the smallest items there. To alleviate this, the context view uses a layout similar to the Sunburst technique: All thread traces in the context view share a common circle center C and radius Z for depth 0 (Fig. 4.23). Instead of mapping depth to the y-coordinate, it is mapped to the radius of a circle centered in C. Depth 0 is assigned the zero-level radius Z, and subsequent depth levels are assigned increasing radii such that the icicle plots grow from the center of the circle to the outside. The start and end coordinates of each bar are mapped to an angle. Deeper stack levels are thus assigned more space for the same time span than higher stack levels. As a result, labels of bars representing low-level, short-duration function calls are better readable as more characters can be shown.

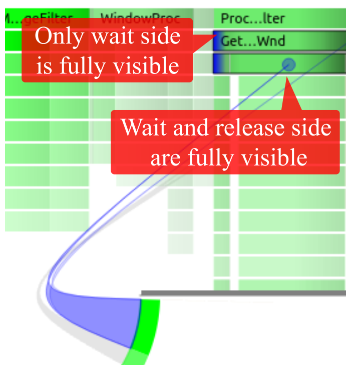
Last, the correspondence view surrounds the circle center. It is the key element of this visualization for showing relationships between thread traces, which is described next.

4.3.3.3 Correspondence Visualization

In the correspondence view, a *correspondence curve* connects two bars of two thread traces to represent a correspondence between the threads. As function calls can generally wait for or release more than one object, a correspondence curve is rendered if wait and release call share at least one object. Wait calls on multiple objects may be classified into those that wait for all objects and those that wait for any. The first kind returns if all objects are released, and the latter kind returns as soon as a single object is released. To differentiate



(a)


Only wait side is fully visible
Proc...lter
Get...Wnd
Wait and release side are fully visible

(b)

Figure 4.24: *(a) Detailed view of a single correspondence curve; (b) Representations of partially and fully visible release markers.*

them, the correspondence view renders a distinct function name for each. Note that the following text often refers only to a single object instead of a set of objects to simplify the explanation.

A correspondence $corr(f_w, f_r)$ is represented by a shape with three anchor points (Fig. 4.24(a)). Two of them denote the start and end waiting moments of the waiting function call f_w, and the third denotes the release moment in the releasing function call f_r; a circle distinguishes the release moment. On its way from one thread trace to the other, a correspondence curve is bended towards the circle center. Together, these result in an asymmetric curve shape that allows for identifying who waits for and who releases an object. Curves are semi-transparent to facilitate following individual curves in case of overdraw.

Additionally, a wait marker (see Fig. 4.24(a)) is rendered as a semi-transparent overlay for the bar representation of f_r to show the waiting time range $[t^s(f_w), t^e(f_w)]$, i.e., the period of time that f_w spent waiting. This enables us to examine what happens in the releasing thread trace (T_r) while f_w in thread T_w is waiting for an object owned by T_r.

Since wait markers of different thread traces may overlap, it is not clear where a wait marker starts. To overcome this ambiguity, the correspondence view renders a *start shading* (see Fig. 4.24(a)) in the color of the waiting thread (col(T_w)), which aids in keeping track of T_w while exploring the area surrounding the start of the wait marker. This is especially helpful when the release moment is outside of the viewport and, thus, several release markers overlap at the edge of the viewport.

To ensure that users only see correspondence curves that are *relevant* to the current view configuration, these are filtered as follows: The correspondence view generally renders correspondence curves only if at least a part of the respective wait call is visible in the viewport. This includes correspondences whose release moment is partially/completely outside the viewport. Those correspondences are rendered more transparent and the circle around the release moment is omitted. So, it can happen that a wait marker is still visible while the release marker is outside of the visible time range. For instance, we can observe this circumstance in Fig. 4.24(b). To still see which correspondence belongs to which wait marker, the end of the correspondence curve is kept in the same stack level as the wait marker. Here, the missing circle indicates, that the end of the correspondence doesn't mark the release time. If the wait marker also starts outside the viewport, users can notice this by the missing start-shading (cf. Fig. 4.24(a)).

4.3.3.4 Multiscale Visualization

Both the main view and the context view aggregate the icicle plots per thread trace T_m as described in Section 4.2.2.2. This strategy significantly reduces visual clutter in an automatic manner and is thus especially helpful when showing many traces in zoomed out views, as in this case.

To further avoid overloading the visualization, it is necessary to aggregate correspondence curves, too. The curve aggregation bases on the union $\mathbb{A}$ of all aggregation clusters A_i that were previously computed for aggregating the icicle plots. For every T_m the correspondence view then computes

$$\mathrm{rel}_{\mathbb{A}}(T_m) = \{t^s(f_r) | f_r \in T_m \wedge f_w \in \mathbb{A} \wedge \mathrm{corr}(f_w, f_r)\}. \tag{4.10}$$

Here, $\mathrm{rel}_{\mathbb{A}}(T_m)$ is the set of all release moments for which a correspondence curve would be drawn to T_m from the function calls in $\mathbb{A}$. Next, for each thread T_m, a compound correspondence-curve is rendered (Fig. 4.25). It consists of a filled path from the aggregation start moment to its end moment, the maximum release moment, and the minimum release moment back to the start.

The appearance of such path's ends is, in contrast to non-aggregated curves, asymmetric on purpose to keep such curves' wait side distinguishable from their release side: On the release side of the path, the curve is bended into the direction of the circle center, thereby forming a concave shape. These aggregated paths typically occupy a larger area screen-space area than their non-aggregated counterparts. Thus, to prevent them from drawing too much visual attention, the correspondence view renders them below non-aggregated correspondence curves.

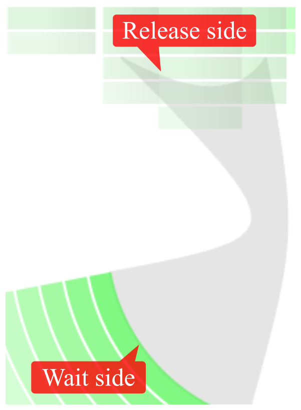


Figure 4.25: *Aggregated correspondence curves: The release side is bended into the direction of the circle center.*

4.3.4 Interaction Techniques

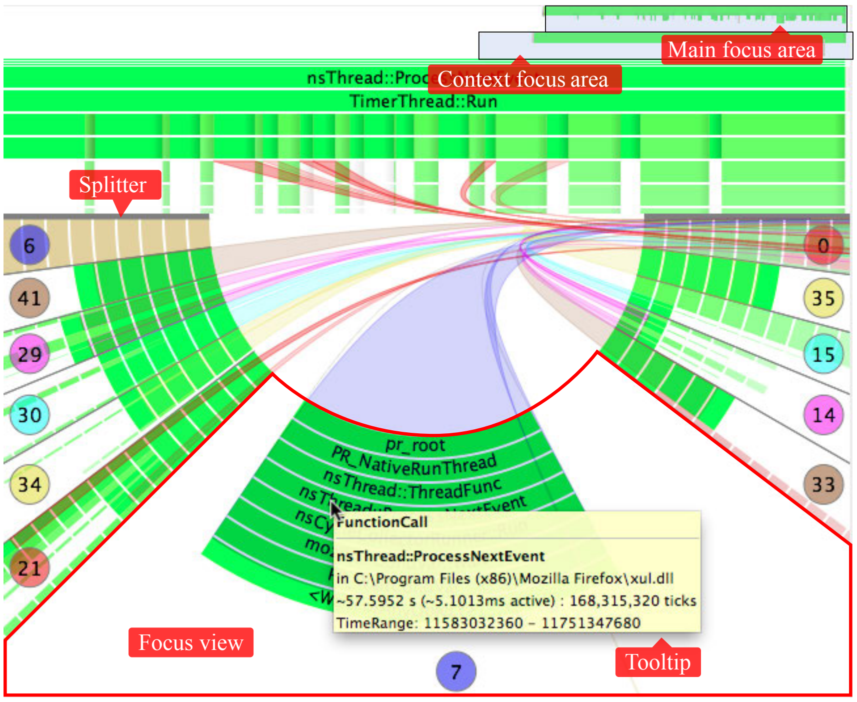
4.3.4.1 Basics

After loading a software trace, thread traces can be selected from the selection window (see Fig. 4.23). Every newly selected thread trace is then displayed in the main view and any previously displayed thread trace is moved to the context view.

Both, the main and the context view are equipped with pan and zoom, so the viewport for every thread trace can be set individually. In addition, vertical drag triggers the stack-folding technique, which incrementally reduces the vertical height of successive stack levels (Section 4.2.1.2) by setting $\rho_l = 0$ and modulating only ρ_u in Eqn. 4.1 as magnification function. This technique is especially helpful in the given use case in which efficient use of available screen space is crucial.

4.3.4.2 Configurable View Layout

To allow for an on-demand partitioning of the screen space, main view and context view are separated vertically by a movable splitter. Also the zero-level radius is adjustable to either increase the screen space for showing correspondences or to show more stack levels in the context view. Further, a thread trace T_{cf} in the context view may be *focused* to



FunctionCall
nsThread::ProcessNextEvent
in C:\Program Files (x86)\Mozilla Firefox\xul.dll
~57.5952 s (~5.1013ms active) : 168,315,320 ticks
TimeRange: 11583032360 – 11751347680

Figure 4.26: *'Focusing' a thread-trace in the context view: The section of one selected thread trace on the semicircle is temporarily increased for more detailed analysis.*

give it more space than the others (Fig. 4.26). For the focused T_{cf} an overview version is rendered in the overview bar and the focus area is indicated like for the main view.

To reorder thread traces, users can drag&drop them on top of each other, which swaps the two affected thread traces (Fig. 4.27). By this, thread traces can be reordered from context view to main view or within the context view to different sections of the semicircle. In case users want to quickly filter out several thread traces, they can double click on the thread-ID label to hide the corresponding thread trace.

4.3.4.3 Detail on Demand

When users hover over items in the activity view, a tooltip with detailed information is shown: For individual bars, this shows information concerning the represented function call; for aggregated bars, the respective original bar below the cursor first needs to be

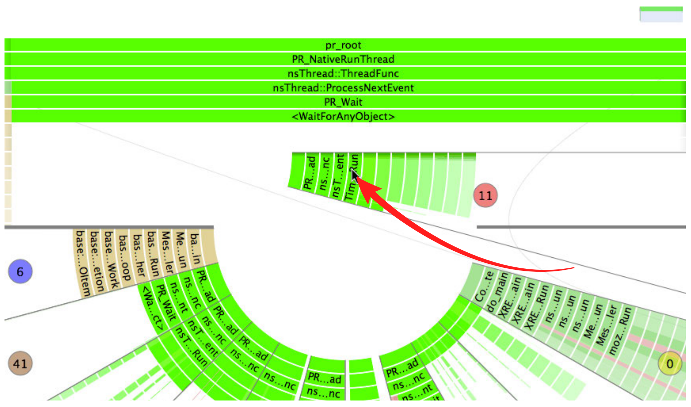
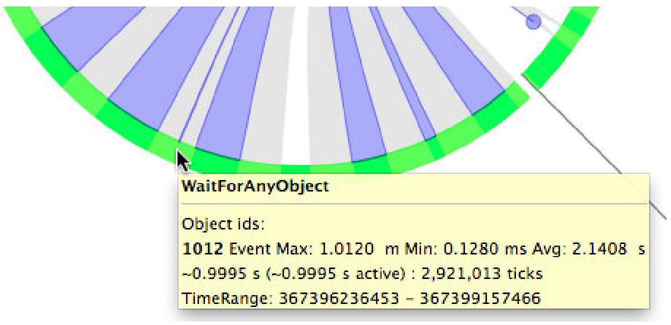


Figure 4.27: *Re-ordering thread traces using drag&drop.*



```
WaitForAnyObject
Object ids:
1012 Event Max: 1.0120  m Min: 0.1280 ms Avg: 2.1408  s
~0.9995 s (~0.9995 s active) : 2,921,013 ticks
TimeRange: 367396236453 – 367399157466
```

Figure 4.28: *The tooltip reveals which function calls are contained in the aggregation. In addition, the associated correspondences of the hovered call are displayed.*

estimated since the aggregated bars can have sub-pixel size. In addition, whenever the hovered bar represents a wait call, the associated correspondence is highlighted (Fig. 4.28).

Moreover, aggregated correspondence curves are hidden by default. Hovering anywhere in the section of a thread trace T_h then shows aggregated correspondences for T_h. More precisely, only correspondences for which function calls $f_i \in T_h$ wait for objects, i.e., $f_i \in T_h \cap S_w$, are visible. If, in contrast, *all* aggregated correspondences would be rendered (opaque and without borders) for many selected thread traces, the resulting image would be unreadable due to visual clutter.

4.3.5 Color Mapping

In the following, a set of attribute mappings is described that has proven suitable for the subsequent application example.

4.3.5.1 Activity Views

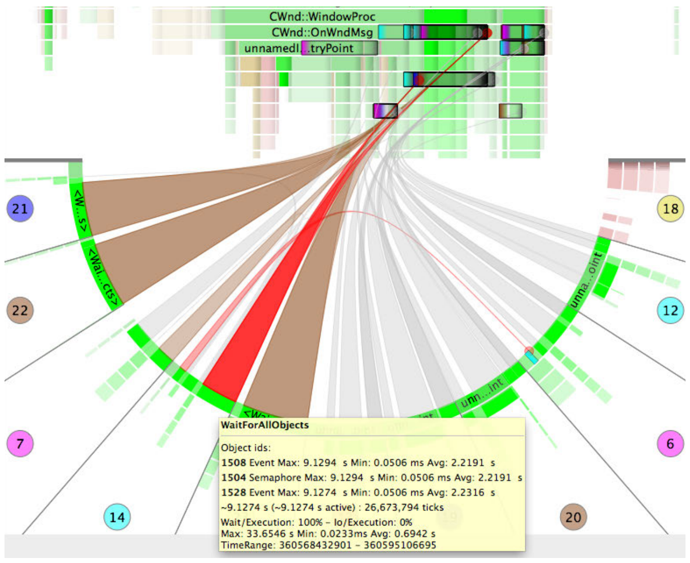
All function calls in the activity views are rendered in one of three hues (green, red and brown). A function call is colored brown if neither it nor any of its descendants is doing I/O or waiting for an object. In contrast, a call f which performs I/O (synchronization) operations or has any descendants doing so, is colored depending on the ratio $w_r(f) = \frac{\mathrm{dur}^o(f)}{\mathrm{dur}^w(f)}$: For $w_r(f) < 0.5$, it is colored red, and green otherwise.

The intensity of green/red is proportional to $\frac{\mathrm{dur}^w(f)}{\mathrm{dur}(f)}$, $\frac{\mathrm{dur}^o(f)}{\mathrm{dur}(f)}$ respectively. That is, calls that exclusively wait or perform I/O are rendered with the highest intensity.

4.3.5.2 Correspondence Curves

In addition to the shapes of correspondence curves, which visually connect corresponding threads, two attribute mappings emphasize specifics of these correspondences: The first mode, *thread color mode*, colors correspondence curves according to the thread color $\mathrm{col}(T_w)$, where T_w is the waiting thread trace ($f_w \in T_w$; see, e.g., Fig. 4.23). This helps distinguishing curves originating from different thread traces, and determining who waits for whom and who releases locks for whom. The second mode, *object color mode*, highlights correspondence curves based on shared synchronization objects which are waited for in *multiple* threads. This emphasizes and points to correspondences between such two waiting threads (Fig. 4.29).

In this mode, an object's correspondences are grayed out if only a single thread waited for that object during its life time. In contrast to aggregations, however, they still have an



```
WaitForAllObjects

Object ids:
1508 Event Max: 9.1294  s Min: 0.0506 ms Avg: 2.2191  s
1504 Semaphore Max: 9.1294  s Min: 0.0506 ms Avg: 2.2191  s
1528 Event Max: 9.1274  s Min: 0.0506 ms Avg: 2.2316  s
~9.1274 s (~9.1274 s active) : 26,673,794 ticks
Wait/Execution: 100% - Io/Execution: 0%
Max: 33.6546 s Min: 0.0233ms Avg: 0.6942 s
TimeRange: 360568432901 - 360595106695
```

Figure 4.29: *In object color mode, hovering over a wait call shows correspondences that refer to the same objects.*

outline and are semi-transparent. Upon hovering over a wait call for a *shared* object, the following color mapping is applied to the respective correspondences:

A correspondence is colored

1. red, if it belongs to a wait call that operates on the same set of objects as the hovered one,
2. brown, if its wait call shares at least one object with the hovered call,
3. and gray otherwise

4.3.6 Application in Industry-Scale Software Systems

This section shows the usage of SyncTrace with the help of a software trace captured from the web-browser *Firefox*[6] (version 12.0.1, approx. 6.7 million SLOC[7], thereof 3.8 million SLOC in C and C++) while it loads and renders a page of search results from Google. After removing the most often called 5190 functions (cf Section 2.1.4), the resulting software trace consists of approximately 14 million function calls in 43 thread traces. For all threads, about 2700 synchronization and I/O calls were recorded.

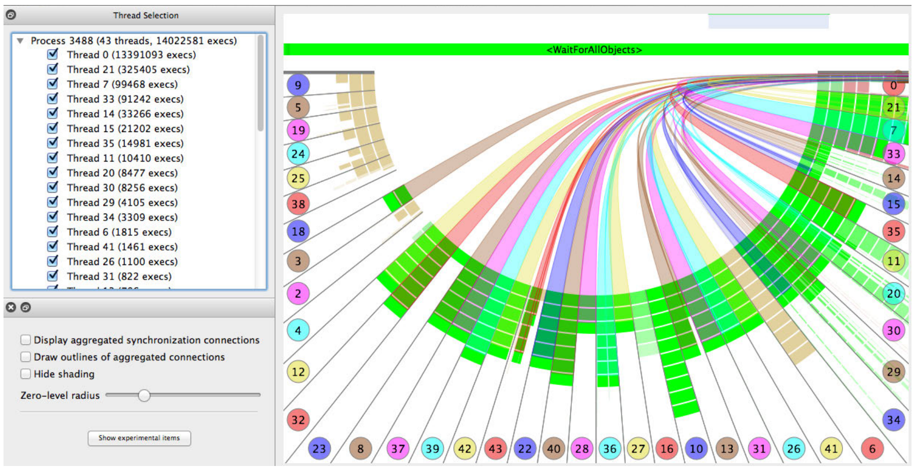
After loading the trace and displaying all thread traces at once, we see an overview representation of the correspondences (Fig. 4.30). Apparently, some threads don't wait for synchronization primitives at all (threads 9, 5, 19, 24, etc.), whereas many of the other threads are waiting for a long and continuous period (threads 3, 12, 32, ..., 35, 7). We can deduce this because only one wide correspondence curve is rendered for the latter thread traces. Further, the visualization shows that all major wait calls are released by thread 0. This hints to a farmer/worker pattern, with thread 0 being the farmer. If so, we can tell by the long continuous wait calls that many workers are mostly idle, and, hence, that too many workers might have been spawned initially.

[6]`http://www.mozilla.org/firefox/`, last accessed 07/11/2013
[7]`http://www.ohloh.net/p/firefox/analyses/latest/`, last accessed 07/11/2013

Since some of the threads expose interrupted wait periods, they actually did some work in between. If there is in fact a farmer/worker pattern present and all waiting threads are workers, this indicates a workload imbalance among them.

To verify the farmer/worker hypothesis we examine the stack trace of those threads. For this, we drag one thread after another into the focus view to examine its activity and function names in more detail. Some of the threads execute *nsThread::ThreadFunc*, which in turn calls *nsThread::processNextEvent*. So, there are in fact threads that just do work item processing, and which have no other dependencies.

However, we found other kinds of threads. One of them drew our attention, since it executed *mozilla::HangMonitor::ThreadMain*, but was waiting for thread 0 to release it during its entire lifetime (Fig. 4.31). The function's name indicated that it is responsible for observing the rest of the application and to come into action if the software was hung. However, if the application would actually be hung, nobody could notify (i.e., wake up) the thread, so it is crucial that it wakes up periodically by itself to check whether the application is still responsive. By hovering over the waiting function call, the tooltip revealed that this thread is not waiting for a timer, but a semaphore. This rules out the possibility that the trace is too short to capture the thread while waking up, i.e., the thread


Thread Selection

- Process 3488 (43 threads, 14022581 execs)
  - ☑ Thread 0 (13391093 execs)
  - ☑ Thread 21 (325405 execs)
  - ☑ Thread 7 (99468 execs)
  - ☑ Thread 33 (91242 execs)
  - ☑ Thread 14 (33266 execs)
  - ☑ Thread 15 (21202 execs)
  - ☑ Thread 35 (14981 execs)
  - ☑ Thread 11 (10410 execs)
  - ☑ Thread 20 (8477 execs)
  - ☑ Thread 30 (8256 execs)
  - ☑ Thread 29 (4105 execs)
  - ☑ Thread 34 (3309 execs)
  - ☑ Thread 6 (1815 execs)
  - ☑ Thread 41 (1461 execs)
  - ☑ Thread 26 (1100 execs)
  - ☑ Thread 31 (822 execs)

☐ Display aggregated synchronization connections
☐ Draw outlines of aggregated connections
☐ Hide shading
Zero-level radius

Figure 4.30: *Initial overview of a software trace after loading all thread traces.*

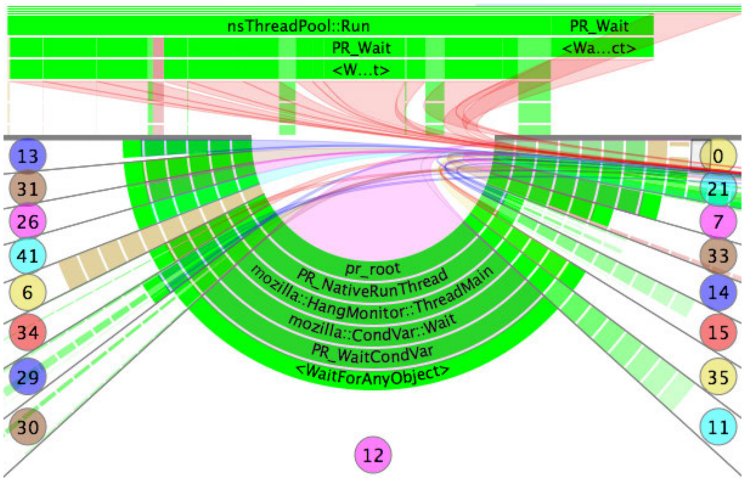
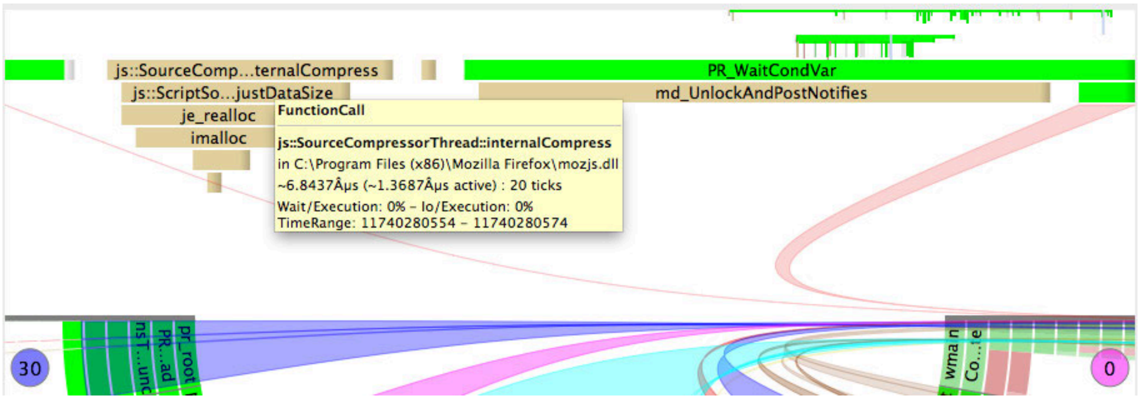


Figure 4.31: *Defunct hang monitor in* Firefox*: The hang monitor waits during its entire lifetime for an object to be released.*



FunctionCall

js::SourceCompressorThread::internalCompress
in C:\Program Files (x86)\Mozilla Firefox\mozjs.dll
~6.8437Âµs (~1.3687Âµs active) : 20 ticks
Wait/Execution: 0% - Io/Execution: 0%
TimeRange: 11740280554 - 11740280574

Figure 4.32: *A (mostly idle) worker thread responsible for compressing Javascript.*

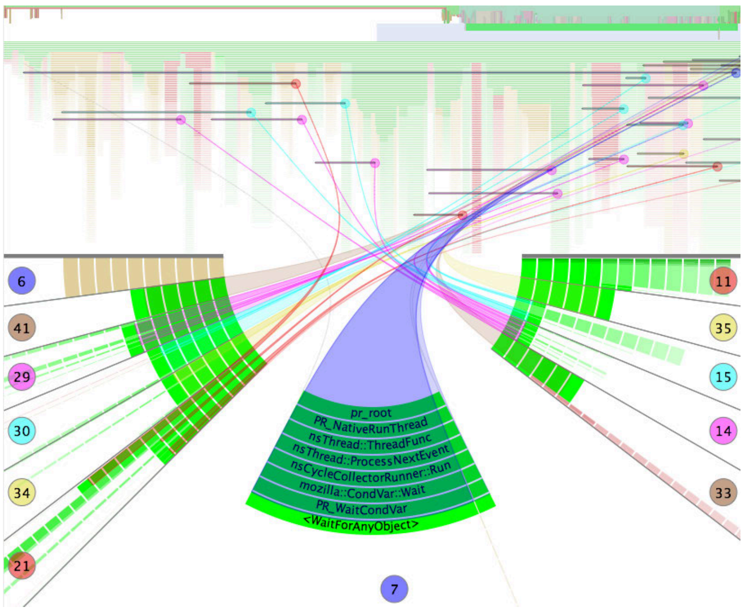


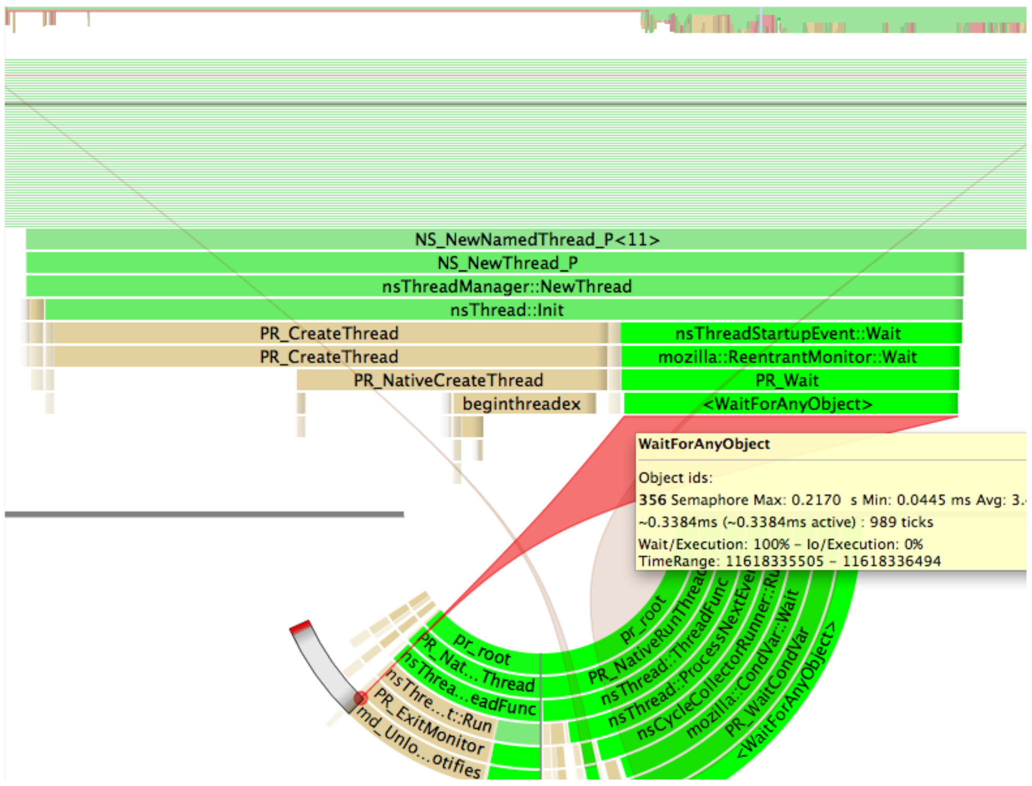
Figure 4.33: *Call stack of thread 0 for an overview. Release calls (colored circles in main view) are made on varying stack levels.*

can not wake up by itself. Finally, by examining source code of the *HangMonitor* class, we found that the wake up timeout was set to infinity and, thus, the hang monitor was effectively disabled.

Another thread (15) always called *js::SourceCompressorThread::internalCompress* after waiting for thread 0. Obviously, this is a dedicated Javascript compression thread. Since it always waited for thread 0, it seems to process compression tasks generated by this thread. We expected a compression task not to do any synchronization or I/O. This hypothesis was supported by the brown shading of the compression functions, which would have been colored green or red otherwise.

Since we did not load a Javascript-rich website, the compressor thread was mostly unoccupied and in many cases the compression tasks took less time than notifying thread 0 of the result (Fig. 4.32). This mostly idle thread suggests a waste of operating system resources. In addition, its very short task processing times may be too short to actually speed up the application or might even slow it down compared to a single-threaded version.

Next, we focused on thread 0 since all of the other threads' long wait calls are released by it. To get an overview, we dragged thread 0 to the main view and folded all stack



WaitForAnyObject

Object ids:
356 Semaphore Max: 0.2170 s Min: 0.0445 ms Avg: 3.
~0.3384ms (~0.3384ms active) : 989 ticks
Wait/Execution: 100% – Io/Execution: 0%
TimeRange: 11618335505 – 11618336494

Figure 4.34: *Thread creation: The creator has to wait for the synchronous call to return.*

levels. If there were a loop, for example, we expected to see that releases would be made in the same stack levels, resulting in visible patterns. However, the resulting image did not show such patterns (Fig. 4.33). This, in contrast to our hypothesis, suggests that thread 0 processes many separate (i.e., independent) tasks throughout its lifetime. So, the distribution of release moments, together with the variable stack depth of releases are indicators for the behavioral complexity of this thread.

Since there are no visible wait correspondences going from thread 0 to another thread, thread 0 spent the majority of its lifetime computing instead of waiting. This is supported by the fact that only a few sections of its activity are colored green and if they are, they are only slightly saturated. Hovering over the thread's start function reveals that only 0.5% of its lifetime is spent waiting.

We recall that function calls that are influenced by blocking calls are highlighted in red or green, and the intensity indicates the amount of blocking. The green color of *do_main* signals that its descendants had to wait somewhere for a synchronization object.

To find out which subsequences of thread 0 are influenced by blocking calls (calls that could cause a scheduler to suspend the thread for a while) we follow the green function bars (wait calls) downwards (increasing stack level). By panning and zooming to the region of interest, aggregations reveal their content and previously aggregated correspondences are rendered individually. This enables us to follow single correspondences. For example, we can see in Fig. 4.34 that creating a new thread via *nsThreadManager::NewThread* implied waiting for the new thread to leave a monitor (synchronization).

In addition to synchronization calls, I/O operations may block a thread. To find out more about the software artifacts that perform I/O, we follow the red colored functions bars as we did for the green ones before. We find that much of the I/O operations are performed by an SQLite database layer, where, in the most cases, I/O accounts only for

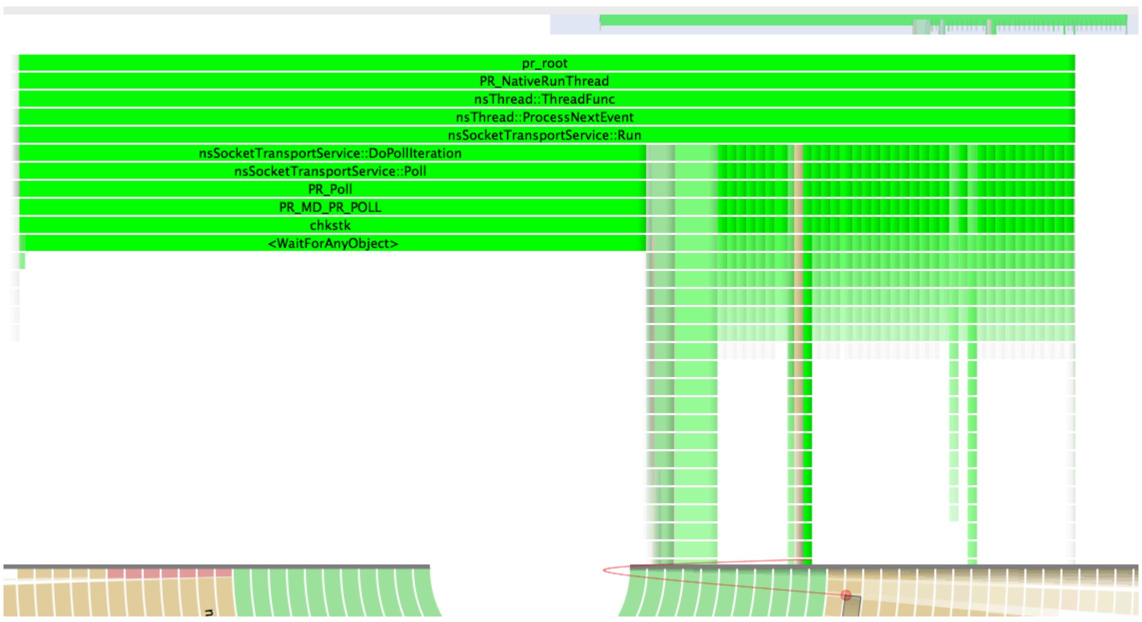

pr_root
PR_NativeRunThread
nsThread::ThreadFunc
nsThread::ProcessNextEvent
nsSocketTransportService::Run
nsSocketTransportService::DoPollIteration
nsSocketTransportService::Poll
PR_Poll
PR_MD_PR_POLL
chkstk
<WaitForAnyObject>

Figure 4.35: *Repetitive pattern of polling calls, strongly variable call duration.*

a small fraction of the time spent for such database operations. However, we found an intensely red-colored aggregated activity that took approx 500ms to finish. The call stack reveals that it belongs to *nsDocShell::AddURIVisit*. Here, the tooltip further revealed that *AddURIVisit* takes around 0.1ms on average, only a tiny fraction of this 500ms outlier. Investigating other occurrences of *AddURIVisit* reveals that it normally performs only short-lived I/O operations or none at all. Consequently, calling *AddURIVisit* is sometimes a costly operation in terms of runtime, although, in the most cases, it is not. Since this call happened in the user interface thread, we suspect it was a cause for rendering *Firefox* temporarily unresponsive.

By dragging thread 10 to the main view we see a repetitive call pattern (Fig. 4.35). Zooming in to read the function names, shows that this thread is polling every second. The stack pattern resembles the one to the left of the visible subsequence of the trace. However, while a single poll iteration took a minute, the frequency has now increased to a poll every second. This indicates that the poll interval was reduced in between.

To reveal object sharing among threads, we switch into the object color mode (cf. Section 4.3.5). Fig. 4.36 shows that all connections are greyed out, which indicates that *Firefox* uses separate synchronization objects for every thread. This is in contrast to, for example Microsoft WordPad, which makes excessive use of shared objects (see Fig. 4.29 on page 68).

The visual analysis outlined above took about 15 minutes. We note the added value of the stack folding technique, which enabled us to quickly gain an overview in which levels in the call stack synchronization objects are released. The start shading for function bars enabled finding small-scale repetitive patterns which we would have missed otherwise. Despite some crossings in the correspondence curves, we were able to discern the threads' individual correspondences with the help of our interaction techniques, highlighting, and the provided attribute mappings.

4.3.7 Discussion

Generality While the SyncTrace approach is demonstrated on large software traces, it can be used to visualize any given set of hierarchical event sequences with correlations.

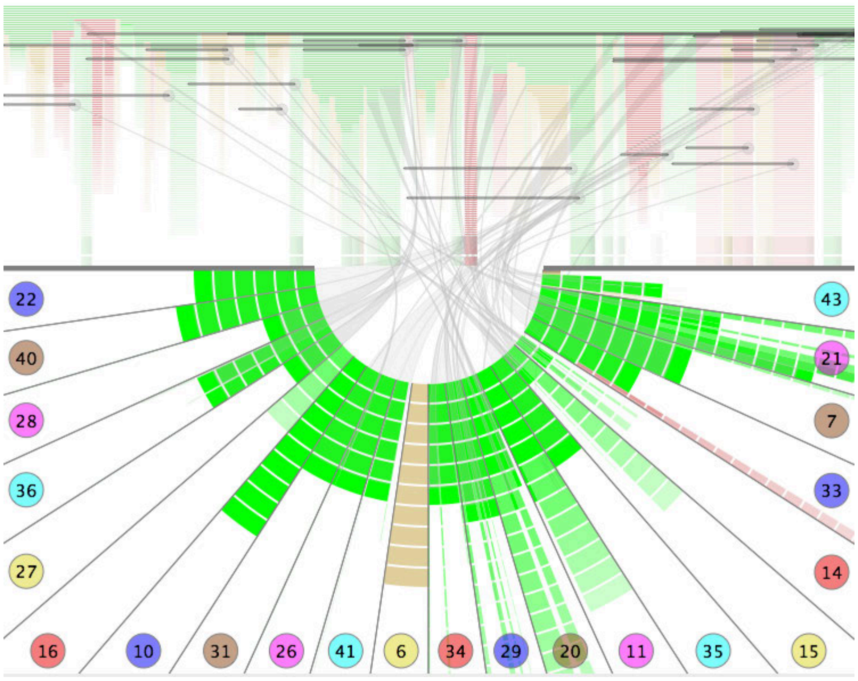


Figure 4.36: *Object color mode in the* Firefox *software trace: Since all connections are grayed out every thread has its own synchronization objects to wait on.*

The correlations are not restricted to leaf nodes, i.e., they can be one-to-one matches on any hierarchy level.

Visual Scalability The enhanced spatial layout enables the parallel depiction of many threads' activities and inter-correspondences between those threads. SyncTrace reduces visual clutter by specific interaction and multiscale aggregation techniques. This lets users visually analyze hundreds of thousands of calls and correspondences in multiple thread traces in an interactive manner.

Ease of Use The zoom&pan interaction techniques enable users to freely adjust the shown subsequences and level of detail. Drag&drop can be used to quickly re-define the focused thread as well as to reposition threads in the context view. Moreover, removing uninteresting threads from the current analysis set is as easy as double clicking their visual representation.

Flexibility By these inputs, users quickly configure the views as needed. The correspondence aggregation automatically adjusts the level of detail to the current zoom level. The provided color mappings can be adapted to highlight certain aspects of the trace data and thread interplay such as the shown object sharing.

Limitations With the visual design being more flexible than Code Flows (SyncTrace can show correspondences between any two trees) and scalable to concurrently show many trees, it as well has its limits in terms of very large software traces (many function calls) or massively multi-threaded software traces.

Due to the way wait correspondences are extracted, the application examples (Section 4.3.6) are limited to analyzing blocking system calls. In contrast, analysis of library calls on already-released objects, which are non-blocking, can thus not be shown here.

Benefits SyncTrace is a visualization technique for the analysis of dependencies between threads of execution in concurrent software systems. It combines common straight and bended icicle plots in a hybrid juxtaposed view to implement a focus+context approach. By this, SyncTrace enables the depiction of more threads of execution in the same screen space than existing techniques. Moreover, by the juxtaposition, SyncTrace can use the center of the viewport to depict relationships between any two threads of execution as multiscale edge bundles. It uses attribute mapping to colors and shape of the edge bundles to encode important runtime meta-data. The technique is demonstrated on large-scale software traces from industry-scale software systems.

Chapter 5

Analysis of Correlated Behavior-Structure and Behavior-Behavior

Correlating and comparing massive high-dimensional data can yield useful insights, e.g., into complex cause-effect relationships. Due to several reasons, though, it can be a challenging task. Besides the size of the data and its dimensionality, the basic task of correlating two data sets can already be hard. It is especially hard if the means used to correlate the data requires users to perform a significant amount of context switches: The switches generally occupy (a considerable fraction of) a user's short-term memory, and require mental effort and time [60]. The switches further interfere with the task of correlating data that also requires the users' short-term memory for comparing items from the data sets. This is one of the main reasons why traditional side-by-side list representations do not scale well to massive and/or high-dimensional data [46].

Likewise, correlating software traces with other software artifacts is a particularly difficult task: Traces are by far one of the largest and most complex artifacts related to software systems. Despite this, there are a several use cases for correlating traces. Among others, understanding relationships between traces and system structure can be helpful to

- conclude from low-level function calls (captured as a trace) to the high-level behavior of system components and vice versa: For instance, to relate features to code [109] and to identify execution phases [68];
- identify starting points for the exploration of a trace by filtering it based on the relationships to structural entities [171].

Correlating traces with each other is useful for

- filtering a trace by one or more other traces, e.g., to identify feature-relevant and feature-irrelevant function calls [12];
- find semantic duplication by identifying overlapping control-flow paths [137];
- understand how different execution contexts influence behavior [218, 219].

Moreover, comparing traces with code changes can, e.g., support finding correlations between changes to the code base and changes in behavior.

This chapter presents two techniques to ease the task of correlating traces in two respects: ViewFusion (Section 5.1) is a visualization method for integrating a structure view (e.g., to show a system structure) and an activity view (e.g., to show a trace) within a single screen-space area to facilitate correlating data shown in the two distinct views. TraceDiff, the second visualization technique (Section 5.2), addresses correlating hierarchical event sequences with each other: It is implemented as a tool that supports comparing executions recorded as traces of similar behavior.

5.1 ViewFusion: Visualization of Correlated Behavior and Structure

This section is based on the following publication(s):

- Jonas Trümper, Alexandru Telea, and Jürgen Döllner. ViewFusion: Correlating Structure and Activity Views for Execution Traces. In Proceedings of the Conference on Theory and Practice of Computer Graphics, pages 45–52. Eurographics, 2012 [303]

Structure views are used to display data such as organization layers, software system containment relations, catalogs, and directory structures. Treemaps [236], icicle plots, and node-link graph layouts [19] are effective tools for creating structure views that support users in tasks such as getting overviews of a given dataset, comparing substructures of interest, and correlating the distribution of metrics of interest with the structure. Activity views are just as important: They convey insight into the dynamics of a process, such as the evolution in time of several metrics' values, an event sequence, or the history of an activity. Activity views help reasoning about cause-effect relations, discover trends, and grasp the overall dynamics of a time-dependent process.

Certain methods in software analysis require combining both structure and activity insights. One such method is program comprehension through dynamic analysis [18, 159, 280]. Equally important to separately understanding static aspects (e.g., software structure) or dynamic aspects (e.g., order of function calls) is correlating these two. Among others, it can help engineers find high-activity packages, map execution phases to software module structures, and reason about performance problems at system component level.

In information visualization, many solutions exist for separate visualization of structure and activity. However, combining structure-and-activity data in a single image is still hard. This section presents a new approach to this problem. Rather than using linked views, the approach 'fuses' both structure and activity information spaces in a single view. Hereby, their spatial distance is reduced and valuable screen-space is arguably used more efficiently than with the linked-views technique. In addition, the fusing causes less disruption during interaction as the two views' average spatial distance is lower than with the linked-views technique. This section further proposes several rendering variations and interaction modes that enable users to easily map foci of interest between the two spaces, thereby supporting the task of correlating insights. The techniques are easy to implement and use, and can be applied to fuse structure and time-dependent data beyond software visualization.

5.1.1 Classification in a Task-Oriented Model

In the task-oriented model of Maletic et al. the visualization can be classified as follows: The *task* is to help users to correlate structure and activity data, specifically to (a) understand how activity maps to structure and vice versa, and (b) map metrics related to either structure or activity between the two while reducing the required screen space.

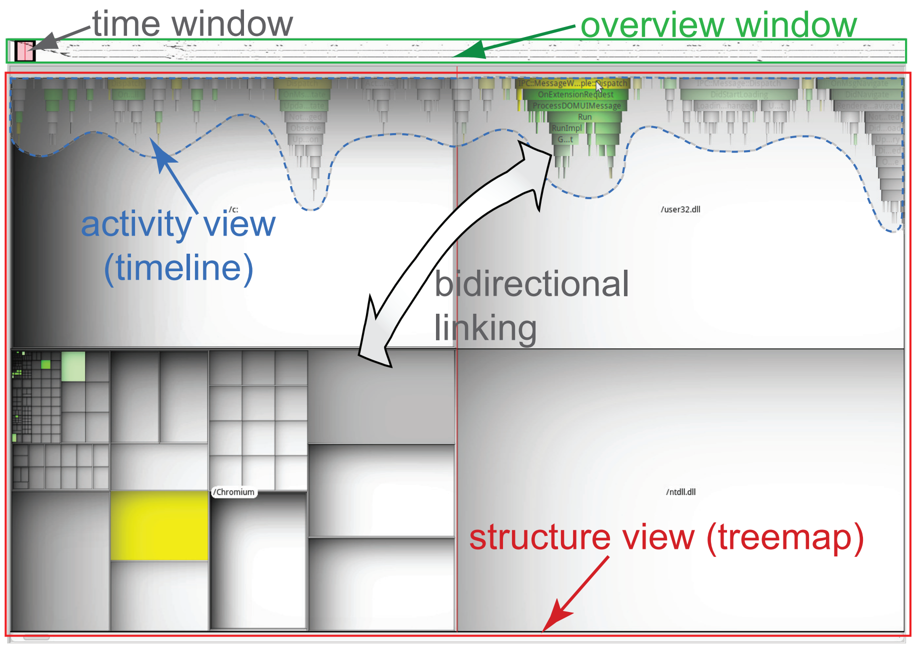


Figure 5.1: ViewFusion*: Combined structure-and-activity view design.*

The *audience* includes software engineers who want to understand correlated aspects of execution of and structure of large software systems. The visualization *targets* static software structure, behavior information (function calls and call durations) from a single thread trace, and relationships between the structure and the trace. These data are *represented* using space-filling plots (for an overview and the structure view), and an icicle plot for the call structure. Finally, the visualization *medium* consists of a standard screen with two linked views.

5.1.2 Visual Design

The goal of this section is to create a visualization design that

- combines structure and activity views in a scalable way;
- enables users to easily correlate subsets of the information shown in the two views;
- is efficient and simple to implement, and portable.

5.1.2.1 Exploration Workflow: Overlaid Layout

The design starts by using a treemap and an icicle plot for the structure and activity views, respectively. These choices are motivated by the high scalability of both views, as shown in numerous cases [68, 134, 267]. Unlike existing linked view solutions which keep such views spatially separated, the design *overlays* the views (Fig. 5.1).

Users initially select behavior data and correlated structure data to show. They either start by navigating the structure or the behavior data. While the structure view shows all structure data in a space-filling way, the behavior data can be navigated at coarse scale in an overview window to select medium and fine-scale contents for the activity view. Bidirectional linking between structure and activity view enables users to correlate (groups of) items.

5.1.2.2 Structure View

This view uses a treemap to display a hierarchy H. In the given use-case, H stores containment relationships in a software system. The structure view uses here a simple strip layout [22], which has some advantages as discussed later (Section 5.1.3.3). To maximize information density (one of the design goals), the structure view does not render borders between sibling cells, except a 2-pixel border on the topmost level to emphasize high-level borders. Hence, it needs other ways to show neighbor cells belonging to the same subtree. An option is to use cushion treemaps (CTMs) [267]. CTMs use a Phong-shaded height map built by summing up parabolic profiles $\psi_i : [0, 1]^2 \to \mathbb{R}^+$ centered atop of tree nodes $n_i \in P$ belonging to the same path $P \subset H$. If a slanted light vector is used, shading discontinuities convey the tree-distance of neighbor cells, i.e., the larger the shading discontinuity between those cells, the larger the distance of the respective nodes in H.

The parabolic cushions in the original CTM design have linearly-changing gradients. Thus, to create high-contrast images that convey the tree structure, CTMs use relatively high Phong specular coefficients. This can visibly darken the result (see [267], Fig. 6). Also, CTMs require per-pixel computations that cannot be efficiently done except if using pixel shaders. Although this is possible [173], it conflicts with the portability and simplicity requirements.

The structure view takes here an approach similar to CTMs, but which is simpler and generates brighter, easier to read images. For each node $n \in H$ at depth $d_i \geq 0$ from the root of H, it uses a luminance profile as in Eqn. 4.3. Setting $f_x = 1, f_y = 0.8, k = 0.2$ gives an effect similar to the original CTM design.

To render the treemap, the structure view multiplies – at each screen pixel (x, y) – the profiles ψ_i for all treemap cells that cover (x, y). It does this easily by storing ψ_i, for all depths d_i, as 2D luminance textures, and rendering H with textured cells in depth-first order with multiplicative alpha blending. Fig. 5.2 shows the rendering of the hierarchical tree of a software system with 8,850 elements. Several differences are apparent between this design and CTMs [173, 267], as follows. The image is much brighter: In contrast to CTMs, the flatness of the profiles ψ_i increases with tree depth, due to the increasing exponent d_i in Eqn. 4.3. Thus, deep tree cells have a relatively much wider highlight than cells higher in the tree. The asymmetric shading profile, which visually separates neighbor cells whose nodes are far apart in H, is preserved. Overall, this shading slightly reduces the mapping of tree-depth to luminance (present in CTMs) but yields an overall brighter image. As we shall see next in Sections 5.1.3 and 5.1.4, this is useful for color-mapping metrics to the structure view.

5.1.2.3 Activity View

The activity view reuses the basic visual design as introduced in TraceVis. It combines this design with the advanced rendering techniques for icicle plots (cf. Section 4.2.2) to better convey structure and selection in the overlaid layout, which is described in the next section. In contrast to TraceVis, ViewFusion depicts only a single thread trace $T \in \mathbb{T}$ of a software trace $\mathbb{T}$ (for definitions see Section 2.1.4).

5.1.3 Interaction Techniques for Linking and Occlusion Reduction

In many applications, like the given program comprehension use-case, the tree H in the structure view and the activity data T in the activity view are *correlated*: Each function call – which will be referred to as event $e \in T$ in the following – is associated with a structural element $n \in H$ via an activity-to-structure mapping $m : T \to H$. Note that m

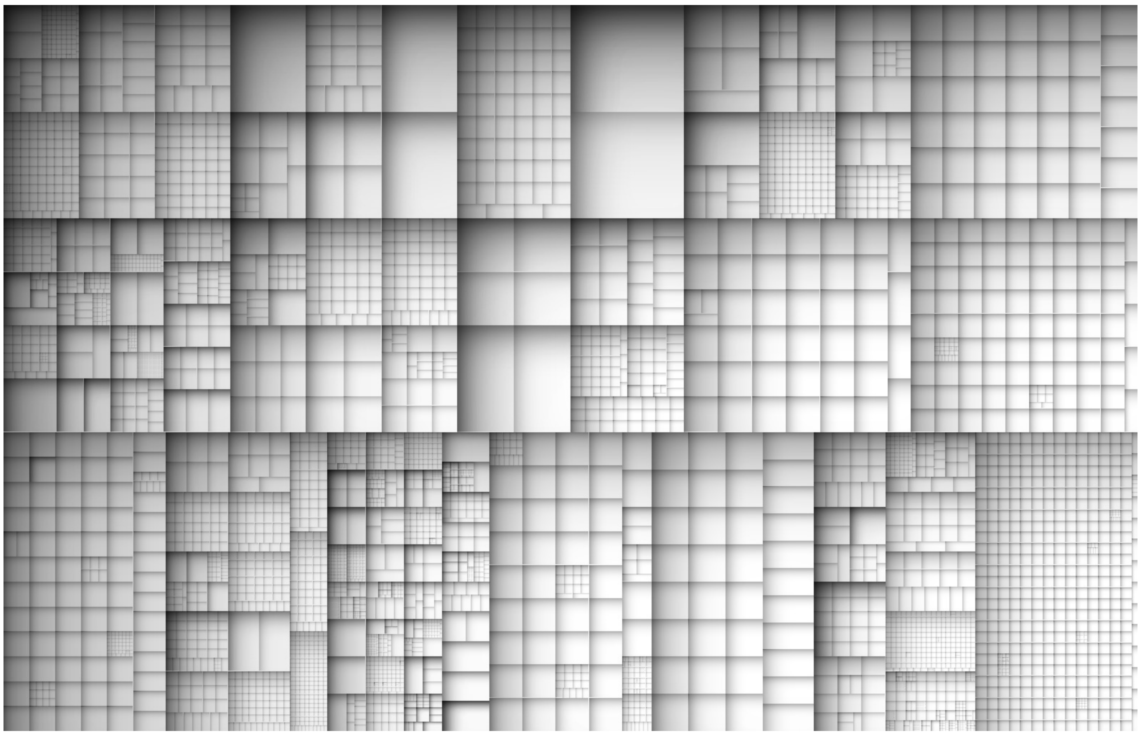


Figure 5.2: *Treemap rendering for a dataset of 8,850 nodes.*

need not be injective. For instance, in the given trace dataset, m maps from function calls to function declarations; most software traces comprise several calls to the same function. Showing such correlations by visualizing m and its inverse m^{-1}, i.e., *linking* the two views, is an important requirement.

A separate issue regards the *occlusion* created by drawing the activity view atop of the structure view (Fig. 5.1). As noted, ViewFusion does this to minimize the space needed to show both views. Indeed, if it was to stack the views, this would double the required screen space in the worst case. Overlaying the views is space-efficient, but creates undesired occlusions.

5.1.3.1 Selection by Brushing

Both issues mentioned above, i.e., visualize the activity-to-structure mapping m and its inverse m^{-1}, and reduce the activity-view *vs* structure-view occlusion, are addressed by interaction. To explain this, this section reuses the three key elements of the interaction design for activity views as described in Section 4.2.1: Focus point $\mathbf{p}_f$, focus item o_f, and focus subset s_f. Users can toggle the view V to interact with, called the *input target*, via the Control key. If $V = H$, $o_f \in H$ is a node in the tree shown in the structure view. If $V = T$, $o_f \in T$ is an event in the sequence shown in the activity view.

By suitably choosing o_f and s_f, the view-linking and view-occlusion issues can be addressed, as described in Section 4.2.2. So, by defining a focus point and interaction mode, focus item and focus subset can be determined. As out-of-focus items $(T \setminus s_f)$ in the activity view are rendered with a high transparency and low saturation, this reduces occlusion in two ways. First, one can move the mouse within, or around, the activity view to bring different items in focus, as outlined in Section 4.2.1. Secondly, one can grab the activity view and pan it horizontally. The two operations allow for de-occluding any part of the structure view by a mouse gesture (and optional click-to-pan).

As for the activity view, interacting with the structure view requires a focus item o_f and focus subset s_f. The focus point $\mathbf{p}_f$ is always within a treemap cell. o_f is set to this cell, and s_f to the subtree of H containing o_f and starting at a user-specified height h, which is controlled by turning the mouse wheel. Items in $s_f \setminus H$ are rendered with low saturation.

5.1.3.2 Color Linking

Colors serve two purposes in this design: First, to *color map* attributes of interest of the items in both the structure and activity views, e.g., package ID, call stack depth, and function-call starting time. Secondly, color *links* items in focus between the two views, as explained next.

As outlined in Section 5.1.3, a goal is to bidirectionally link the activity and structure views so one can correlate data shown in both: For an item $u \in T$, we want to show the items $m(u) \subset H$; for an item $v \in H$, we want to show the items $m^{-1}(v) \subset T$. Visualizing m or m^{-1} for *all* items in T and H respectively is hard or even impossible, since H and T may have thousands of items and there is only a limited number of visual variables and colors that humans can discern [276]. In this design, it would require, e.g., using a node-link metaphor that connects related items with lines. This can easily lead to unacceptable clutter. Other designs such as shared view axes [68, 188, 270] are not possible given the spatial overlaid design.

To solve this, ViewFusion restricts itself to show m and m^{-1} only for the focus subset s_f. For this, two color-linking designs are presented next. In the first design, called *data-in-focus*, items in s_f are colored via a task-specific colormap. Fig. 5.4 shows this for calls in $s_f \subset T$ colored by relative stack depth. Corresponding structure items $\{m(u)|u \in s_f\} \subset H$ use the same colormap. This shows how the metric (call stack depth) for the selected calls (s_f) varies over the function definitions ($m(s_f)$). Items $u \in T \setminus s_f$ are drawn in both views with no color mapping, i.e., gray. As the user changes s_f by brushing over T, the colored items change in both views, which allows for linking subsets of interest in these views.

In the second design, called *data-outside-focus*, items in s_f and $m(s_f)$ are left gray, and items in $T \setminus s_f$ and $H \setminus m(s_f)$ are color mapped. As the user changes s_f by brushing over T, linked items appear as gray items in both views. In contrast to the first color linking design, users can now use *different* color mappings in the two views, e.g., to show call count in the activity view and call duration in the structure view (Fig. 5.5), since linking is shown by the common gray color. This mode supports the task of identifying linked items in both views, shown in the context of view-specific metrics mapped in each view by view-specific color maps.

Color linking works for selections s_f done in both the activity and structure views. In other words, users can either select items in the activity view and see where they map in the structure view (m mapping), or select items in the structure view and see where these map in the activity view (m^{-1} mapping). As both views are drawn in the same screen rectangle, users can toggle the input target view by pressing the Control key, as outlined in Section 5.1.3.1.

5.1.3.3 Constrained Structure-View Layout

One can further exploit the treemap layout to minimize structure-view *vs* activity-view occlusion by defining a function $\gamma : H \to \mathbb{N}$ equal to the number of relations of a node $n \in H$ to the thread trace T, i.e., $\gamma = \|m^{-1}\|$. When using the strip treemap-layout, nodes

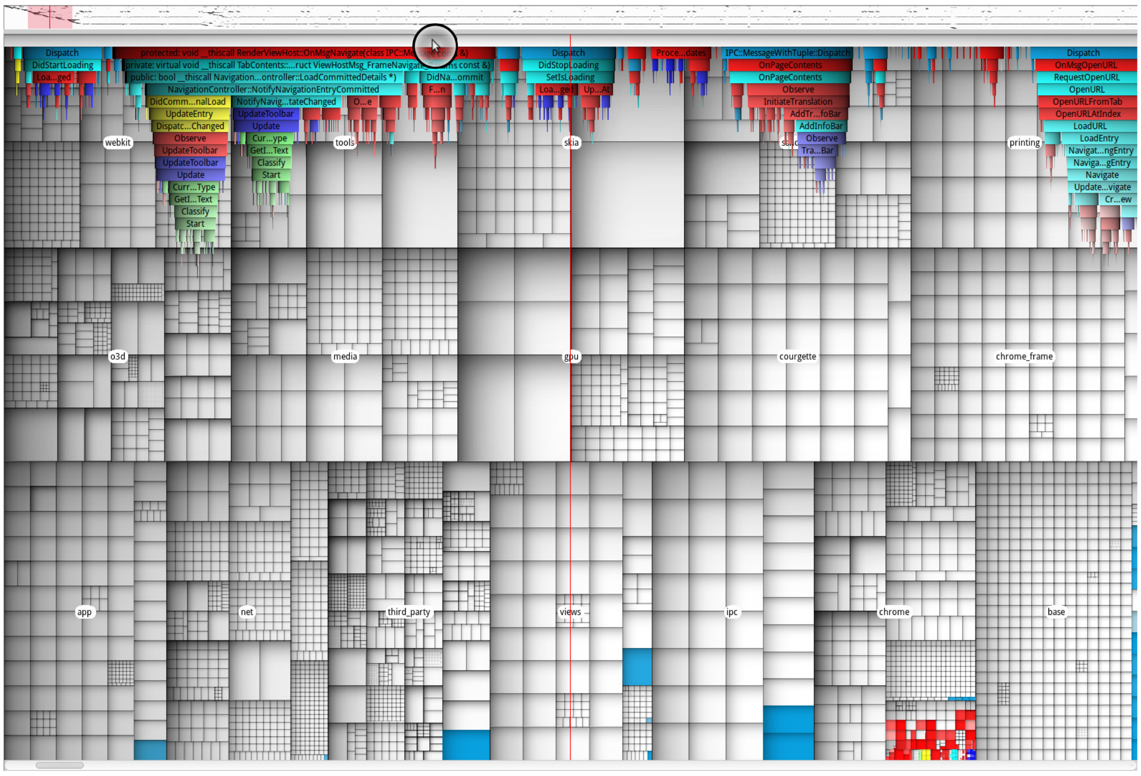


Figure 5.3: *Colors: Package ID. Input target: Activity view, Interaction mode: Detail.*

in H are sorted on γ. Treemap cells that have many relations to the activity view, e.g., often-called functions, are placed at the treemap bottom, while items with few relations go to the top. This reduces the likelihood of occlusion between both views during color linking.

5.1.4 Application in Industry-Scale Software Systems

The proposed visualization is implemented atop of TraceVis (cf. Section 4.1). As input data, a software trace of *Chromium* (version 6.0) was captured. The structure tree contains 8,850 files and folders, in total 2.7 million SLOC in C/C++. The trace, pre-filtered as the system was instrumented only partially, results in about 9,000 calls to 914 function bodies.

Within all color mappings presented next, two special colors indicate missing data: Items in the focus subset which have no data are *white*; items outside the focus subset which have no data are *blue*. The next sections present several analyses centered on correlating software structure with trace data, implemented with the proposed techniques.

5.1.4.1 Temporal Locality and Coherence of Packages

As a first use-case, we analyze how distinct packages of *Chromium* collaborate in time (Fig. 5.3). We use a categorical color mapping for both views (data-in-focus color linking, Section 5.1.3.2). Colors map the package ID for a few top-level folders of interest in the structure tree H. Next, we move the mouse to select the topmost node in the activity view, i.e., focus on the entire visible trace. We see that the yellow and green packages contribute to the execution at the beginning of the shown time frame. In contrast, the red, cyan and blue packages are involved over the entire time frame.

5.1.4.2 Structural Locality, and Coherence of Events and Packages

We now analyze structural properties of events shown in the activity view. Given a focus subset $s_f \subset T$ in the activity view, we first partition s_f into subsets of events $s_f^j = m^{-1}(n_j)$, where $\{n_j\} = \{m(e)|e \in s_f\}$ are all files in H related to events in s_f. For each file n_j, we then define two metrics m_a (activity view) and m_s (structure view) based on the events in s_f^j. Both metrics are normalized to $[0, 1]$ based on their global minimum and maximum values for all events in s_f. The metrics are then color-mapped using a 'criticality' scheme (green=low, yellow=middle, red=high).

Stack Depth: Analyzing Per-File Coherence In trace analysis, stack depth is an important measure: It gives a first hint of whether a specific function implements high-level or low-level functionality, also measured as utilityhood [112]. Typically, low stack depth means low utilityhood and high stack depth means high utilityhood. While this metric is known to work quite well at function level, we want to see how it behaves at file level, given that files can contain multi-level functionality.

To analyze coherence of utilityhood in the trace, we set $m_a(n_j) \equiv m_s(n_j)$ to the maximum call-stack depth of all correlated events in s_f^j. We visualize these metrics using data-in-focus color-linking and the criticality color scheme (Fig. 5.4). In the activity view, we can see that the file containing *TabContents::NotifyNavigationStateChanged* (1 in Fig. 5.4) is, despite its low 'visual' stack depth, colored red instead of the expected green. This means that this file not only contributes to high-level, but also to low-level functionality. In contrast, most other events in the activity view show a green-to-red downward gradient, i.e. they are defined in files whose overall functionality is homogeneous. In the structure view, we see a rather heterogeneous distribution of functionality levels in the *chrome* package.

5.1.4.3 Call Count and Duration: High-Activity Packages

Call count and call duration are often used to find how actively packages take part in the execution of specific functionality. To show these, we define $m_a = \|s_f^j\|$, i.e., the number of calls to n_j (call count per file), and $m_s = \sum_{e \in s_f^j}(t^e - t^s)$, i.e., cumulated call duration of n_j. Since $m_a \neq m_s$, we use the data-outside-focus color linking design (Section 5.1.3.2).

When brushing the structure view, we first see that most treemap cells are blue (Fig. 5.5). As this color indicates missing (trace) data, this shows that our recorded execution trace 'samples' the system tree only sparsely. This is indeed so given our partial instrumentation of the code stack. This is a quick way for users to assess the overall code coverage of a given trace. Next, we see that the color distribution in the activity view is heavily shifted to low values (green and yellow). This is due to an outlier that is visible with the naked eye in the activity view (zoom-inset 1 in Fig. 5.5): This is a timer function, *ResetBaseTimer*, called every few milliseconds and thus having a very high call count. In contrast, for the call duration metric shown in the structure view, we see no such outlier (no red treemap cells): Called functions have similar durations. Further on, in the *chrome* package, which contains the current focus subset s_f brushed by the user (2 in Fig. 5.5), several files are correlated to events in the activity view. In contrast, in most other packages only few files are correlated to events in the activity view. This is a further indication that the trace examined here mainly 'targets' functionality in the *chrome* package.

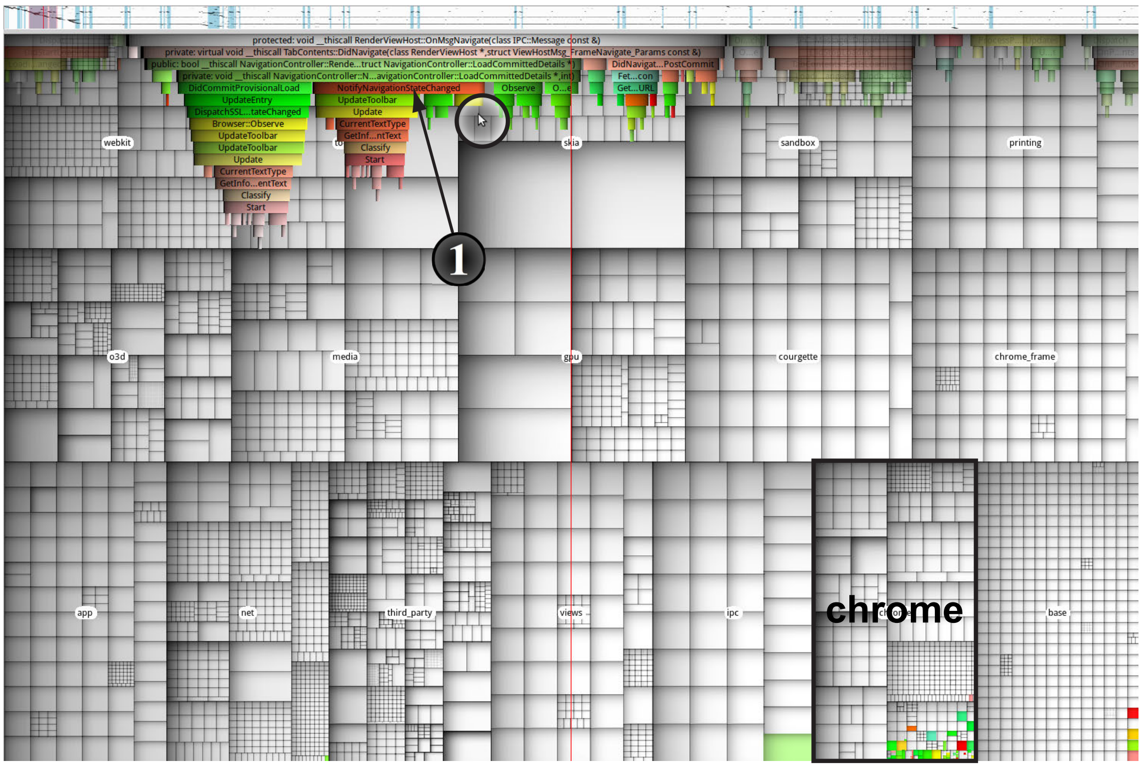

protected: void __thiscall RenderViewHost::OnMsgNavigate(class IPC::Message const &)
private: virtual void __thiscall TabContents::DidNavigate(class RenderViewHost *,struct ViewHostMsg_FrameNavigate_Params const &)
public: bool __thiscall NavigationController::Rende...truct NavigationController::LoadCommittedDetails *)
DidNavigat...PostCommit
private: void __thiscall NavigationController::N...avigationController::LoadCommittedDetails *,int)
Fet...con
DidCommitProvisionalLoad
NotifyNavigationStateChanged
Observe
O...e
Get...URL
UpdateEntry
UpdateToolbar
DispatchSSL...tateChanged
Update
Browser::Observe
CurrentTextType
UpdateToolbar
GetInf...ntText
UpdateToolbar
Classify
Update
Start
CurrentTextType
GetInfo...entText
Classify
Start
webkit
skia
sandbox
printing
1
o3d
media
gpu
courgette
chrome_frame
app
net
third_party
views
ipc
chrome
base

Figure 5.4: *Colors: Maximum stack depth. Input target: Activity view, Interaction mode: Approach.*

5.1.5 Discussion

Generality ViewFusion can show a tree and a hierarchical event sequence, and highlight relationships between items in the two datasets. Although this approach is demonstrated on static software structure and software traces, the proposal is in no way restricted to software engineering datasets; it generally applies to structural data and correlated temporal data.

Scalability Given the space-filling treemap layout and the space-efficient icicle plot layout, the design scales well to datasets containing thousands of elements. The shaded cushions used in both views convey additional structural cues, e.g., nesting of events in the activity view and tree in the structure view. This makes the views usable in zoomed-out mode when items are only a few pixels large.

Ease of Use The proposed interaction techniques are easy to use: Selecting item subsets of interest in either of the views, and zooming and panning to a time range of interest in the activity view, are done using only the mouse motion, mouse wheel, and the Control key. Although a formal user evaluation was not conducted, insight so far shows that the proposed method is intuitive and easy to use within minutes. Integrating multitouch-based input devices likely allows for further increasing the easiness of user input.

Flexibility The proposed brushing-based selection techniques enable easy and quick adjustments of level-of-detail of selection by moving the mouse or turning the mouse wheel. The two color-linking techniques (data-in-focus, data-outside-focus) based on the selection help correlating two *different* metrics at a time. Also, the linking techniques enable users to project any metric computed on one of the two datasets to the other dataset, effectively enabling a ‘push’/‘pull’ of metric data from one view to the other.

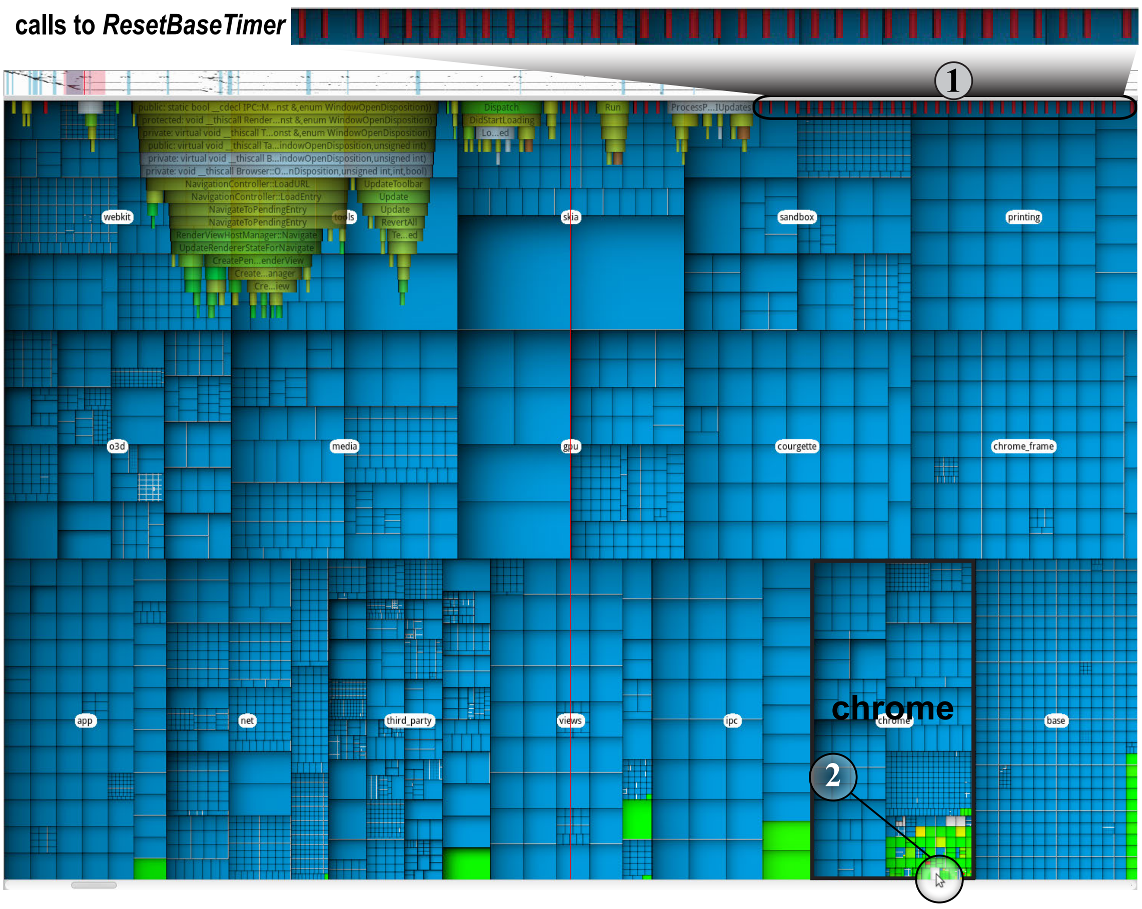


Figure 5.5: *Colors: Call count (in activity view) and call duration (in structure view). Input target: Structure view.*

Limitations Occlusion reduction by pan-and-brush combined with transparency (Section 4.2) works well even for large datasets, e.g., deep icicle plots overlaid on deep treemaps - one can 'see through' any icicle plot area just by moving the mouse and/or panning this plot. However, this still requires a (small) amount of user interaction. The likelihood of occlusion is further reduced by our strip-treemap reordering based on relation count. Even more aggressive occlusion reduction could be done by exploiting treemap layouts which allow for more flexible reordering schemes.

The color-linking technique (Section 5.1.3.2) can only show relationships between *groups* of items, i.e., the focus subset s_f *vs* its mapping $m(s_f)$. Although one-to-one relations can be inferred by seeing how highlighted elements change while brushing and/or selecting smaller subsets s_f, this does not replace a detailed relationship visualization. Further refinements could add, e.g., carefully routed bundled edges [123] atop of our design to emphasize such details.

Benefits ViewFusion is a method for the visualization of combined structure tree and event-sequence datasets. It addresses visual scalability by a new overlaid layout of icicle plots and treemaps. By this, the two views are linked with less disruption. It uses simple, brushing-based interaction for selection of items of interest and occlusion reduction. Combined with color-linking, this allows for querying relationships between parts of the two displayed datasets. Computationally-efficient shaded cushion variations are proposed for structure and focus enhancement. The technique is illustrated on large datasets from program comprehension, but can be used on other structure-and-activity datasets.

5.2 TraceDiff: Multiscale Visualization of Behavioral Correlations

This section is based on the following publication(s):

- Jonas Trümper, Jürgen Döllner, and Alexandru Telea. Multiscale Visual Comparison of Execution Traces. In Proceedings of International Conference on Program Comprehension, pages 53-62. IEEE, 2013. [305]

Trace comparison can yield unique insights into certain behavior aspects. Comparing multiple runs of the *same* software system can help investigating how changes to the *execution context* of this system influence its behavior. Examples include questions such as why different inputs do (or do not) result in different outputs (variable execution context), and why multiple executions of the same functionality result in different outcomes (non-determinism) [256]. For example, the latter is important to ensure stability of execution in various deployment configurations, which includes finding why programs exhibit different behavior when running on different machines [158]. These scenarios are the focus of this section. Apart from that, comparing traces can aid in several other use cases. For instance, one can also analyze which effects code changes have on behavior by differencing traces captured from successive software versions.

However, comparing traces is hard for several reasons. According to a classification of tree-comparison problems by Guerra-Gómez et al. (cf. Section 3.4), comparing traces is a type 4 problem, which they consider the most complex in their classification: Changes affect both the tree topology (here: call relationships) as well as attribute values (here: execution duration) of both leaf and non-leaf nodes. Moreover, the sheer amount of data generated during tracing already poses various analysis and (visual) representation challenges [286]. Its analysis already has to depict many types of information: Time-stamps, call identities, static software structure, and the relationships between such entities. If we want to compare two traces rather than analyze a single trace, the data volume only doubles, so we need scalable and effective ways to show similarities and differences between the two traces.

This section presents a visualization method, named TraceDiff, for the interactive comparison of two large thread traces, e.g., from multiple runs of the same software system. The visualization design focuses on two main goals. First, it addresses visual *scalability* by a multiscale design that supports exploration from coarse-grained events of interest, such as aggregated execution similarities and differences, to fine-grained events such as function-level call similarities. For this, it uses and extends several visual metaphors: space-filling plots (for the overview), shaded icicle plots and tube bundles (for the intermediate level), and attribute color-mapping and edge bundles (for the fine-grained level). Several interaction mechanisms are proposed to help specific user tasks at each level-of-detail, such as finding the most (dis)similar execution fragments; explaining these (dis)similarities by highlighting differences such as execution swaps, call time-shifts, and call durations; and finding execution fragments replicated several times between traces.

5.2.1 Classification in a Task-Oriented Model

According to the task-oriented model of Maletic et al., the visualization can be described as follows: The *task* is to help users compare two large-scale traces (type 4 tree-comparison problem), specifically to (a) detect execution regions that are (dis)similar; (b) explain the (dis)similarities at several levels of detail; and (c) correlate execution (dis)similarities with static software structure. The *audience* includes software engineers who want to understand execution aspects of large software systems. The visualization *targets* static software structure, trace information (function calls and call durations) from two thread traces, and similarity relationships between the two traces. These data are *represented* using a space-filling plot (for overviews), icicle plots (for the call structure), and a multiscale bundling metaphor (for the trace-to-trace similarity relationships). Finally, the visualization *medium* consists of a standard screen with two linked views.

5.2.2 Trace Comparison

5.2.2.1 Trace Similarity

To compare two thread traces (see definitions in Section 2.1.4) T_A and T_B, a so-called similarity function $s : T_A \times T_B \to [0, 1]$ is defined, where for any two calls $f_A \in T_A$ and $f_B \in T_B$, $s(f_A, f_B)$ gives the similarity of the entire call stacks $S(f_A)$ and $S(f_B)$. s is computed using the method originally applied for detecting repeating patterns in a single trace [32], but now using two traces, as explained next. Each function definition F is given a unique ID. For each call stack $S(f)$, the comparison stage of the algorithm computes the set $\Gamma(f)$ containing all IDs of calls in S

$$\Gamma(f) = \{F' | \exists f' \in S(f), F(f') = F'\} \tag{5.1}$$

Next, given two calls $f_A \in T_A$ and $f_B \in T_B$, it computes

$$s(f_A, f_B) = \frac{\|\Gamma(f_A) \cap \Gamma(f_B)\|}{\|\Gamma(f_A) \cup \Gamma(f_B)\|} \tag{5.2}$$

where $\| \cdot \|$ denotes set size. For more details on this comparison function, refer to the literature [32]; an overview of this and various other tree comparison methods is given by Adamoli and Hauswirth [4]. This comparison can be applied to thread traces originating from one or more software traces. For an application to multiple software traces, the ID assignment has to ensure that equivalent function definitions (originating from any input trace) are assigned the same unique ID.

5.2.2.2 Match Groups

The above process delivers a potentially very large set of pair-wise similarities between call stacks in the two traces, so this set is reduced as follows in a grouping stage. In practice, we are interested only in call stacks having a large similarity $s(f_A, f_B) > \tau$. Setting $\tau \in [0.1, 0.3]$ has given good results in our trace comparisons. When such stacks exist, we say that there exists a *match* $k(f_A, f_B)$. For any such k, there exist also several matches $k'(u, v)$, where $u \in S(f_A)$, and $v \in S(f_B)$: Two similar call stacks share similar sub stacks. The set of matches k', which explain k, is called a *group* $G(f_A, f_B)$. Further, k is called the *root* match of G, and is denoted as $R(G) = (G_A, G_B)$ with $G_A = S(f_A)$ and $G_B = S(f_B)$.

The grouping stage next partitions all computed matches into a minimal set of groups, as follows. Starting with no group, it traverses one of the trees, e.g., T_A, in breadth-first order. For each encountered call $u \in T_A$, it iterates over the call's matches $k(u, v \in T_B)$. If there is no current group G or u is not contained in G_A or v is not contained in G_B, the grouping stage creates a new empty group and makes it current. Otherwise, it adds k to G and continues the traversal. The groups will later be used as input to the visual trace comparison (Section 5.2.3.3).

The trace comparison described above delivers a set $K = \{k = (f_A \in T_A, f_B \in T_B)\}$ of groups containing matches between sufficiently similar stacks, i.e., $s(f_A, f_B) > \tau$. The next section shows how these groups and the hierarchical trace data are used to visually analyze trace similarities.

5.2.2.3 Temporal Alignment

The input data – two thread traces – are not guaranteed to be captured in parallel but at different absolute points in time. Consequently, t^s and t^e of both traces' root calls can have different absolute values even if their duration is equally long. TRACEDIFF addresses this by aligning the two thread traces temporally.

Based on the rationale that capturing the two traces commonly starts at the same point in the execution (e.g., triggered by a specific function call), but may end at different points in time due to variations in the executions, the two traces are by default aligned with respect to their beginning: TRACEDIFF thus uses one of the start moments of the two root calls f^A_{root}, f^B_{root} in T_A, T_B as pivot for the alignment. This can be easily done by computing an offset t_o between f^A_{root}, f^B_{root} and subsequently shifting any time stamps in one of the two traces – here T_A – by this offset. So we set

$$t_o(T_A) = t^s(f^B_{root}) - t^s(f^A_{root}) \tag{5.3}$$

$$t_o(T_B) = 0. \tag{5.4}$$

Other options include aligning with the strongest match. This, however, is potentially ambiguous if there are multiple equally strong matches.

5.2.3 Visual Design

5.2.3.1 Exploration Workflow

The visual design uses a focus-and-context approach as shown in Fig. 5.6. Users initially select a pair of already computed thread traces (T_A, T_B) that they wish to compare. If match data (Section 5.2.2.1) is available for this pair, it is used directly, else it is computed on demand and stored for later use. After the trace pair and match data are loaded, users navigate via the *overview* window (described next in Section 5.2.3.2) to find interesting execution areas, or *focus* areas, in the two traces. These can be areas in a trace where many matches exist with the other trace. Alternatively, users can select specific focus areas in the two traces and compare them – for instance, compare two executions around the same moment. After selecting the focus areas, users can interactively examine the *comparison* window to get insight on the (dis)similarities of the call stacks in focus (see Section 5.2.3.3).

5.2.3.2 Overview Visualization

The overview visualization shows an aggregated view of both thread traces' full time extent (Fig. 5.7), with the upper part of this view dedicated to T_A and the lower part

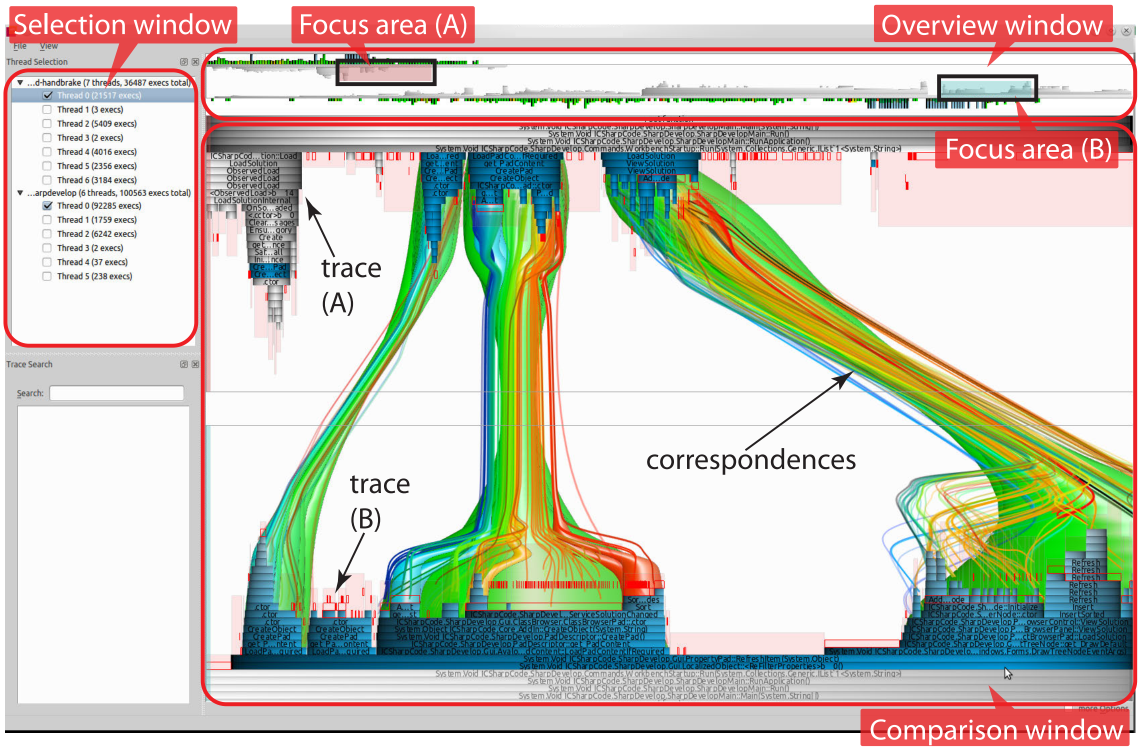


Figure 5.6: TRACEDIFF*: Trace comparison visualization design.*

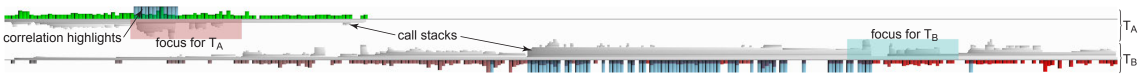


Figure 5.7: *Overview visualization. The top and bottom parts show the match highlights and call stacks for the two compared traces, respectively.*

to T_B. For T_A (Fig. 5.7, top trace), the lower graph shows a structure-based overview (cf. Section 4.1.2.3). The upper graph shows a bar chart drawn over N equal-sized time intervals, with N set so that each interval maps to 10 screen pixels. For T_B (Fig. 5.7, bottom trace), the two graphs are mirrored in the y direction.

For each such time interval $[t^s, t^e]$, the overview draws a bar whose height encodes the sum of similarities s of the matches that the trace has over $[t^s, t^e]$. The bar chart interpretation is simple: High bars show execution areas where there are many strong matches.

Bar backgrounds help panning the foci of the two traces to find matching calls. Using these cues, users can select the areas of interest for their specific use-cases, and next use the comparison window to gain finer-grained insight.

5.2.3.3 Match Visualization

Given a match-rich focus area in a trace-pair, the match visualization in the comparison window shows details on these matches, as follows (see Fig. 5.8). First, it renders the two thread traces using two mirrored activity views as in TRACEVIS (Section 4.1.2.3), including the shading and brushing as described in Section 4.2. Finally, the match visualization outlines found call repetitions (Section 5.2.2) in red (cf. [32]).

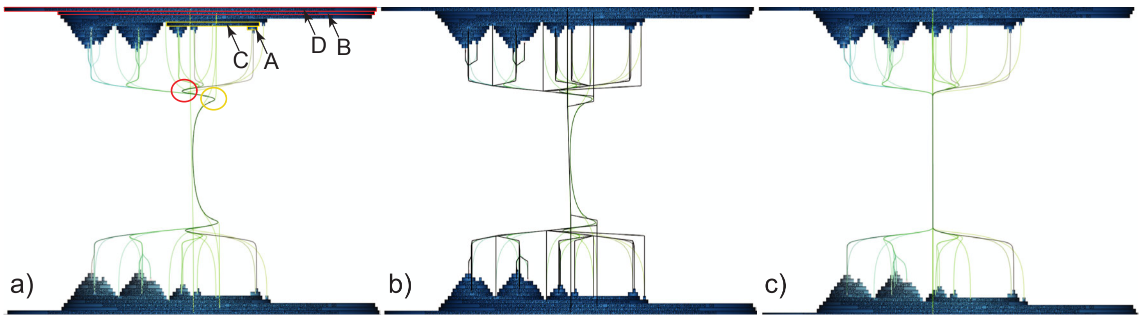


Figure 5.8: *Original HEB bundling (a) and its control tree (b); undulation artifacts due to node widths are visible; the modified bundling (c).*

Next, edge bundles are used to connect matched calls in the two traces. The algorithm starts by the hierarchical edge bundling design of Holten et al. [123, 124]: Match edges are routed along a tree structure computed using the hierarchy represented by the icicle plot nodes, mirrored along the x axis. However, the subject data differs from Holten's in several important respects:

R_1 - *Non-Uniform Icicle Plots:* Since width of the icicle-plot nodes encodes call duration, they have highly different widths;

R_2 - *Matches Between Non-Leaf Nodes:* The match visualization has to draw matches between non-leaf nodes (which encode execution similarities at coarser scales);

R_3 - *Vertical Overlap:* The empty vertical space between the two activity views can be quite small (due to the arbitrary depth of the call stacks). Nodes of the upper icicle plot can fall below the lower icicle plot nodes (Fig. 5.9 b). This never happened in earlier HEB designs;

R_4 - *Subpixel-Sized Icicle Nodes:* Many icicle plot nodes have subpixel size (short function calls). For large traces, including such nodes in the HEB layout creates cluttered bundles and slow-to-draw images.

Given the above, using the original HEB method creates strong visual artifacts. Several examples follow. Fig. 5.8 a shows two wave-like structures in the bundles (marked in red and yellow). These appear since the nodes A and C are not centered horizontally within their respective parents B and D (issue R_1). Figs. 5.9 a,b show several very sharp bundle bends (red markers) and undulations (green marker) due to the relatively small space between the activity views (issue R_3). For large traces containing hundreds of thousands of calls such problems become only bigger and more frequent.

To solve such issues, a modified HEB layout is applied, as follows. First, the match visualization aggregates call stacks as described in Section 4.2.2 (pink rectangles in Figs. 5.9 a,b, (issue R_4)). Next, it reserves a horizontal band B centered around its mid-line (issue R_3, see Figs. 5.9 a,b). This is the area in which bundling will take place. For each group G, the match visualization builds a separate control tree-pair, one tree for $S(G_A)$ and one for $S(G_B)$. As tree control points, it uses only the centers of those nodes in $S(G_A)$ and $S(G_B)$ which are broader than 1 pixel *and* also fall outside B, which are next called *key nodes.*

When constructing the control tree for a group G, the match visualization also clamps the x coordinate of each node f to the x range of the control-points of all child nodes

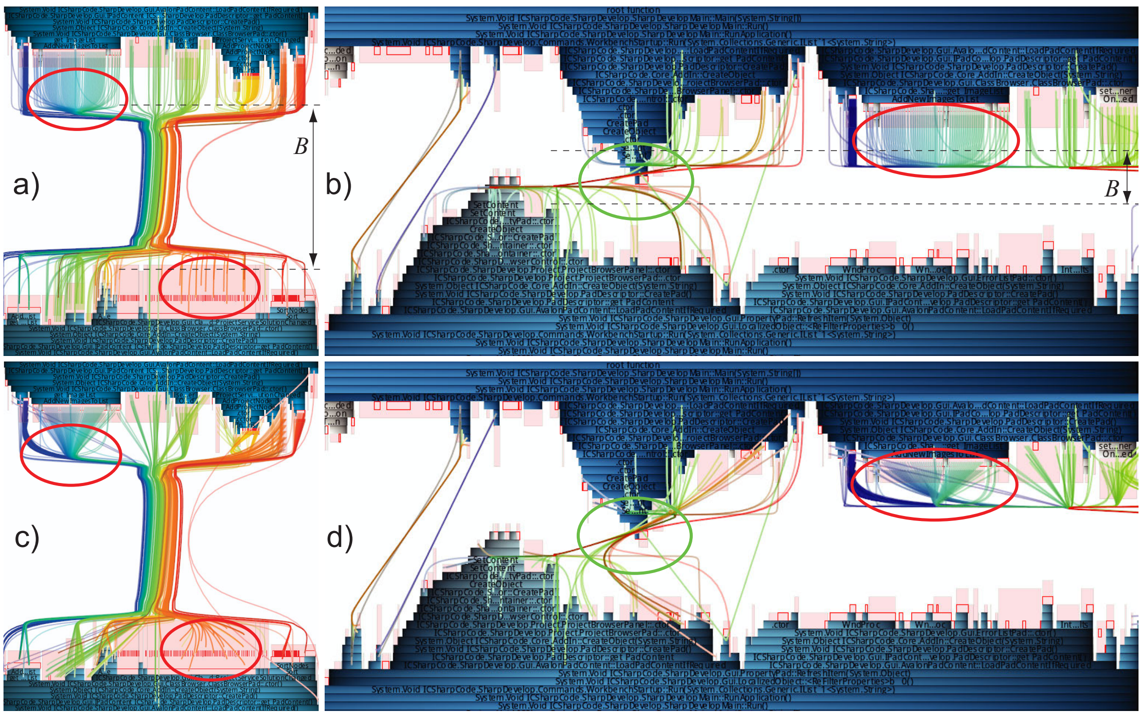


Figure 5.9: *Original HEB bundling (a,b) showing sharp bends (red markers) and undulations (green marker); the modified bundling (c,d).*

in $S(f)$ that have matches in G. This shifts the control points horizontally so that the resulting control tree has far less right-left twists. In turn, this reduces the amount of horizontal undulations in the resulting bundles, and thereby removes artifacts of type R_1 and R_2 (compare Fig. 5.8 c with the original HEB in Fig. 5.8 a).

Given the above control tree, the match visualization next adds the non-key nodes (narrower than 1 pixel or falling within B) to the tree, as follows. For each non-key node n, it ascends the node's call stack until it finds a parent p_n which was added to the control tree. Such a parent always exists, as nodes closer to the call-stack root are broad and far away from B. Next, the match visualization scans the control tree downwards from p_n and finds the node q whose control point is geometrically closest to n, and adds n as a child of q in the control tree. Hence, all non-key nodes are added as leaves to the control tree. As such, the coarse-level structure of the control tree and, more importantly, its height are not changed, and matches from non-key nodes are smoothly 'merged' into the bundles of key nodes. Comparing Figs. 5.9 c,d, which use the modified bundling, with Figs. 5.9 a,b (original HEB), we see that the undesired sharp bends and undulations have been removed, and the bundle appears centered within the x extents of the matched nodes.

Most existing applications of HEB feature edges which connect equally-sized nodes, located at the same hierarchy level, and which are laid out regularly, e.g., along lines [20, 124] or circles [124]. The modified bundling relaxes these restrictions, so it can be used in other contexts in which the original HEB method delivers suboptimal results.

5.2.3.4 Multiscale Match Visualization

Although the modified HEB method helps answering questions on the sizes and time offsets of matching execution fragments, several questions remain (see, e.g., Fig. 5.10). First, a HEB rendering cannot show *permutations* in matched sequences, since the inherent

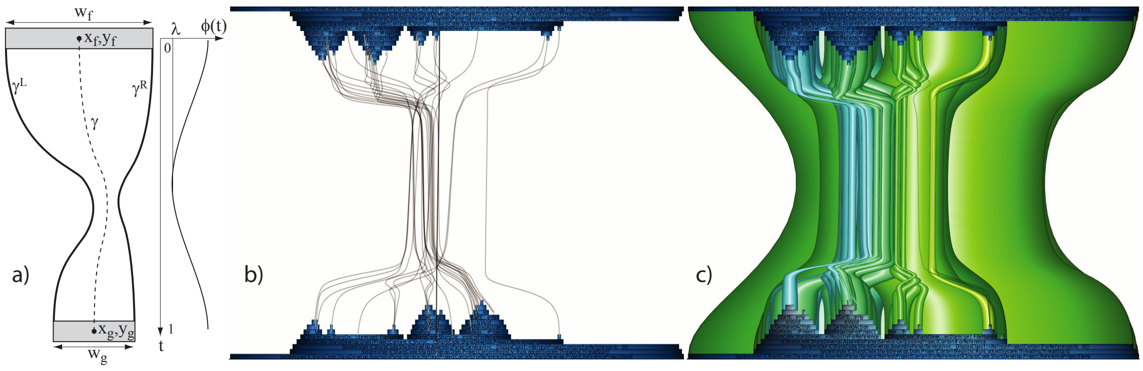


Figure 5.10: *Tube design (a). Tube bundles (c) add more information on the match start and endpoints and nesting than line bundles (b).*

overdraw caused by tight bundles makes it very hard, if not impossible, to follow individual edges. Such permutations are inherent to our match computation. Second, we recall that matches between shallower call levels are more relevant than deeper-level matches (Section 5.2.2). However, HEB renders all edges identically, so one cannot easily spot more important edges. Finally, HEB represents edges as 1D curves. This makes it hard to see, for such an edge, which are the durations of the matched elements it connects.

The above issues are addressed by a multiscale HEB visualization, inspired by image-based edge bundling (IBEB) [259], i.e., edges are drawn as shaded 2D tubes instead of 1D curves as in HEB (Fig. 5.10 c). This is explained next.

Tube layout Consider two calls f and g that are connected by a match k. Let x_f, y_f, w_f be the x and y coordinates of the center and width of the icicle plot rectangle for f, and x_g, y_g, w_g the similar quantities for g (see Fig. 5.10 a). Let γ be the modified HEB curve that connects the two centers, computed as described in Section 5.2.3.3. Let $t : [0, 1]$ be an arc-length parameterization for γ. The tube algorithm constructs two curves γ^L and γ^R which represent the left, respectively right, curved borders of a tube shape. If $\gamma^L = (\gamma^L_x(t), \gamma^L_y(t))$, we set

$$\gamma^L_x(t) = \gamma_x(t) - \phi(t)\left((1-t)\frac{w_f}{2} - t\frac{w_g}{2}\right)$$

$$\gamma^L_y(t) = \gamma_y(t)$$

where $\phi : [0, 1] \to [0, 1]$ is a function that models the gradual shrinking, or thinning, of the tube from its ends towards its center. Profiles that generate bundle-like tubes are given by

$$\phi(t) = \lambda + \frac{1-\lambda}{2}(1 + \cos 2\pi t). \tag{5.5}$$

Similarly, we construct the tube's right-border curve γ^R as

$$\gamma^R_x(t) = \gamma_x(t) + \phi(t)\left((1-t)\frac{w_f}{2} - t\frac{w_g}{2}\right)$$

$$\gamma^R_y(t) = \gamma_y(t).$$

Tube shading Pseudo-shading visually emphasizes the tube bundles using a cushion-like luminance texture, dark at the borders γ^L and γ^R and bright in the center (γ). For this, a 1D convex parabolic shading profile ranging from 0 (dark) to 1 (bright) is defined, similar to the well-known cushion treemap design [267]. Next, the shading algorithm renders each shape by discretizing γ^L and γ^R with 50..100 sample points, and drawing the resulting quads using a 1D texture encoding the shading profile. Fig. 5.10 c shows the result: The tubes appear like 3D shaded shapes that smoothly connect their corresponding icicle-plot elements. The design of the profile ϕ ensures that the tubes follow the shapes of their corresponding edge bundles – compare, e.g., Figs. 5.10 b,c. This allows for smoothly toggling between line and tube visualizations, or generating visualizations containing both tubes and lines in the same image.

The shaded tubes are visually quite similar to IBEB bundles. However, important differences exist: First, while IBEB constructs tube-like shapes in order to *simplify* an existing HEB drawing, the shaded tubes represent *one-to-one* the edges, as our aim is to show the time (horizontal) extents of all matched icicle-plot nodes. Second, IBEB has a highly involved implementation, which uses edge clustering, image blurring, distance transforms, and skeletons. It delivers interactive frame rates only for static control trees. In this trace-comparison use case, however, the control tree is dynamic, i.e., its layout changes when users change the horizontal offset of T_A relative to T_B through panning (see Section 5.2.4). The tube shading only uses a few simple curve interpolation and hardware-accelerated 1D texture mapping operations. Consequently, our method provides interactive frame rates on typical trace pairs containing thousands of matches. This is essential for their interactive analysis.

Tube stacking To combine the shaded tubes in a final image, tube stacking adds a z (depth) coordinate γ_z to the curve γ, computed by linearly interpolating the call-stack depths of its endpoints f and g, and next sets $\gamma^L_z = \gamma^R_z = \gamma_z$. Rendering the tubes with standard depth (Z) buffering shows higher-level (coarser) matches behind lower-level (finer) ones. Due to Z buffering, this also directly handles matches that connect different stack-depths in the two traces.

Fig. 5.10 c shows the overall result of the bundled tubes. In contrast to line bundles (Fig. 5.10 b), we now clearly see the time extents of the matched call stacks, encoded as tube thickness, and we can separate coarse matches (thick tubes) from fine ones (thin tubes). Also, crossings are now clearer. The overall result is a multiscale match visualization, in which matches of high-level and long call stacks (which are more important) are visually prominent.

Finding execution replications Tube bundles also help finding fragments from a trace that are replicated several times into the other trace. Fig. 5.11 shows this: At a coarse level, the largest visible tube T_f (behind all other tubes) shows a strong similarity between the largest part of the top trace (stack rooted at f_1) and the first part of the bottom trace (stack rooted at f_2). Finer-grained tubes *explain* this similarity: For example, the tube T_g shows that the above stacks are similar because the sub-stack rooted at g_1 (top trace) is similar to the bottom-trace sub-stack rooted at g_2. This similarity is in turn explained by the tube T_h, which shows that the sub-stack rooted at h_1 (top trace) is similar to the one rooted at h_2 (bottom trace). However, we see that the sub-stack rooted at g_1 (top trace) is *also* similar to a second sub-stack rooted at j_2 (bottom trace). This is shown by several diagonal tubes marked as T_j.

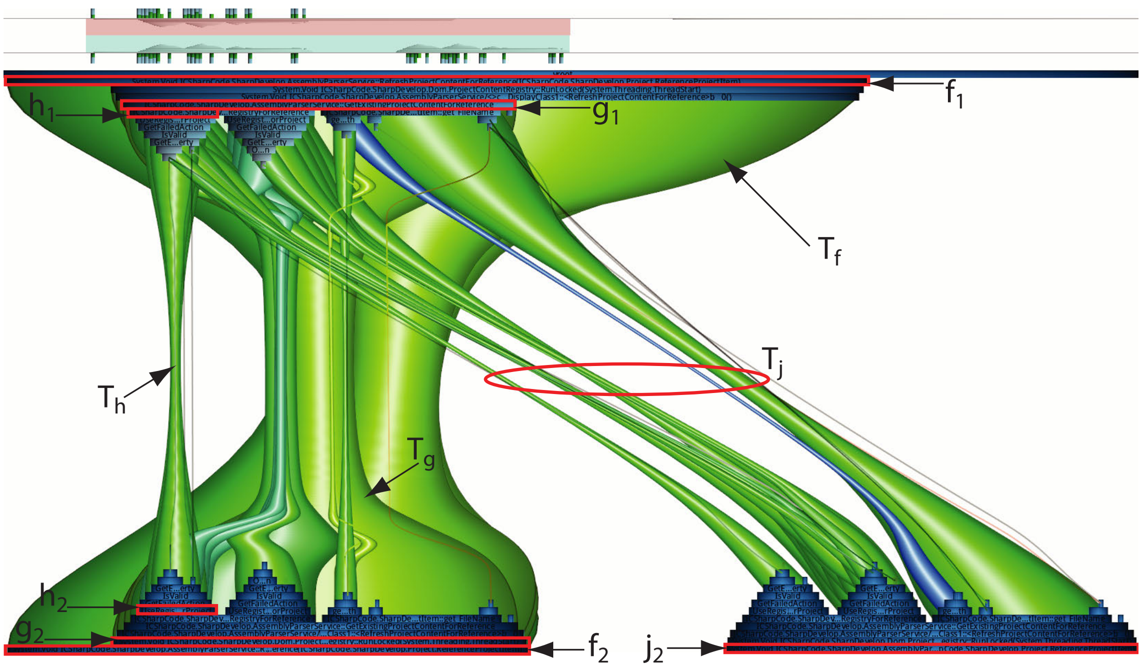


Figure 5.11: *Finding execution duplicates.*

5.2.4 Interaction Techniques

Both traces in the match visualization can be interactively zoomed and horizontally panned (see Fig. 5.12 a and b respectively). Brushing with the mouse over a call stack restricts color mapping in the match visualization (Section 5.2.5.2) to matches contained in groups rooted in that stack. All other matches are drawn in gray. This helps focusing the analysis on specific match groups.

The bundling parameters can be adjusted to obtain several effects: Tube thickness $\lambda \in [0, 1]$ (Eqn. 5.5) can be set to create thinner tubes (with less occlusions, Fig. 5.12 c) or thicker tubes (which better show call nesting, Fig. 5.12 d). The thickness of the band B (Section 5.2.3.3) can be set to create shallower bundle control trees (which help following the main bundles, Fig. 5.12 e) or taller control trees (which help seeing where the tubes connect to the icicle plots, Fig. 5.12 f).

5.2.5 Color Mapping

Mapping several attribute values that are relevant to questions of interest further enriches the trace visualization. This concerns both the overview visualization as well as the match visualization.

5.2.5.1 Overview Visualization

An important use-case for attribute mapping in the overview visualization is top-down navigation for quickly identifying regions of interest. For each bar in the overview visualization, it thus computes the relative start-time δ of the matched calls in the other trace over $[t^s, t^e]$, i.e.

$$\delta = \sum_{f \in T | k(f,g) \in K \ \wedge \ t^s \leq t^s(f) \leq t^e} |t^s(f) - t^s(g)| \tag{5.6}$$

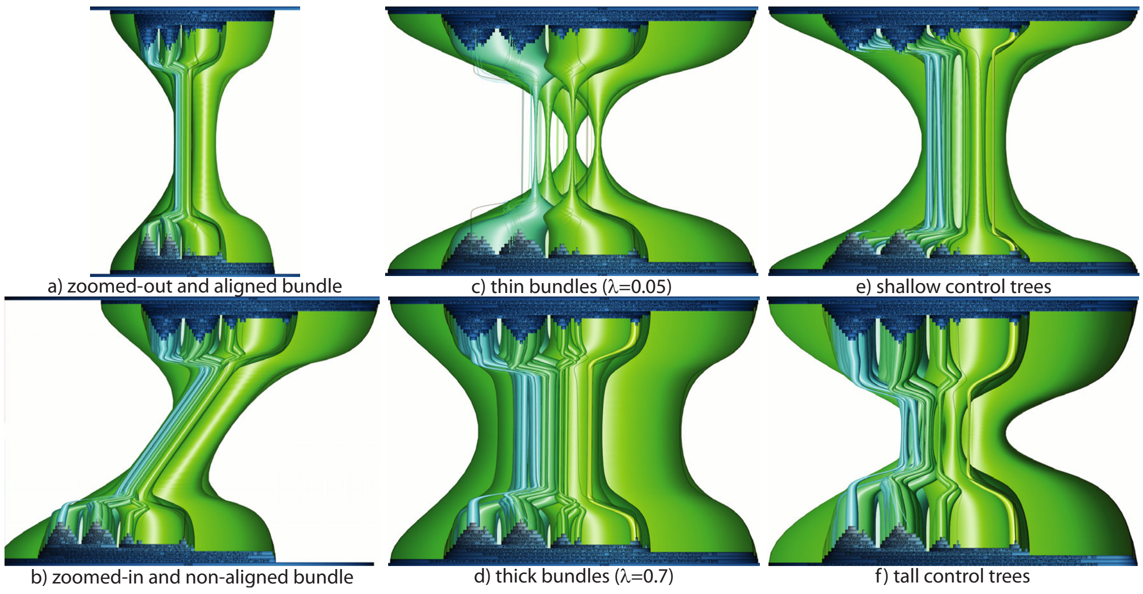


Figure 5.12: *Bundling parameters. The resulting bundle shapes are stable and readable for various zoom, pan, and tube shape values.*

and – by default – colors the bar based on δ using a red-gray-green color map. Red bars indicate matches from the current trace to the *past* of the other trace; gray bars show matches between the two traces which are *aligned* in time; and green bars show matches from the current trace to the *future* of the other trace. When hovering the mouse over a call f, the background of all bars which have matches of calls in $S(f)$ are set to blue. For example, in Fig. 5.7, users see calls that appear compactly grouped in the focus region of T_A. The blue bars in T_B show that these calls have matches scattered over a large portion of T_B. Very few of these are in T_B's focus. Thus, to better examine these matches, users can now shift T_B's focus to the left.

As a result, low bars show execution areas which have no, or very low-similarity, matches in the other trace. These areas are likely less interesting for further analysis. High gray bars indicate areas which have well-aligned matches. High red or green bars indicate areas which are executed at different time instants in the other trace, which are arguably the most interesting to analyze.

5.2.5.2 Match Visualization

A key use-case for attribute mapping in the match visualization is to *explain* the computed matches: Given two matched stacks, connected by HEB curves or tubes, we want to know *why* the stacks are considered similar and *where* they differ. TraceDiff addresses these questions as follows.

Finding permutations As outlined earlier, the match computation is insensitive to permutations. This is desirable for discovering stacks that match regardless of call order. However, permutations mean important execution-order differences that should be highlighted. Visually detecting small-scale permutations can be hard using the tube metaphor only. Given a match $k = (f, g)$ with call start times $t^s(f)$ and $t^s(g)$ respectively, this problem is addressed by mapping the difference $|t^s(f) - t^s(g)|$ to the saturation of a base color (red) and using the resulting color for the HEB curves or tubes. Fig. 5.13 shows the result: Matches with similar starting times show up as gray. Matches with different starting times appear as red. In this example, call A from the beginning of trace T_1 matches B at the end of trace T_2, and call C from the end of T_1 matches three times (D, E, F) at the beginning of T_2.

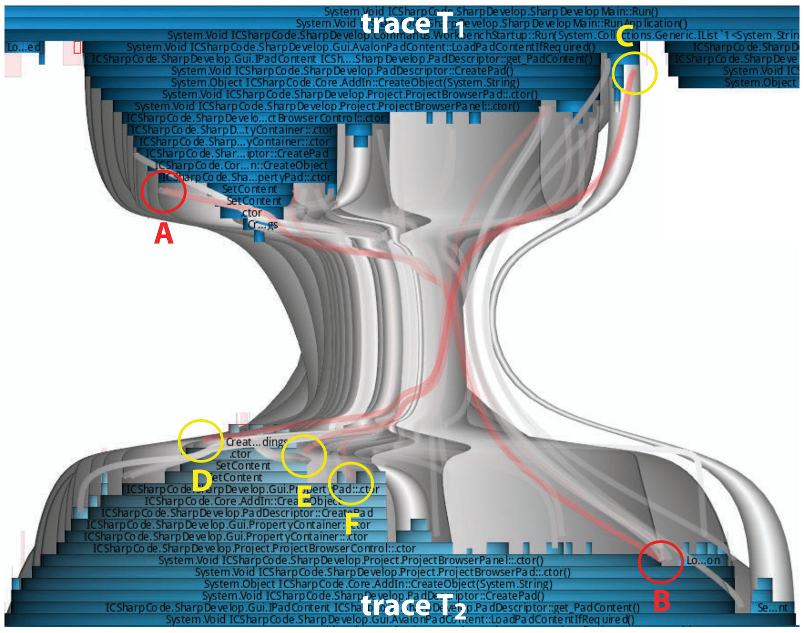


Figure 5.13: *Finding execution permutations.*

Finding trace-centric outliers A generalization of the above use-case is to show whether a call f in a stack T_A occurs at the same relative position (with respect to T_A's root) as its match g in a stack T_B. To show this, the partition of the traces into groups (Section 5.2.2) can be used. For a group G, rooted at G_A and G_B respectively in the two traces, users first select a viewpoint, i.e., decide if they want to examine matches from the perspective of T_A or T_B. This can be done by moving the mouse cursor in the upper half, respectively lower half, of the match visualization. If users select the viewpoint of T_A, each match $k(f_A, f_B) \in G$ is next colored by the value $\frac{t^s(f_B)-t^s(G_B)}{t^e(G_B)-t^s(G_B)}$ using a rainbow (blue-to-red) colormap. The interpretation of this color mapping is as follows (see Fig. 5.14): Matched calls, which occur at the *same* relative moments in the two traces (with respect to the start times of the root calls G_A and G_B respectively), have a color that follows the rainbow gradient, i.e., are blue if they occur early, and red if they occur late. Calls $f_A \in T_A$ that occur relatively *later* in T_B or calls $f_B \in T_B$ that occur relatively later in T_A appear as red outliers on a cold (blue..green) background – see insets in Fig. 5.14. Similarly, calls in one trace that are matched at *earlier* moments in the other trace appear as cold outliers on a warm background.

Depicting similarity A final use-case is to show the similarities s of the detected matches (Section 5.2.2). This allows for further separating strong (relevant) matches from less relevant ones, i.e., further explain why two stacks are similar or not. For this, the similarity s of each HEB tube is mapped to its transparency or strength of a white specular highlight: For tubes thinner than 16 pixels, the mapping uses transparency, since these tubes are too thin to show a specular highlight. For thicker tubes, the mapping uses highlights, since these tubes must be opaque so that the nesting effect (Section 5.2.3.4, Fig. 5.10) is visible. Fig. 5.15 shows the result: Tubes with strong specular highlights, like the red tube to the right, stand out in the image, and indicate strong (important) matches. Diffusely

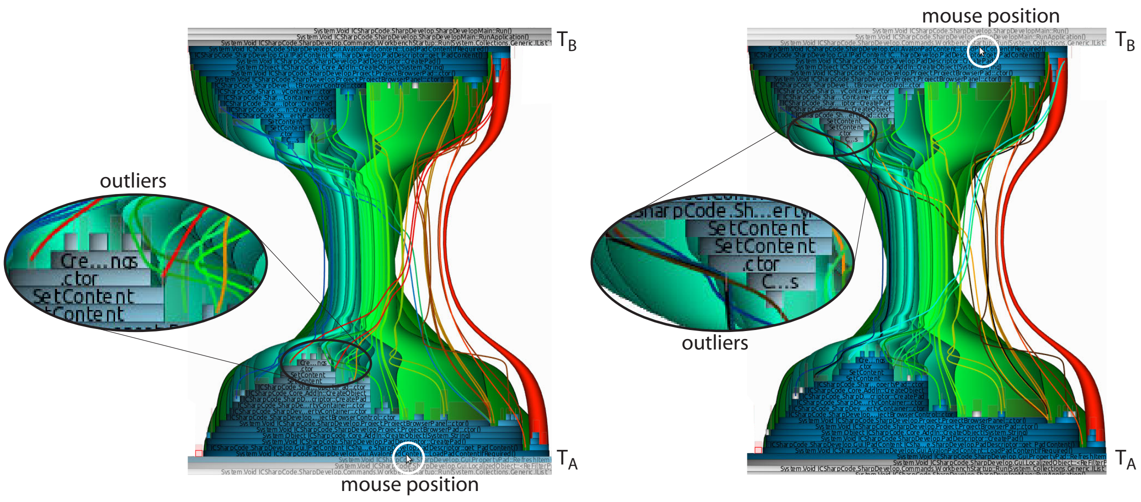


Figure 5.14: *Finding trace-centric outliers with respect to the bottom trace (left) and the top trace (right)*

shaded or half-transparent tubes attract less attention, which is in line with them being weak (unimportant) matches. For example, we see that all tubes that diagonally cross the image are both thin and half-transparent. This tells that the two compared traces are quite similar, the differences being relatively short-lived call stacks (thin tubes) which are permuted between the two traces (crossing tubes) and which are not strongly similar (transparent tubes). Encoding the similarity in specular highlights and transparency has the advantage that we still can use hue for showing other attributes, as described earlier.

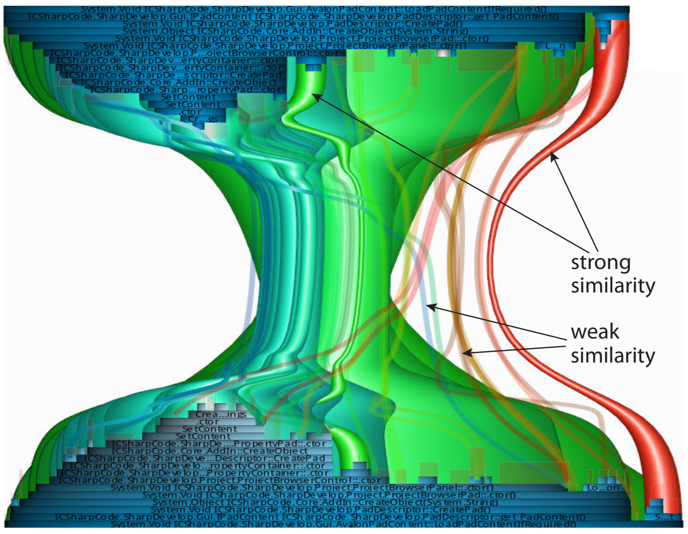


Figure 5.15: *Depicting match similarity.*

5.2.6 Application in Industry-Scale Software Systems

This section describes the usage of TRACEDIFF for the analysis of a large trace pair – approx. 150.000 calls and 1.500 function definitions. The two traces were recorded while an instrumented open-source C# IDE (*SharpDevelop*[1], approx. 1 million SLOC[2] in C#, 45

[1]`http://www.icsharpcode.net/`, last accessed 10/04/2013
[2]`http://www.ohloh.net/p/SharpDevelop/analyses/latest/`, last accessed 10/04/2013

contributers, 8 years of development) loads two different solution files, i.e., varying input data. As the same code is run twice, we expect to see strong overall correlation across the two traces. Figs. 5.16 a,b show a completely zoomed-out view of the compared traces. As we can see from the overview window, trace 1 (Figs. 5.16 c,d top) takes roughly a third of the execution time of trace 2 (Figs. 5.16 c,d bottom). Our questions thus are: Since these traces have significantly different lengths, do important similarities in the recorded executions yet exist? Where are these similarities, and which parts are different?

In the overview visualization (Figs. 5.16 a,b), we see that icicle-plot shapes for the two traces differ a lot. Hence, the two traces encode quite different dynamics in terms of call lengths and stack depth. Further inspection of the overview shows that trace 1 contains matches to trace 2 only within its first two-thirds (blue bars, overview top), while trace 2 contains matches over its full extent (blue bars, overview bottom). This is our first hint that the traces contain similar execution patterns.

Next, we want more insight into these patterns. For this, we focus on the first two-thirds of trace 1 and on the entire trace 2. When we look at the match visualization, we see that there exist quite a number of execution similarities. At the coarsest level, we identify five match groups ($A - E$). Three such groups (C, D, E) account for relatively short-duration sequences. The remaining two groups (A, B) account, together, for over 50% of the execution. Also, we discover that there are no matches between the begin and end phases of the two traces. We next use the permutation colormap (Fig. 5.16 c) to examine groups A and B, and quickly see several saturated red lines appearing: These are matched execution fragments that occur at different moments in their respective match groups. In group A, we see that the first phase of trace 1 (small bundle A') matches very well the last phase of trace 2 – the executed pattern has been shifted between the two traces. In group B, we find a more complex pattern: The first phase of trace 2 matches a large interval of trace 1 – the red lines in group B are concentrated at the bottom but fan-out at the top. Hence, the first phase of trace 2 has been spread over the whole execution of trace 1.

To learn more about the discovered matches, we now apply the trace-centric color mapping (Fig. 5.16 d), and move the mouse into trace 2. If we look at the color gradient in group A (trace 2), we spot several outliers (permutations) of the standard blue-to-red colormap (white markers, Fig. 5.16 d). These are functions that are called relatively earlier (blue lines) or relatively later (red lines) in trace 1. To see where these calls match in trace 1, we can visually follow the line colors from bottom to top. This outlines a second usage of our color mapping: Besides identifying time-shift outliers, colors help in following correspondences between the two traces. In contrast, the color gradient in group B (trace 2) does not show such interruptions of the rainbow pattern, which smoothly goes from blue (left) to red (right). Hence, the matches in group B indicate that the execution order between the two traces is preserved. A second difference between groups A and B is visible: At the right of A, we see a shiny orange tube (T, Fig. 5.16 d). This indicates that the last part of the sequences described by group A has a very strong similarity. We see no such shiny tubes in group B. This shows that the execution of A contains much stronger similarities than the execution of B. Finally, if we look at the last part of the matched sequences in group B (red lines), we see that these lines have a large vertical spread, both in trace 1 and 2 (dotted markers, Fig. 5.16 d). Hence, the last parts of these matched sequences occur over a short period of time (narrow red bundle) and deep call stack (large vertical spread). The entire analysis described above took around five minutes.

Finally, we note the two-fold added value of aggregating small-duration calls (Section 4.2.2.2): Besides a) reducing high-frequency portions in the icicle-plot representation, b) we can see that there are several such tall rectangles, which have around 30% of the height of the match visualization. If the visualization did not perform the call aggregation, there would be very little, if any, vertical space between the two traces in which to draw the bundles, and this would lead to an unreadable match visualization. The aggregation and subsequent modified HEB layout creates sufficient vertical space for the bundle visualization.

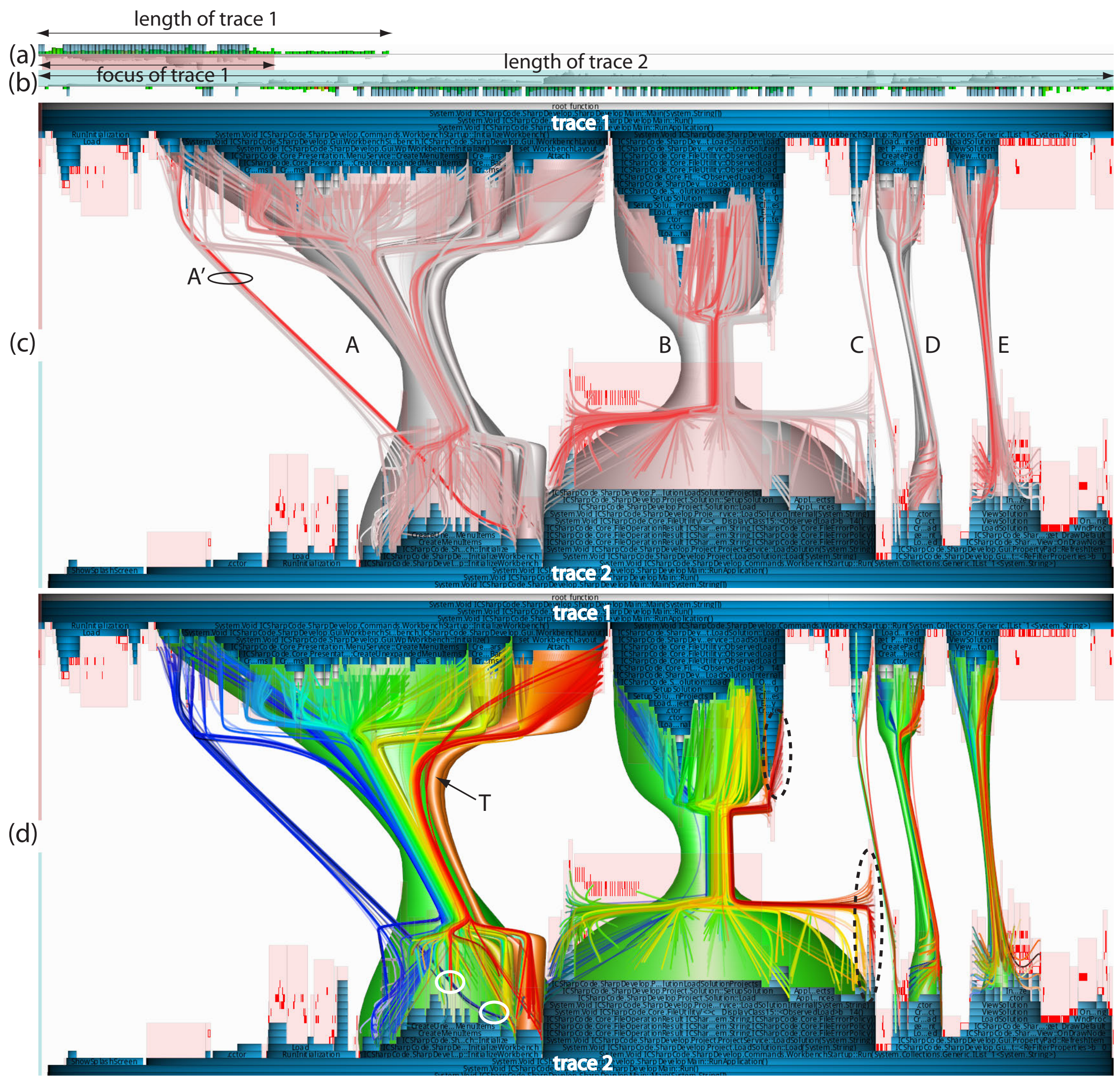


Figure 5.16: *Analysis for a large trace-pair: a,b) Overviews of traces 1 and 2; c) Finding coarse-level matched sequences and their permutations; d) Finding trace-centric time shifts.*

5.2.7 Discussion

Generality The correspondence visualization, though demonstrated on traces, works for any hierarchical sequence comparison for which match data is available. The matches are not restricted in any way, i.e., they can be many-to-many matches on any level in the hierarchy. Thus, TRACEDIFF can for instance also be used to correlate two hierarchies representing static system structures. In this case, the data would not be time-dependent and the width of the icicle-plot nodes could be set to have uniform width – or to represent a metric different to time.

Visual scalability An enhanced HEB technique eliminates the visual artifacts created by the original HEB. By combining this with a multiscale correspondence visualization, additional attributes, such as the width of matched elements, can be encoded in the correspondences. All in all, this allows for visually comparing hundreds of thousands of calls in two traces at interactive rates.

Ease of Use The interaction techniques allow for easy user input; to explore the underlying trace and match data, users only need to learn how to point, click, zoom, and pan. Brushing techniques implicitly translate point actions into selections both on the call stacks and the correspondence visualization.

Flexibility By these inputs, users can easily adjust the analyzed subset of the data and adjust level of detail. The multiscale correspondence visualization automatically adapts to the selected level of detail. The user-configurable color mappings in the overview visualization as well as the match visualization address specific questions in the given context of the given trace-comparison use case. For instance, color in the overview's bar charts could also be used to specifically highlight permutations.

Limitations While the visual design is definitely more scalable than those of other techniques, such as HEB or Code Flows, it will as well create clutter for very large hierarchical sequences and numerous many-to-many matches.

Benefits TraceDiff proposes several novel interactive visualization techniques for the analysis of the similarity of large execution traces. It addresses visual scalability and readability by introducing a modified hierarchical edge bundling layout and extending edge bundles to shaded tube bundles. The tube bundles enable visualizing the time-extents of execution patterns, and also explain execution matches by multiscale nesting. Attribute mapping to colors and highlights further adds similarity information and also assists finding execution permutations and time shifts. The technique is demonstrated on the comparison of two large execution traces.

Chapter 6

Analysis of Software Structure and Related Artifacts

This chapter is based on the following publication(s):

- Jonas Trümper, Martin Beck, and Jürgen Döllner. A Visual Analysis Approach to Support Perfective Software Maintenance. In Proceedings of the International Conference on Information Visualisation, pages 308–315. IEEE, 2012. [302]
- Martin Beck, Jonas Trümper, and Jürgen Döllner. A Visual Analysis and Design Tool for Planning Software Reengineerings. In Proceedings of the International Workshop on Visualizing Software for Understanding and Analysis, pages 54-61. IEEE, 2011. [294]

While analysis of behavior mostly deals with dynamic aspects, structural analysis focuses on static aspects of development artifacts such as system structure, versioning information from software repositories, design documents, and issue tracking data. Similarly to dynamic analysis, structural analysis is non-trivial, most prominently because of the complexity and size of the data involved.

The techniques presented in the previous Chapters 4 and 5 mainly target software engineers and architects as audience by providing technical views. This chapter presents two techniques that target both technical as well as non-technical staff in software projects. Among others, the techniques' goal is to bridge the communication gap (Section 2.3.2) between those two sides.

One evident problem here is that non-technical staff has problems inferring potential code quality problems upfront [33], which are often known by and visible to engineers, but hard to communicate and often invisible to others (cf. Section 2.2 and Section 2.3.2). One way to achieve better transparency in this respect is to use structural analysis for evaluating high-level facts, such as software metrics, in their structural context [33]. Based on this, even non-technical staff can infer such potential problems upfront. This chapter presents techniques which allow for more comprehensive structural analysis (3D-CBV, Section 6.2) and for (subsequently) restructuring software systems (D+D-CBV, Section 6.3) to address found problems, which is a hard task in itself [35, 226, 241]. In software analysis, this more comprehensive structural analysis helps extending the types of code-quality problems, which can be found.

Another problem is to define and use adequate levels of coordination in projects: Coordination typically needs to reflect technical dependencies between software components, which is hard to achieve in practice [104, 245]. Insufficient coordination of development on technically dependent components are known to lessen development productivity and increase software failures [51]. Dependency analysis is thus also an important means for project management, apart from its use for the analysis of change impact and of restructuring possibilities [35, 261].

6.1 Circular Bundle View (CBV)

The following sections build upon the CBV visualization method [123]. In the literature, this concept has many names, including "radial view", "radial plot" or – short – "bundle view". A CBV is constructed from a compound directed graph $\mathbb{S}$ (cf. Section 2.1.5). Its hierarchy representation, which can be thought of as a circular icicle plot, is created by recursively segmenting concentric rings. Fig. 6.1 shows an example of such graph depicted as rooted tree (left, a1, a2) and its respective CBV representation (right, b1, b2). Each ring segment represents a node in the input hierarchy H (b1 in Fig. 6.1(b)). Higher-level nodes are located on the outer ring segments and lower-level nodes in inner ring segments – leaf-nodes on the innermost ring. Non-hierarchical relationships R (a2 in Fig. 6.1(a)) on leaf-nodes are then depicted as HEBs in the inner circle (b2 in Fig. 6.1(b)). The root node (red stroked node in a1, Fig. 6.1(a)) is typically implicit and invisible in the CBV representation.

For structures with attributed nodes (cf. Section 2.1.5) – multivariate structures – the attributes are commonly mapped to the visual variables size and color of ring segments.

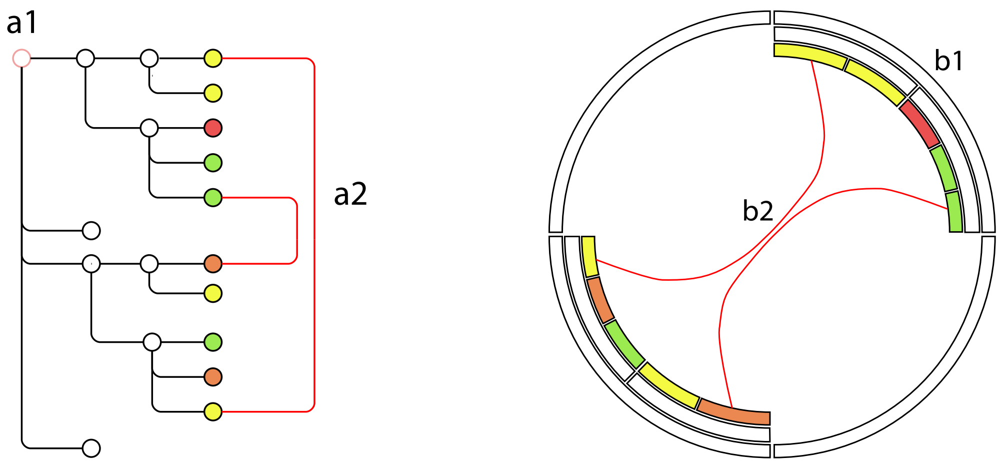


Figure 6.1: *Structure depicted as rooted tree (tree a1, relationships a2) and as CBV (b1, b2). The root node (red stroked in a1) is typically omitted in the CBV representation.*

6.2 3D-CBV: 3-Dimensional Circular Bundle View

In many cases, including perfective maintenance, the subject structure is multivariate and includes several attributes per node. The original CBV, however, is limited in that it can only be used to depict two attributes per node in parallel. The technique presented here adds two main extensions to the original CBV:

- The use of the third dimension as visual variable.
- Interaction techniques for iteratively exploring $\mathbb{S}$ and node attributes.

6.2.1 Extended Multivariate Analysis

Similar to 3D treemaps [29], the 3-dimensional CBV (3D-CBV) presented here adds the third dimension to the original CBV (Fig. 6.2(a)) by extruding the ring segments (Fig. 6.2(b)). So, an additional visual variable – the *height* of ring segments – can be used for the representation of multivariate data. By being able to integrate more data dimensions in a single view than with the original CBV, less context switches to other views are required for users to assess the same amount of data dimensions. With this, the 3D-CBV can in fact alleviate the need for an additional linked view.

Compared to using other visual variables such as texture, saturation, and luminance in the original CBV, the advantage of the extension to 3-dimensional space is that height can be better visually distinguished from the already used variable color. Some of those other variables – e.g., texture – may also interfere with superimposed text labels, rendering the latter unreadable.

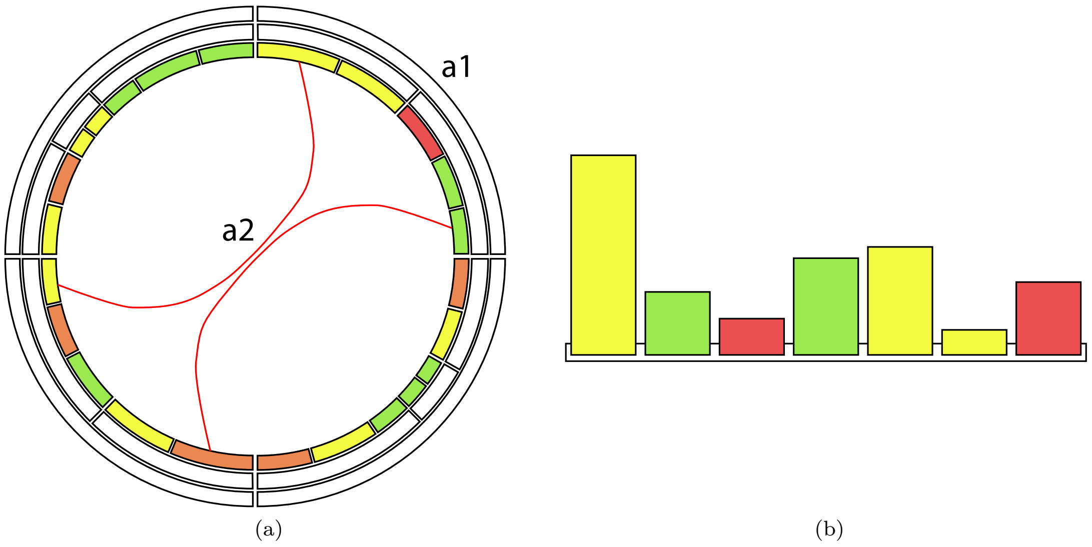


Figure 6.2: *Conceptual sketch: (a) Top view onto a 'conventional' circular bundle view with hierarchical relationships projected to the outer circles (a1) and non-hierarchical ones in the inner circle (a2). (b) Side view, metric values projected to node height.*

Humans generally compare three-dimensional objects – if there is no particular context – by their volumes instead of by their three separate dimensions [266]. Here, however, the context defines that the dimensions width and height of a 3-dimensional ring element have a distinct meaning and can thus be taken into account separately when comparing two objects. Adding height, however, introduces an occlusion problem: Strongly extruded nodes may occlude child nodes or even relationship edges in the HEB plot (b2 in Fig. 6.1(b)). For the case of software analysis, 3D-CBV addresses this by (1) only mapping values to a node's height on the innermost ring. Here, the innermost nodes are more important than nodes on the outer rings – as finally the leaf nodes represent the software artifacts containing a system's implementation. The higher-level nodes (outer rings) 'only' structure those artifacts. Moreover, occlusion is addressed by (2) making nodes semi-transparent when users hover those nodes or other nodes connected by an edge.

To be able to analyze different hierarchy levels concurrently and to compare attribute values across these different levels, attribute values of all *visible* nodes on the innermost ring are normalized.

The ExtraVis CBV implementation [67] added collapsing of nodes by hiding lower-level ring segments and aggregating dependencies to and from them. However, it routes all incoming and outgoing dependencies to the center of a collapsed ring segment, which hinders assessing the distribution of dependencies to hidden ring segments. To alleviate this, 3D-CBV fans out all dependencies across the full angular size of collapsed ring segments, routing each dependency to where the respective hidden segment would be located.

6.2.2 Techniques for Interactive Exploration of CBVs

In the analysis of large structures, users first need to gain an overview of such structure before being confronted with highly-detailed views. This section presents interaction techniques to implement a drill-down approach for CBVs. Initially, the extended CBV thus shows only a few top levels of the hierarchy H. For the subsequent iterative exploration the following interactions are possible:

Drill-Down Operations

1. *Collapse/expand node* n enables users to perform level-wise drill down into a subtree rooted at n, i.e., additionally shows/hides all direct children of n.

2. *Collapse/expand per depth level* allows for step-wise analysis per level of a hierarchy, i.e., additionally shows/hides all direct children of nodes $\{n_i\}$ parented in the currently deepest visible level. Descending one level in this way shows deeper nested nodes while still retaining an overview of present dependencies.

3. *Restrict to subtree rooted at node* n takes n as input and reduces the depicted hierarchy to direct and indirect children of n for detailed analysis. While reducing visual clutter and helping to focus on details of the subtree, it also hides dependencies to and from nodes outside of the selected subtree.

Search, Filter, Collapse, and Expand Search and filter mechanisms (left in Fig. 6.10) enable to view distinct aspects of a structure and to quickly find specific nodes. Searching for text fragments in node identities $\{N\}$ (cf. node definition in Section 2.1.5) highlights relevant ring segments in the visualization and clicking on them reveals further detailed information. The visualization of the structure can further be restricted to only a subtree and its descendants. Only nodes and relationships within this subtree are displayed.

6.3 D+D-CBV: Techniques for the Interactive Manipulation of CBVs

Not only are larger structures generally hard to present visually, the task of modifying system structures is faced with several additional challenges. In many cases, a modification is first planned, then its change impact estimated, and finally applied. For example, a typical reengineering scenario [77] can be decomposed into the following steps (see Fig. 6.3): (S1) Assess and understand the as-is *source design* (program comprehension, reverse engineering), (S2) identify potential problems in the source design (*design flaws*), (S3) plan the desired *target design* (*forward engineering*), and (S4) determine a set of actions to transform the source design into the target design.

We here see that the act of modifying a complex system structure also includes *planning* a set of change operations, which all in all constitute a modification of the

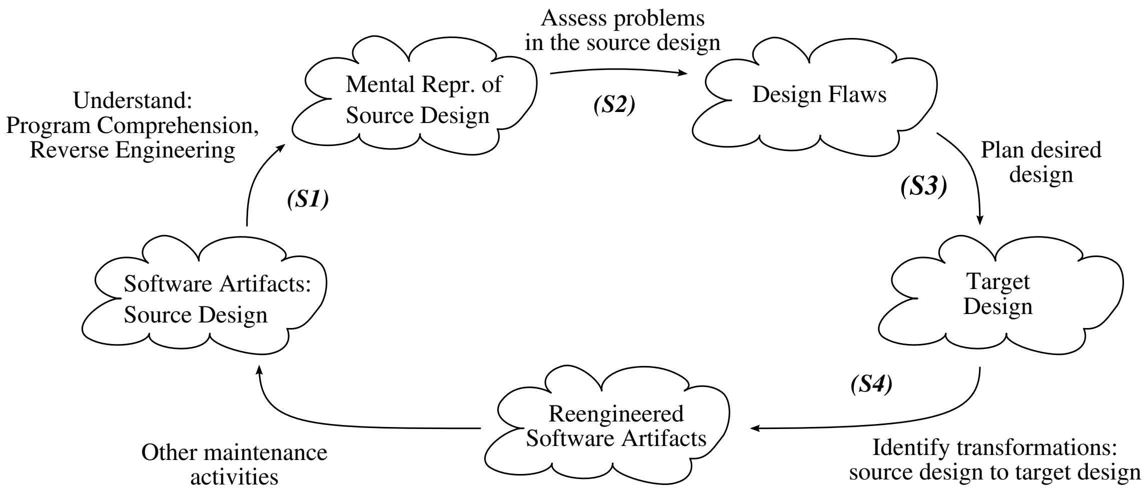


Figure 6.3: *Typical reengineering scenario within the software-maintenance cycle: (S1) Understand the source design of a software system, (S2) assess design flaws therein and (S3) plan modifications to the design, and (S4) transform a system's artifacts from the source design into the desired target design. Finally, other maintenance activities involve modifications to the system's artifacts (and design) that potentially introduce new design flaws.*

structure. Applying the changes is often connected to a significant effort, so engineers rather determine a suitable target design iteratively before applying any change. Then, however, memorizing a larger set of changes to transform the source design into the target design is non-trivial. Moreover, the target design has to obey certain constraints such as a proper layering of system components. This section therefore presents a set of techniques – Drag&Drop-CBV (D+D-CBV) – for virtually planning modifications to a given structure $\mathbb{S}$.

Formulating Changes Using Drag&Drop By using drag&drop, users can create and modify their plans to reorganize a structure. Fig. 6.4 illustrates moving a single node n to its new place in the tree, i.e., changing its parent p. While moving, relationships follow the dragged node and the layout changes accordingly as n leaves its old and approaches its new parent. When users choose to move a non-leaf node, the whole subtree $S(n)$ is moved along with n. Furthermore, selection of multiple nodes enables users to move more than one node at a time.

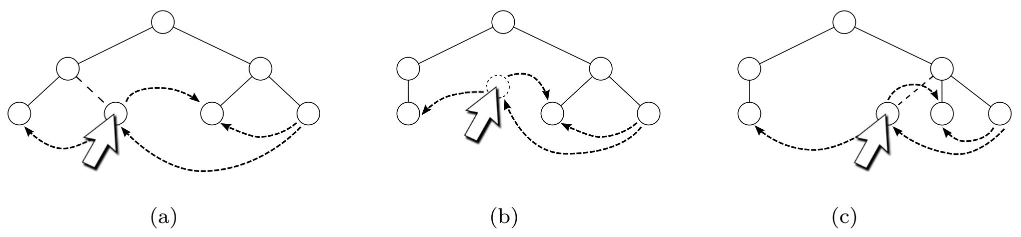


Figure 6.4: *Drag&drop concept illustrated using a rooted-tree layout: (a) When users start to drag a node, the representing ring segment is detached from its parent and the layout recomputed. (b) While moving the node, its incoming and outgoing relationships change as well and stay connected to the detached node. (c) Dropping the node to a new place reinserts it into the hierarchy and, again, recomputes the layout.*

Creation and removal of nodes enable users to further modify the structure's tree H. New nodes provide a containment where users can drag nodes to. By this, subtrees can be merged.

When moving modules across a structure, users may create relationships which constitute violations of certain structural properties such as directly connecting two subtrees. Fig. 6.5 shows how to dissolve such unwanted configurations: Users grab a relationships in its middle and drag it to a third node. Next, the tool splits up the relationship edge into two new ones, introducing an indirection. In terms of software engineering, this could for instance represent a callback mechanism in a utility library.

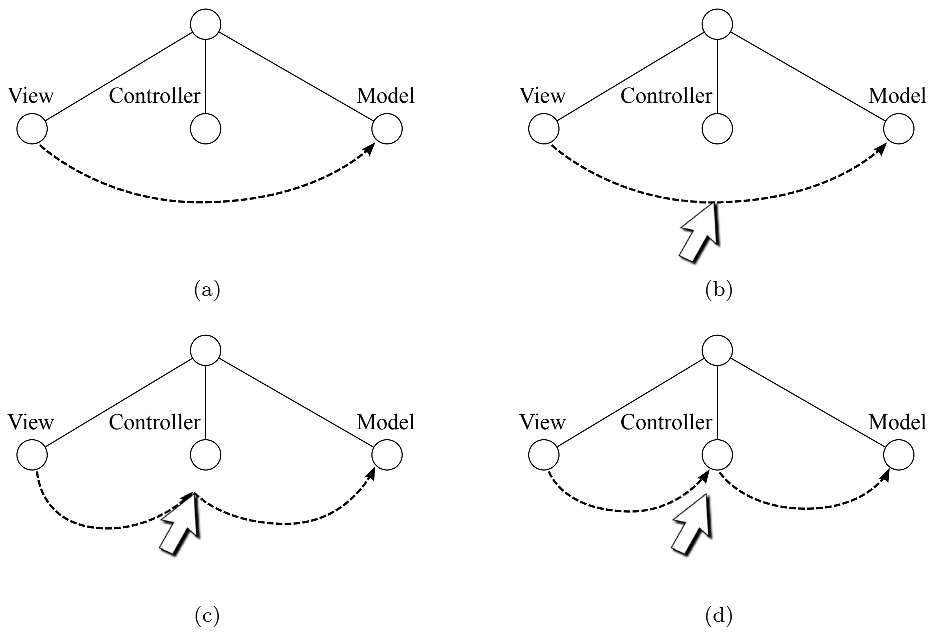


Figure 6.5: *Dependency splitting: (a) A dependency violates the intended model-view-controller architecture. (b) The dependency edge is grabbed in the middle. (c) When starting to drag the dependency edge, it is split into two separated edges. (d) Both new dependency edges are dropped onto the controller module.*

Another way of dissolving such cross-tree relationships $n_i, n_j)$ is to grab them in front of a node, e.g., n_i. If the ring segment representing n_i is collapsed, dragging the relationship edge pulls out every child node of n_i that has a relationship to n_j. Otherwise, the node itself is moved similar to the drag and drop mechanism above.

Operation History After finishing all virtual modifications to a structure, users are presented a list of intended changes – the operation history (upper right in Fig. 6.10 on page 114; Fig. 6.15 on page 117), e.g., "Move module A to module B". The tool automatically removes intermediate steps and reverted actions. Using this compressed task list, users can estimate effort requirements for planned modifications.

Selective Visualization of Relationships For the interactive discovery of structural constraints, the concept provides three visualization modes for relationships. In the first mode, all relationships are shown. The second mode shows only relationships related to specific nodes n, i.e., their incoming and outgoing dependencies. By hovering and clicking ring segments, users select nodes of interest. If n is a non-leaf node, all relationships ending or starting in $S(n)$ are shown. Other nodes that are part of those relationships are highlighted and their labels are enlarged. A third mode extends the second mode by also showing indirect relationships.

All three modes share the ability to filter visible relationship edges by incoming or outgoing relations. Thereby, users can concentrate on identifying constraint violations. In the case of software engineering, an example would be a system library that uses modules from higher architectural layers. Emphasizing relationship edges which cross subtrees, points users at possibly forbidden relations. A measure for these cross-border relationships is the distance between source and destination modules in the hierarchy graph when navigating through their least common ancestor. Additionally, each mode enables users to hover nodes with visible relationship edges and, thereby, highlight these relationships. This way, users can distinguish them from other relationship edges. To inspect single edges whose source, destination, and path may be occluded, users can highlight edges by hovering.

6.4 Application in Industry-Scale Software Systems

This section first presents two application examples for the CBV extensions in static analysis. The 3D-CBV is applied in perfective maintenance (Section 6.4.1), and D+D-CBV is used in a reengineering scenario (Section 6.4.2). Second, potential applications in dynamic analysis are outlined (Section 6.5).

6.4.1 PerfectMaintenance: Prototype Tool Based on 3D-CBV

The 3D-CBV technique is implemented as a prototype tool named PerfectMaintenance and evaluated by applying it to large-scale, grown software systems. It is applied in two scenarios: First, analyzing maintainability of a code base is discussed in detail and, second, identifying unused code is outlined briefly.

6.4.1.1 Francotyp Postalia GmbH: Analyzing Maintainability

As outlined in Section 2.3.2, management personnel often faces dilemmas in perfective maintenance such as: Where and how to invest the limited resources for an optimal positive effect on future maintainability? The given resources typically do not suffice to fix all quality issues in the code. In addition, the most prominent quality issue may well be the hardest to fix. So responsible personnel will carefully consider the pros and cons of any action that refers to code quality before it is undertaken. They will only consider approving such action if its chance to increase or ensure future maintainability outweighs its costs. Yet, weighing these pros and cons in practice is typically based on subjective judgment obtained by experience, discussions with engineers, etc. Interestingly, there is generally a lack of specialized visualization techniques and tools that address this problem.

Assuming any actions can be identified by engineers and architects, approval by other stakeholders, namely management, is still difficult to obtain. For it, this section presents the application of 3D-CBV to help identify software artifacts that impose a high risk by causing increased future maintenance efforts. It further enables exploration, analysis and planning of preventive measures to artifacts with maintainability problems that should be tackled first. Insofar, this application provides clearly understandable *indications* within a single view. In particular, it serves to bridge the communication gap between technical and non-technical staff in software projects.

The technique was evaluated by applying the prototype tool (Fig. 6.6) to an industrially developed software system. Francotyp Postalia GmbH[1] (*FP*) is one of the world's largest vendors for franking machines. We analyzed an embedded software system for one of their machines. This software system is written in C/C++ and consists of approximately 900k SLOC with an average of 10 engineers working on it for more than two years. While most of its development takes place internally, some modules and libraries were purchased from third-party vendors.

Classification in a Task-Oriented Model In the task-oriented model by Maletic et al., this application scenario can be characterized as follows: The *task* is to support users in assessing code quality and potential maintenance problems. More specifically, users want to assess (a) the static structure of a software system (its hierarchical composition of artifacts) and (b) the evolution of (quality-)related attributes of those artifacts. The *audience* includes software engineers, architects, and project managers who want to understand aspects related to software maintenance and code quality. The visualization *targets* static software structure and relationships between entities in this structure, software metrics, and change data. These data are *represented* using a circular view (for the structure), edge bundles (for the relationships), and bar charts (for the change information). Finally, the visualization *medium* consists of a standard screen with multiple linked views.

Exploration Workflow Users start by exploring an overview of the entire system structure in the main view (left in Fig. 6.6). By interacting with the main view, they can then interactively drill down into parts of the hierarchy depicted as circular view. At any point in the analysis, the settings window (right in Fig. 6.6) can be used to further tune the visualization: Sub-trees of the hierarchy depicted in the main view can be hidden via an indented-tree representation in the settings window; the mapping of attributes to visual variables can be (re)configured and specific relationships can be hidden. Via the timeline plot (bottom in Fig. 6.6), users can easily adjust the depicted time range to identify trends in the data.

Setup and Configuration The shown structure $\mathbb{S}$ corresponds to the system's file system hierarchy (as H) and static dependencies (as R). Node attributes represent file-related software metrics. These metrics are mapped ($\mapsto$) to visual variables of node representations as follows:

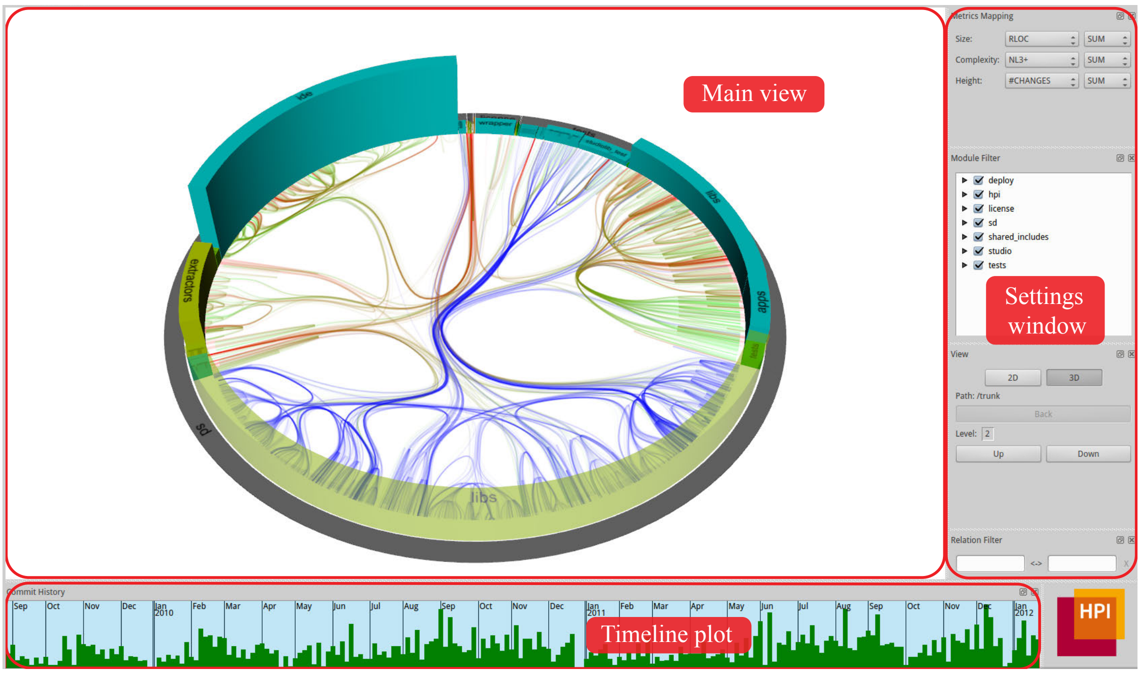
- *Code complexity* $\mapsto$ *module color (green to yellow to red):* is computed as ratio of number of statements in nesting level three and deeper, *NL3+* [if(...) { if(...) { while(...) { } } }], to SLOC. Deeply nested code is commonly considered to be hard to understand and, thus, a high percentage of deeply nested code implies high complexity.

- *Relation direction* $\mapsto$ *edge color (green: start, red: end)*: Similar to the color-coding of modules, potentially dangerous relations of a module (i.e., incoming - other modules rely on that module's implementation) visually stand out.

- *Change frequency* $\mapsto$ *module height:* includes any modifications applied to a file within the last 6 months according to a project's source-code management (SCM) system.

- *Size* $\mapsto$ *angular size of module:* is computed as SLOC.

[1] `http://www.francotyp.com/`, last accessed 01/25/2012



Metrics Mapping
Size: RLOC SUM
Complexity: NL3+ SUM
Height: #CHANGES SUM

Module Filter: ☑ deploy, ☑ hpi, ☑ license, ☑ sd, ☑ shared_includes, ☑ studio, ☑ tests

View: 2D [✓ 3D]
Path: /trunk
Level: 2

Figure 6.6: PerfectMaintenance*: View design.*

Thereby, users can easily identify large, complex modules that have recently been touched. By sizing up incoming and outgoing dependencies, we can assess a module's entanglement with the rest of the system. In the given use case, a common color scheme of the domain, i.e., the traffic-light metaphor, was adopted. Nevertheless, the scheme can be modified as necessary, e.g., to bypass deficiencies in human color perception [240].

Tool Use in Practice During *FP's* review sprints, their development teams typically perform quality-improving tasks such as refactorings or reengineerings. Engineers collect these tasks over time during their daily work and, at the beginning of each sprint, managers, project leaders, and engineers prioritize them together. However, this task list is not necessarily comprehensive because engineers tend to focus on code-centric improvements. By contrast, project leaders and managers want to concentrate on modules that are relevant for future maintainability but they lack detailed system knowledge. Thus, misunderstandings about relevance and priority of task items are common.

We have applied the tool to improve this situation and to support *FP's* managers and project leaders with understanding potential maintainability risks in their software systems. Fig. 6.7 illustrates an exemplary analysis process for their franking machine's software system using the 3D-CBV: We start by analyzing the software system at its uppermost hierarchy level. At the first level, (Fig. 6.7(a)), we can distinguish two modules: A large, complex module that contains product-specific code (1 in Fig. 6.7(a)) and a smaller, medium complex module that represents *FP's* product-independent libraries. While the former has been modified frequently, the library module has barely been touched, indicating mature code. However, comparing code sizes, we can see that the base library is very small as opposed to a large product-specific code base. There might be potential for increasing code reuse between different franking machines.

Descending another level within the software system's modular structure (Fig. 6.7(b)) reveals that the product-specific code is composed of multiple sub-projects, which depend on each other. However, predominant projects from a manager's point of view such as country-specific application projects are hardly visible here because they consist of only

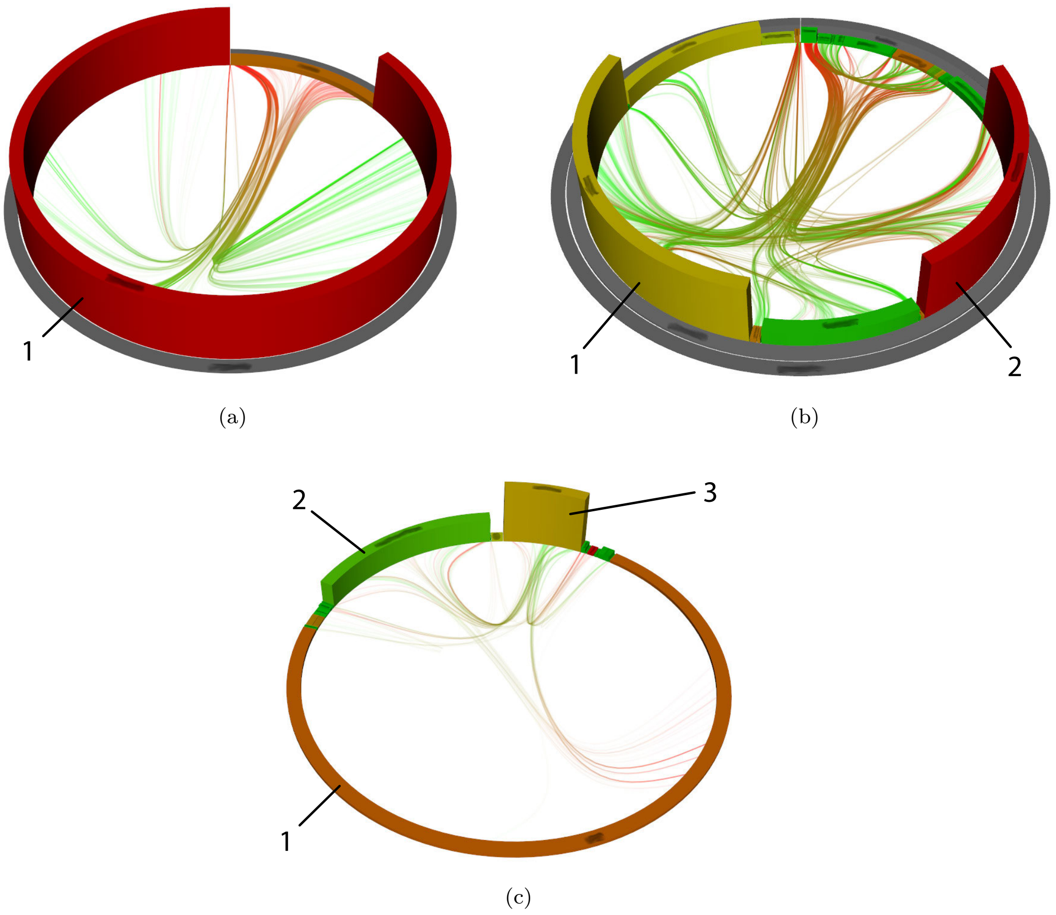


Figure 6.7: *Drilling down into the module hierarchy of* FP's *controlling software. (a, b) We descend in the hierarchy and (c) restrict the view to a subtree.*

a few lines of configuration code. Instead, visual focus lies on two large support projects (1 and 2 in Fig. 6.7(b)) that have been heavily modified and expose high complexity in relation to others.

To further analyze these modules, we investigate the more complex one first (Fig. 6.7(c)): We apply the "restrict to subtree" operation to this module, thereby hiding all other modules but the selected module and its submodules. Disturbing information about currently irrelevant modules is hidden and leaves additional space for submodules on the inner rings. Here, we can see a large module with medium complexity (1). However, it exhibits a relatively low change frequency. This module is a third-party library that takes care of display devices and manages the graphical user interface. Despite its high complexity and strong entanglement with the system, this module should not be a major subject to quality-improving measures because it is modified infrequently and, thus, seems to be in good shape.

Instead, there are two other modules visible that expose high change frequencies, medium and low complexity, and that are relatively large (2 and 3 in Fig. 6.7(c)). Both of them deal with the franking machine's user interface and control the franking machine's human interface. Bugs at this level will definitely affect customer satisfaction. Thus, they should be subject to perfective maintenance.

To check these two modules' entanglement with other modules, we switch back to an overview of the full software system with a hierarchy depth of three. We notice a strong entanglement of both modules (1 and 2 in Fig. 6.8), which indicates that other parts of the system are relying on them. Yet, this overview reveals other relevant modules: First, the largest module from Fig. 6.7(b) is separated into its submodules. An outlier among them exposes high complexity and medium change frequency (3).

Second, we can easily distinguish the two most complex modules within this view because their red color contrasts with surrounding green modules (4 in Fig. 6.8). Furthermore, they also have a strong system entanglement. However, their size in relation to other modules shown on this hierarchy level is rather small and they exhibit almost no modifications during the last six months. Nevertheless, they are both submodules of a printing module within *FP's* product-independent library, which renders them crucial for franking machines. Although the probability of future maintenance work within these modules seems to be low, we suggest further investigating why these libraries have such high complexity.

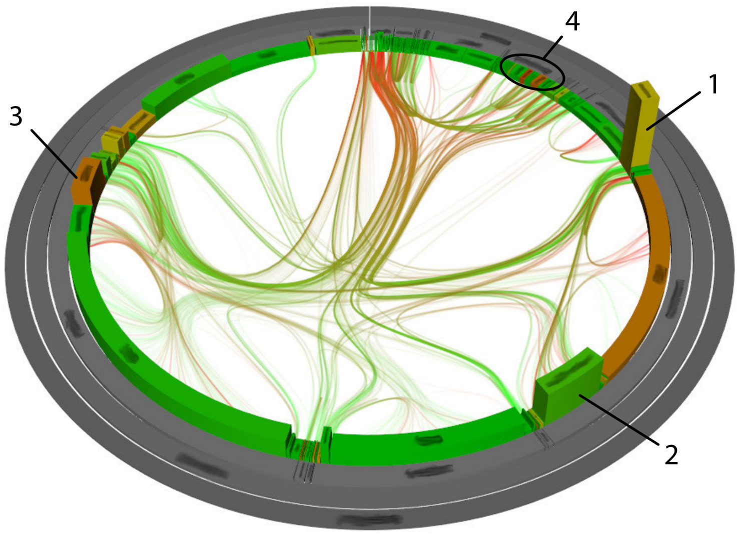


Figure 6.8: *An overview of* FP's *software system at hierarchy level three enables to identify outliers in change frequency and complexity.*

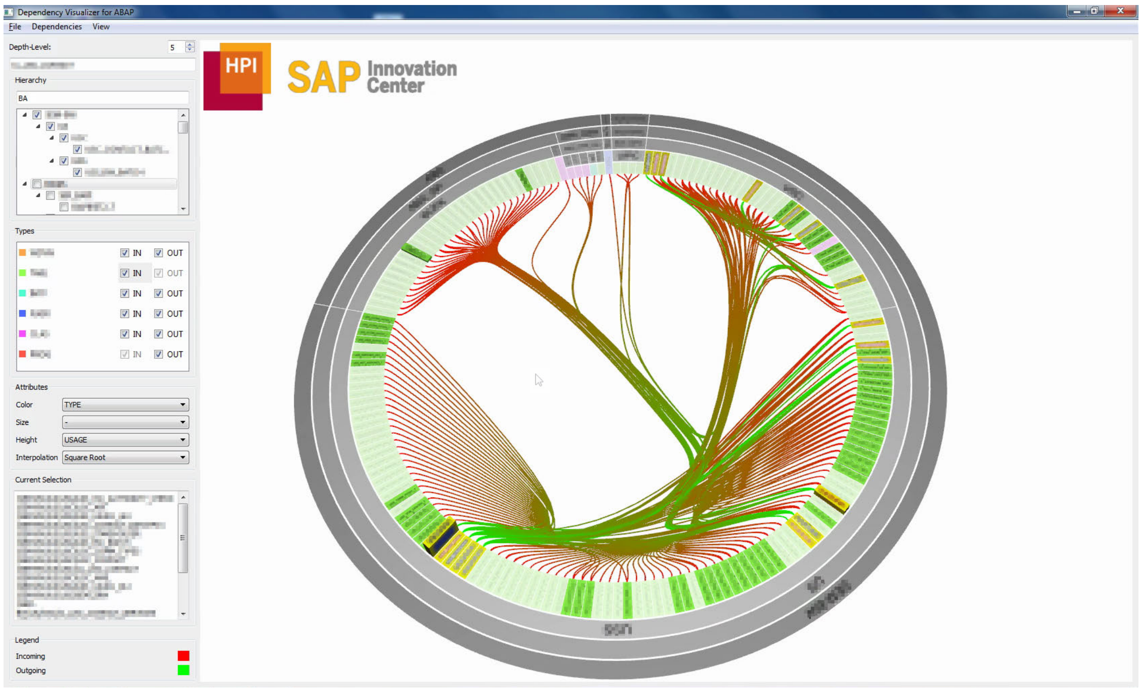
During our discussions with the development teams, project leaders, and managers, our images and interactive visualization tools proved as a viable vehicle for communication. By hiding irrelevant information such as very small modules we could focus on outliers with high complexity, strong entanglement with the system, and frequent modifications. Furthermore, in the 3D-CBV, the users had no problems comparing the distinct variable dimensions width and height of the three-dimensional ring elements separately. We still noticed a few wishes for enhancements that targeted supportive widgets and analysis functionality. Among others, theses included highlighting forbidden dependencies that violate the intended system architecture (which first requires a white- or blacklist for dependencies), more flexible undo/redo functionality, and support for storing and restoring view configurations across sessions.

Overall, visually combining multiple indicators, which are easy to explain and understand, helped the team gain new insights and encouraged further discussions: "emphasizes problems not recognized before".

6.4.1.2 SAP Innovation Center Potsdam: Identifying Unused Code

Enterprise systems in the ABAP-based ecosystem of the SAP AG comprise several million SLOC. In a component-based approach, they are combined in various configurations to implement specific work flows, e.g., in enterprise resource planning (ERP). For software architects, it is hard to tell whether this vast collection of components is used in any configuration. This is particularly true for legacy components that mainly exist to provide backward-compatible APIs in SAP's ERP products. The maintenance of these legacy components is becoming more and more complex with increasing age of each component. In this context, software architects of the SAP AG and the SAP Innovation Center Potsdam started a long-term effort to reduce the amount of legacy code in HANA-based ERP components by eventually removing unused legacy components. For it, they needed to be able to determine which of these components are actually used in any configuration.

In a collaborative project, the 3D-CBV was thus applied to analyze direct and indirect (transitive) component usages. The Dependency Analyzer for ABAP represents the overall system structure and usage relationships between components. Moreover, the height of ring segments is used as visual variable to quantify the number of usages of each component (Fig. 6.9).



Depth-Level: 5

Hierarchy

BA

Types

Attributes

Color TYPE

Size -

Height USAGE

Interpolation Square Root

Current Selection

Legend

Incoming 🟥

Outgoing 🟩

Figure 6.9: *An overview of a part of an SAP HANA-based ERP system showing component usages (edge bundles) and number of direct and indirect usages (height).*

The inclusion of indirect usages enabled the software architects to identify larger sets of interconnected legacy components that can be removed altogether. Moreover, by quantifying the number of usages, they could better estimate the implied change effort and change impact, as the removal of frequently used components requires changes in many other components.

6.4.2 VisualReengineering: Prototype Tool Based on D+D-CBV

Reengineering complex software systems represents a non-trivial process. As a fundamental technique in software engineering, reengineering includes (a) reverse engineering the as-is system design, (b) identifying a set of transformations to the design, and (c) applying these transformations. While methods a) and c) are widely supported by existing tools, identifying possible transformations to improve architectural quality is not well supported and, therefore, becomes increasingly complex in aged and large software systems.

Existing restructuring tools and techniques [157, 261, 278] can, in general, cope with complex systems by providing abstractions and visual representations. However, they typically ignore that more complex restructuring tasks, in particular in terms of reengineering, require a significant amount of exploratory work beforehand for finding a proper transformation of a system's design [151]. As a result, it is often impossible to determine reliable estimations of the required effort and the change impact of reengineering tasks, e.g., because of unknown dependencies.

VisualReengineering is a prototype tool implementation using D+D-CBV's interaction techniques. The tool aims at supporting software architects during reengineering tasks. It helps them identify a given software system's design and visually planning quality-improving changes to this design. Three applications to industrial software systems demonstrate usage and scalability of the approach.

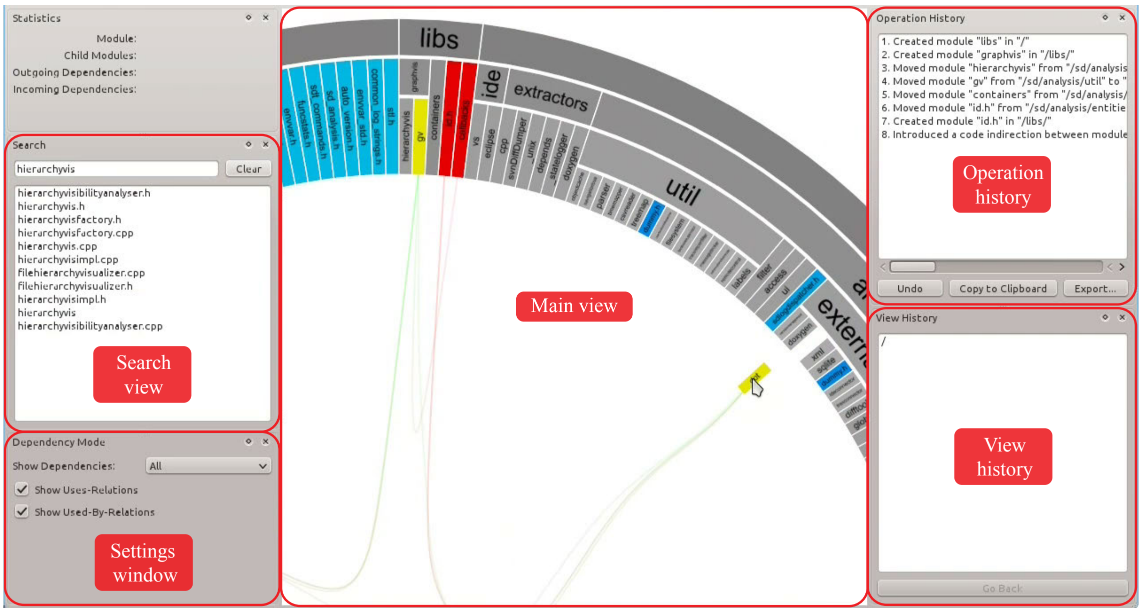
Classification in a Task-Oriented Model The application scenario can be categorized according to the task-oriented model by Maletic et al. as follows: The *task* is to help users in reengineering existing code bases of large software systems. In particular, the technique aims at (a) providing support for assessing the source design and identifying potential transformations of this design for improving its maintainability. Further, by (b) providing a list of action items for such transformations, it helps assessing the change impact and change effort connected to such reengineering. The *audience* includes software architects who want to restructure a given software system in the course of regular software maintenance. The visualization *targets* static software structure, i.e., a hierarchy of entities and relationships between entities in this hierarchy. These data are *represented* using a circular view (for the hierarchy) and edge bundles (for the relationships). Finally, the visualization *medium* consists of a standard screen with multiple linked views.

Exploration Workflow VisualReengineering (Fig. 6.10) aims at supporting software architects in the two ways previously outlined (cf. Section 6.4.2). It consists of a visualization depicting a software system's structure with embedded inner relationships (*main view*), as well as interaction metaphors enabling virtual manipulation of software entities. Since the underlying source code is not changed, this concept facilitates experimentation and testing of reengineering hypotheses. Using the tool, architects can identify architecture violations, cluttered module dependencies or other design flaws. Visual manipulation enables them to plan alterations of a system's structure and to quickly see the results of planned modifications such as resolved indirect module dependencies. An *operation history* provides a persistent 'backlog' of performed actions. The *search view* and *view history* provide further navigation aids to quickly jump to points of interest. The *settings window* allows for hiding outgoing ("uses") or incoming ("used-by") dependencies.

6.4.2.1 Application to Industry-Size Software

This section discusses three application examples on industrially developed and maintained software systems. First, we analyze a reengineering scenario within *BRec*, a 3D building reconstruction software by virtualcitySYSTEMS GmbH[2]. Next, a second case study using *SD Studio* and *SD Developer Edition*'s code base is presented. Both software analysis

[2]`http://www.virtualcitysystems.com`, last accessed 5/10/2011



```
hierarchyvis
```

```
hierarchyvisibilityanalyser.h
hierarchyvis.h
hierarchyvisFactory.h
hierarchyvisFactory.cpp
hierarchyvis.cpp
hierarchyvisimpl.cpp
filehierarchyvisualizer.cpp
filehierarchyvisualizer.h
hierarchyvisimpl.h
hierarchyvis
hierarchyvisibilityanalyser.cpp
```

```
1. Created module "libs" in "/"
2. Created module "graphvis" in "/libs/"
3. Moved module "hierarchyvis" from "/sd/analysis
4. Moved module "gv" from "/sd/analysis/util" to "
5. Moved module "containers" from "/sd/analysis/
6. Moved module "id.h" from "/sd/analysis/entitie
7. Created module "id.h" in "/libs/"
8. Introduced a code indirection between module
```

```
/
```

Figure 6.10: VisualReengineering*: View design.*

and visualization tools are developed by Software Diagnostics Technologies GmbH[3] (*SD*). Finally, we describe a usage scenario in a legacy COBOL software system.

virtualcitySYSTEMS: Identifying Architectural Weaknesses *BRec*, the 3D building reconstruction software developed by virtualcitySYSTEMS GmbH, is maintained by 15 engineers on average and it consists of approximately 100k LOC written in C/C++. The task to accomplish for virtualcitySYSTEMS' system architect was to identify architectural weaknesses that can be quickly resolved. Using the prototype tool, he first saw an overview of the system and its dependencies depicting approximately 350 files and 1,500 relations between them (Fig. 6.11). A relation between two files indicates that one file (green line end) uses one or more functions and variables defined in the other file (red line end). In this view, he noticed a large amount of red (incoming) dependency edges connected to a single file within the *QtGUI* module. Source of these edges were files within the *Core* and *SGI-Port* modules. The architect saw this as an architectural weakness because core libraries should not depend on user interface modules. To identify the files violating this principle, he hovered over the file in the *QtGUI* module. Thereby, he highlighted the file itself, its dependent files and dependency edges in between. As nodes representing files were too small to be labeled in the overview, he zoomed in to reveal the respective file's name.

Fig. 6.12 shows the highlighted and zoomed *triangle.cpp* file that caused the architecture violation. The system architect wanted to estimate the amount of work required to correct this issue. He dragged the file to the *Core* module and watched the moving dependencies. He noticed no new architecture violation being introduced by the reorganization. A check of the modification list confirmed the cheapness of this restructuring in terms of required work.

Without usage of further metrics or formal architecture descriptions, the visual software-reengineering tool enabled virtualcitySYSTEMS' architect to recognize an architectural violation and to estimate the effort required for correcting this issue.

[3]`http://www.softwarediagnostics.com`, last accessed 5/10/2011

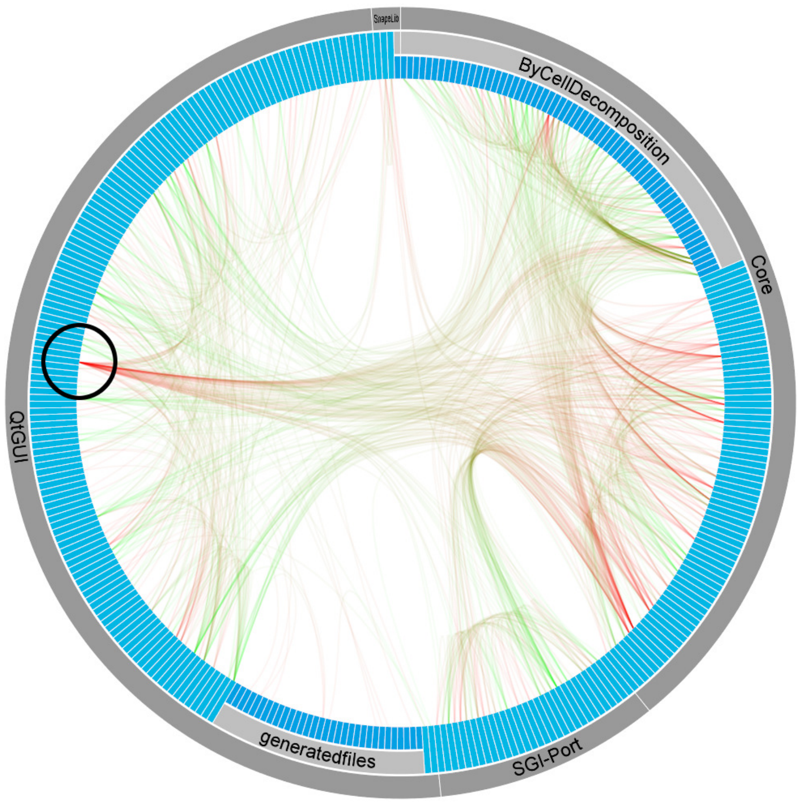


Figure 6.11: *Fully expanded dependency view of* BRec's *code base using CBV. Nodes on outer circle represent hierarchical structure and lines depict a usage relation. Grey nodes correspond to modules, blue nodes are files. A color gradient from green to red indicates a dependency's direction.*

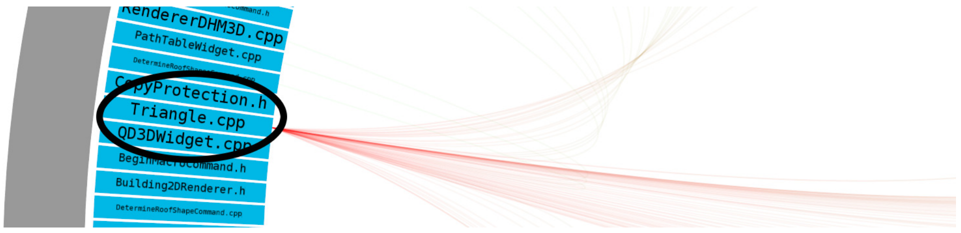


Figure 6.12: *Detailed view after zooming. By zooming to the file with many incoming red dependency edges, the architect found out that the* Triangle.cpp *file is violating the intended architecture.*

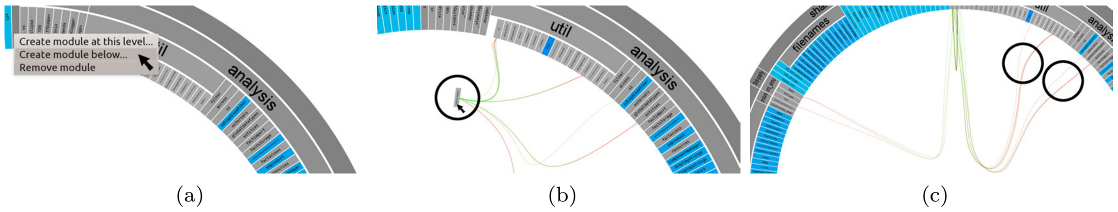
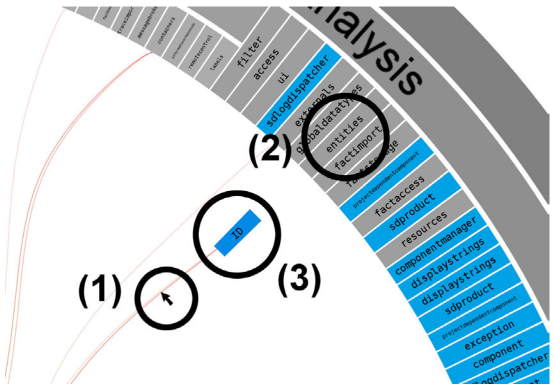
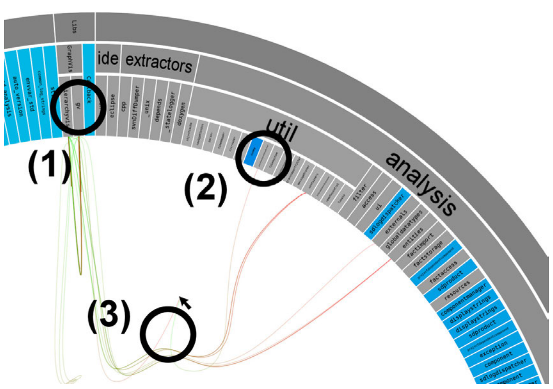


Figure 6.13: *Moving two modules to a new utility library: (a) Creation of a new utility-library subtree using a context menu. (b) Dragging the* HierarchyVis *module to its new place. Dependency edges change along with the dragged module. (c) Both modules have been moved to the utility library. New forbidden cross-module usage dependencies between the utility library and application-specific code have been introduced.*



(a)



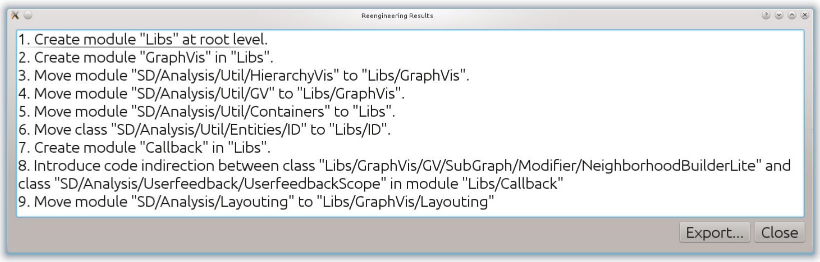
(b)

Figure 6.14: *Dissolving unwanted dependencies: (a) By grabbing the dependency edge (1) in front of the* Entities *module (2), the architect pulled out the* ID *module (3) that caused the unwanted dependency. (b)* Software Diagnostics' *system architect grabbed the dependency edge between the* GV *module (1) and the* Userfeedback *module (2) and, thereby, separated it into two new dependencies (3). Afterwards, he was able to group these new dependencies in the* Callback *module.*

Software Diagnostics: Decoupling Libraries *SD Developer Edition* is a software tracing and visualization tool that supports engineers with bug fixing and program comprehension. *SD Studio* provides visualization tools, such as software maps, to assess a software system's quality, source code evolution and development resources. Both tools share a code base that comprises approximately 230k LOC within 2,100 C++ files.

To improve reusability, Software Diagnostics' system architect planned to remove two graph visualization submodules from the application code-base and to put them into a new "graphvis" utility library. This included eliminating all dependencies from both modules to application-specific code. Based on experiences of *Software Diagnostic's* engineers, the architect expected forbidden dependencies. To avoid unknown risks, he decided to make an effort estimation prior to assigning the task to the engineers. Using our prototypical tool, the system architect visually explored the steps required to complete the reengineering. To concentrate on high-level modules, he first collapsed low-level layers. Fig. 6.13 illustrates his next three steps: First, he created a new container module for the two libraries to move (Fig. 6.13(a)). He dragged both libraries towards the container module and dropped them (Fig. 6.13(b)). During dragging, he already recognized several dependency edges indicating usage of application-specific code within both graph visualization modules: They used code from the application-specific modules *Entities*, *Containers*, *Userfeedback* and *Layouting* (Fig. 6.13(c)).

Relying on his knowledge of the affected modules, the architect used different strategies to dissolve these dependencies. For the *Containers* module, he decided to place it into the new container module as well, because it consists of low-level re-usable collection classes. To find out, which part of the *Entities* module was actually referenced by the graph visualization modules, the architect grabbed the dependency edge in front of the module and pulled out a module *ID* (Fig. 6.14(a)). As the *ID* module is low-level functionality as well, he moved it to the new utility library. To decouple the graph visualization libraries and the *Userfeedback* module, which is responsible for displaying progress dialogs to users, he created a new *Callback* utility library and grabbed the corresponding dependency edge in its middle. By dragging the grabbed edge to the *Callback* module, he divided the dependency into two new ones: Both, the graph visualization modules and the *Userfeedback* module used the new *Callback* library (Fig. 6.14(b)). To remove the last cross-border dependency to the *Layouting* module, the architect decided to put it directly into the new graph visualization library as layouting is a central task for graph visualization.



```
1. Create module "Libs" at root level.
2. Create module "GraphVis" in "Libs".
3. Move module "SD/Analysis/Util/HierarchyVis" to "Libs/GraphVis".
4. Move module "SD/Analysis/Util/GV" to "Libs/GraphVis".
5. Move module "SD/Analysis/Util/Containers" to "Libs".
6. Move class "SD/Analysis/Util/Entities/ID" to "Libs/ID".
7. Create module "Callback" in "Libs".
8. Introduce code indirection between class "Libs/GraphVis/GV/SubGraph/Modifier/NeighborhoodBuilderLite" and
class "SD/Analysis/Userfeedback/UserfeedbackScope" in module "Libs/Callback"
9. Move module "SD/Analysis/Layouting" to "Libs/GraphVis/Layouting"
```

Figure 6.15: *Dialog displaying a detailed task list for decoupling two modules from* Software Diagnostics' *application-specific code base. The list was generated based on the system architect's interactions with our prototypical tool.*

Having eliminated each forbidden dependency, Software Diagnostic's system architect retrieved a list of modifications from the tool (Fig. 6.15). Using this list, he estimated the effort required to fulfill the decoupling task and eventually decided to assign it to engineers.

The visualization technique helped to break down a generic reengineering task, i.e., decoupling of two graph visualization libraries from the application code-base, into clearly formulated subtasks without reading or changing any code. Risks such as unexpected dependencies and architecture violations could be estimated beforehand.

COBOL: Change Impact in Legacy Software Systems The subject software system of this application example is an excerpt of a confidential 100k LOC COBOL system from the banking industry (Fig. 6.16). It consists of approximately 1,100 sections in 150 programs and copybooks (include files). On the uppermost structure level, the visualization separates a copybook and a program section. Next, actual programs and copybooks follow. Finally, programs are further separated into sections based on their procedure division, i.e., a COBOL program's behavioral part.

Besides introducing new engineers to the code base using a visual representation of its structure, the prototypical tool has been used for change impact analyses. By hovering programs to be modified, the system's maintainer visually explored dependent sections and programs. They might be affected because they are called by a subject program or, vice versa, it is called by them. A similar task was to find all programs potentially affected by a copybook modification.

Our tool enabled the maintainer to efficiently estimate efforts required for an intended modification by analyzing its impact visually. He identified program sections that also required changes and that are subject to subsequent quality assurance. Thereby, the risk of introducing new bugs was reduced.

6.5 Discussion

Generality While the CBV extensions are demonstrated on large software-engineering data sets they are by no means restricted to such data sets and can be used to depict any multivariate compound graph compatible to the definition in Section 2.1.5.

Visual Scalability The visual-analysis technique is shown to scale to compound graphs extracted from industry-scale software systems. By applying the drill-down approach during analysis, visual clutter can be significantly reduced compared to the original CBV's

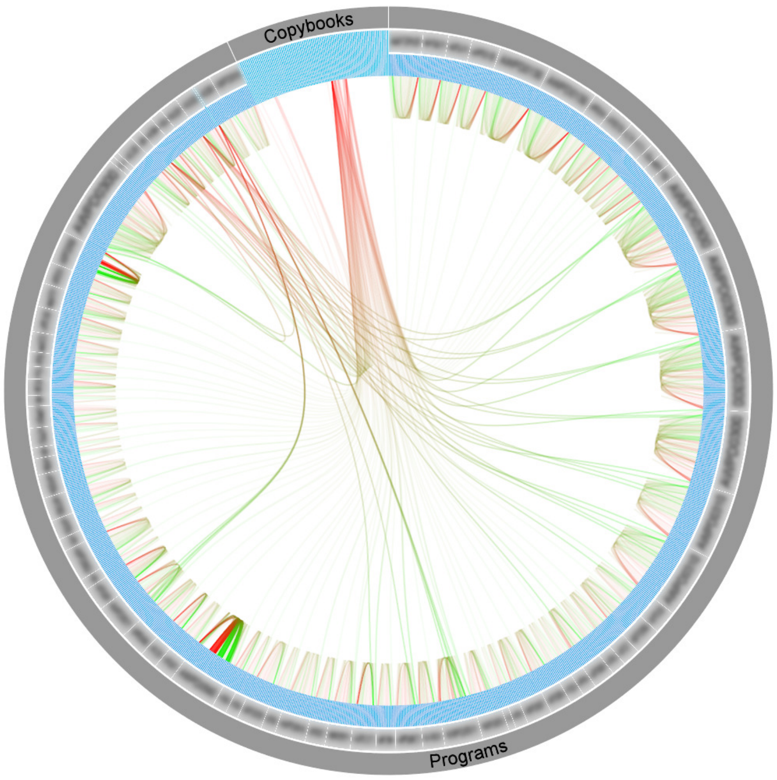


Figure 6.16: *Dependency view of an industrial COBOL system. Grey nodes represent programs and copybooks, blue nodes are sections within programs. Edges correspond to dependencies such as* PERFORM, CALL, *or* COPY *statements. Due to intellectual property reasons, we had to obfuscate most entity labels.*

full-detail views. Furthermore, zooming and filtering help reducing the amount of visualized data to relevant subsets of the raw data.

Limitations 3D-CBV While conducting the case studies, several minor limitations of the approach were identified; most of them can be at least alleviated. First, the human visual system has difficulties in discerning different color intensities, especially when the visual difference is small [276]. Hence, modules with attribute values that are numerically close to each other, but not equal, may nevertheless be seen as having equal values by humans. However, the tool permits to freely configure the color mapping so that it can be adjusted to discern numerically close values as well. Moreover, users cannot configure if attributes mapped to angular size shall be aggregated and how and aggregation of angular size automatically propagates up to higher levels in the depicted hierarchy. This limitation, however, is inherent to the visualization concept.

Since attribute values are first normalized before being mapped to visual variables, absolute values cannot be 'read' from the visual representation. This limitation can be mitigated, if not solved, by tooltips showing absolute values, which are displayed while hovering over modules. Yet, tooltips were rarely used during the case study.

Limitations D+D-CBV First, planning fine-grained reengineerings such as interface changes requires detailed system knowledge, which is hard to obtain using the only the visualization. Additionally, some interactions, e.g., introducing an indirection into a dependency, can lead to unpredictable code modifications within affected modules. Hence, the tool's resulting subtask list may be too coarse-grained in such cases so that architects need additional knowledge to determine reasonable effort estimations. In contrast, a reengineering's result list may become quite large and it can thus be hard for software architects to overview the entire list. In this case, its textual representation aggravates a sensible effort estimation. Introducing a detailed code model and linking the visualization to synchronized code views could provide additional support here, but would as well reduce generalizability of the concept.

Furthermore, the visualization concept does not support differentiating between distinct dependency types. Call relations, for instance, cannot be distinguished from type usages. In addition, defining a single generic modular view of a software system's structure limits the concept's applicability to analysis of one hierarchy at a time. A possible solution, though, is to depict one view per hierarchy (or dependency type) and synchronize these upon modifications.

The inherent limitation of the HEB technique – being restricted to showing only relationships between leaf-nodes – reduces the generalizability of the tool, but can be alleviated by applying the generalized shaded tubes developed for TraceDiff (cf. Section 5.2.3.4).

Benefits 3D-CBV In the given use case – perfective maintenance – depicting the whole system structure and all modules' measured attribute values in three dimensions, and dependencies simultaneously, provides several benefits.

Communicating analysis results to all participating stakeholders is facilitated. With outliers in all measured indicator dimensions becoming visible and recognizable, engineers, software architects, and managers have means to communicate about internal software quality that can be discussed and interpreted by all parties. In addition, analysis results are embedded into the architectural context of analyzed systems.

The visual analysis is lightweight, fast and uses quasi-standard indicators. Management can use the 3D-CBV as an instrument for determining the success of actions for improving internal software quality on a regular basis. The approach enables a human's visual system to fully exploit one of its strengths, namely identifying outliers using pre-attentive perception [116, 276]. Outliers in metric values are easily distinguishable by notably different height or color. Through the CBV's edge bundling, outliers in dependencies visually differ from other dependencies by not being bundled.

Ad-hoc interconnection of multiple indicators that does not require a defined mathematical operation for interconnection becomes possible by mapping the indicator's values to visual variables. By filtering indicators by time range, their progression over time can be analyzed. By this, trends can be spotted and preventive actions can be taken before outliers become a serious problem.

Benefits D+D-CBV Applying the interaction techniques for structure manipulation in a software-reengineering scenario provides benefits as follows. It enables software architects to visually analyze and design changes to software systems. Using a unified view of a software system's structure and inner dependencies, software architects can reorganize a software's modular structure interactively. Drag&drop, a dependency grabbing metaphor, and virtual creation and removal of software entities enable architects to experiment with a software's design and test their hypotheses.

Automatic recording of performed modifications reduces the users' cognitive load and lets them concentrate on the task at hand. This list later helps creating an effort estimation.

6.6 Further Applications in Dynamic Analysis and Other Domains

Aside from applications in static analysis, there are manifold applications in various domains for the presented extensions: Any structural data can be analyzed that is compatible with the definition given in Section 2.1.5 or that can be transformed accordingly. This generally includes organizational structures in companies; structures of legislative text which are typically strictly hierarchical; hierarchical computer networks; and dynamic analysis data.

For instance, one can represent function-based profiling information (call graphs) in the context of the respective system structure. This can help reasoning about performance problems on system-component level as well as analyzing the distribution of calls at runtime. In particular, the latter can differ from the compile-time view due to late binding.

An exemplary mapping configuration is as follows:

- System structure $\mapsto$ ring segments
- Call relationships $\mapsto$ edges
- Call costs per caller-callee pair $\mapsto$ edge thickness
- Call frequency per function $\mapsto$ height of ring segment

Chapter 7

Conclusions and Future Research Directions

"The older is not always a reliable model for the newer, the smaller for the larger, or the simpler for the more complex... Making something greater than any existing thing necessarily involves going beyond experience." — Henry Petroski, 2005 [206]

7.1 Summary

Modern society depends more and more on digital systems, which can automate, support, and fulfill a vast variety of tasks in our everyday private and economic life. Incorporated software increasingly influences many of these systems, in particular with regard to their feature set. As the usage of these typically complex systems is continuously growing, so is the importance of their maintenance. Unfortunately, software aging is a key problem that aggravates the maintenance of complex software systems. Negative side effects of some fundamental principles for good software design only add to the problem: Code reuse, for example, not only reduces code duplication but also increases the complexity of program comprehension tasks: Often it is in part responsible for the fact that implementation artifacts constituting one functionality are non-localized and scattered across a code base.

One central goal of software analysis and visualization is to provide the means to cope with these challenges. For example, software analysis and visualization of behavior and structure provide effective ways for engineers to navigate control flows as their primary information source in program comprehension. Likewise, it can provide architects and managers with higher-level information on the current state of a code base and development activities. There are several open challenges in software analysis and visualization, though, that are not well supported by state-of-the-art tools. In particular, this concerns the simultaneous exploratory analysis of multiple traces (e.g., resulting from multi-threaded behavior), correlation of traces with other software artifacts, and analysis and visual manipulation of large system structures.

Motivated by these challenges, this thesis has put forth techniques for the analysis and visualization of dynamic as well as static information on complex software systems. The focus here is on hierarchical event sequences, such as software traces representing control flow, and compound graphs representing system structure. The techniques' practicability was demonstrated with the help of several qualitative studies using subject data from both industry-scale software systems. This includes applications of the techniques in several collaborative projects with, among others, Francotyp-Postalia GmbH[1], Software Diagnostics Technologies GmbH[2], and SAP AG's Innovation Center Potsdam[3].

While we cannot generalize the results obtained through these studies, they still provide initial evidence that the techniques' application yields useful insights into the subject data in several scenarios. For example, correlating traces to other software artifacts can be key to either understanding phenomena originating not only in behavior, or to preselecting relevant ranges in the trace data based on the correlations, i.e., to reduce the initial search space. This thesis, further, showed that the concepts and techniques presented are generic and, as such, can be applied beyond software analysis for visualizing similarly structured data such as company structures.

A simple and portable implementation is a key requirement for the acceptance of software-visualization tools [141, 150]. The tool implementations presented here, which are compatible with Qt 4 and OpenGL version 1.2, require only basic texture-mapping and alpha blending. They render datasets of tens of thousands of elements at interactive frame rates, including interactive brushing, on moderate hardware such as NVIDIA GeForce GT 330M graphics card and Intel Core i7 M620 (manufactured in 2010).

All in all, this thesis has presented techniques for extending software-analysis tools, primarily with respect to the engineering perspective as well as the architectural and management perspectives. In so doing, it takes a more holistic view of the application scenarios of software analysis and visualization than most existing work in the field.

A Tool Set for Behavior Analysis

Mainly targeting the engineering perspective, a core contribution of this thesis is a set of visualization techniques for the dynamic analysis of multiple thread traces (Chapters 4 and 5). By visualizing function boundary traces captured from such systems, it helps to understand both single-threaded and multi-threaded activity on the function level. Its extensibility further allows for visualizing and analyzing specific aspects of multi-threading, such as synchronization; correlating such traces with system structures; and comparing traces with each other.

TraceVis (Section 4.1) is a fundamental technique for visualizing multiple-thread traces, which enables the synchronized exploration of the traces. A continuously configurable trace compaction thereby provides better visual scalability compared to the direct visualization of the raw data and emphasizes actual activity while de-emphasizing idle times in recorded executions. Several rendering variations and interaction techniques improve on the basic technique (Section 4.2). Shading and automated stack aggregation visually emphasize the structure of the visualized call stacks and de-clutter high-frequency regions in such call stacks, respectively. Interaction techniques allow for seamless transitions between context-dependent selection modes, and for context-preserving folding of call stacks to increase screen-space usage.

On this basis, the following techniques are subsequently built:

- SyncTrace (Section 4.3) is a technique for analyzing thread interplay captured in software traces from multi-threaded software systems. By a combination of customized interaction techniques and modified layout of the activity views, it allows for the simultaneous analysis of activity and threading-related operations such as synchronization. In this way, wait relationships and durations, I/O operations, and concurrency patterns can be analyzed.

[1]`http://www.francotyp.com`, last accessed 11/06/2013
[2]`http://www.softwarediagnostics.com`, last accessed 11/06/2013
[3]`http://global.sap.com/corporate-en/innovation/innovation-center/`, last accessed 11/06/2013

- ViewFusion (Section 5.1) is a method to integrate two distinct views – a structure view and an activity view – within a single screen-space area. It aims at facilitating the task of correlating two data sets. Moreover, it arguably uses screen-space more efficiently than the common linked-views technique by reducing the views' spatial distance. Using brushing and continuous selection techniques presented in Chapter 4, it provides an easy-to-use way of analyzing relationships between behavior and structure of software systems.

- TraceDiff (Section 5.2) extends TraceVis to show and explain (dis-)similarities of two thread traces by multiscale edge bundles. The brushing and continuous selection techniques presented in Chapter 4 are used here to enable the interactive exploration of the (dis)similarities. Through an extension of the HEB technique to shaded tubes and nesting of the tubes, the hierarchical relationship between the bundles becomes explicit and enables the use of two additional visual variables to encode multivariate relationships: Width and appearance of the tubes.

Altogether, SyncTrace, ViewFusion, and TraceDiff collectively demonstrate the extensibility and flexibility of the TraceVis approach and related interaction techniques.

Complementary Techniques for Structural Analysis

Chapter 6 discusses extensions to the circular bundle view (CBV) technique that primarily target the architectural and management perspectives: a) 3D-CBV and b) D+D-CBV. 3D-CBV (Section 6.2) takes the CBV to the 3-dimensional space to allow for the display of additional variables simultaneously. D+D-CBV (Section 6.3) includes interaction techniques for modifying structures in a visual manner. Users directly manipulate the visual representation of a structure without media disruption instead of modifying the raw data. Thereby, they can also benefit from the visual scalability when applying modifications and getting direct feedback on the change impact. For the architectural and management perspectives, these techniques can be applied to support a) the analysis of code quality based on software metrics and dependencies, and b) the restructuring of code bases using a generic structural model for hierarchically composed systems. Applications in dynamic analysis as well as beyond software analysis are possible and discussed in Section 6.6.

7.2 Conclusions

Complex dynamic processes tend to escape our perceptual and cognitive capabilities. The generally massive number of events and the inter-relationships in between are hard for us to memorize as a whole and difficult to correlate. If we are to understand these processes, we commonly must use tools to facilitate their understanding: For example, if such a process is captured as sequence data, we can slice this data into smaller segments and gain an understanding of the individual segments first before trying to understand the overall process.

The problem of understanding dynamic processes only gets harder if concurrency is involved in the process under examination, such as in multi-threaded software behavior: Established tools for analyzing serial processes, such as the aforementioned slicing approach, do not suffice for solving this kind of problem and thus are much less useful here. Fortunately, visualization is generally an effective means to promote the concurrency in dynamics to a *first-class entity* in the understanding process. The direct visualization, though, is not sufficient, as often the massive number of events still overwhelms our perception. We therefore generally require a means with which to produce to-the-point representations of dynamic processes that concentrate on and emphasize their important aspects.

The techniques presented in this thesis are also concerned with these challenges and provide several solutions for applying software analysis and visualization to analyze dynamic processes in complex software systems. Combined visualization of data from separate sources, such as dynamic and static analysis, provides the means to correlate the data. In effect, this provides new methods of software analysis, i.e., by *bridging the gap* between those data sources this thesis contributes to enriching the application scenarios of dynamic and static analysis.

Nevertheless, the hurdle in applying such prototype tools in visual software analysis is often too high in practice: In the case of tracing, they often require manual preparation to instrument the target system. Likewise, setting up the subsequent processing and visual analysis of the trace data typically requires much effort. This hinders their widespread acceptance. In effect, users would likely spend more time setting up the analysis and visualization systems than solving their actual problem. The result is that they restrict themselves to traditional approaches.

One of the purposes of integrated development environments (IDEs) is, therefore, to streamline the use of software-engineering tools. They essentially form tool chains that enable the easy use and combination of the tools contained. So far, only a few visualization techniques have actually been integrated into such tool chains, which would allow one to fully leverage the benefits of the two. One of the few examples is the Software Diagnostics Flight Recorder[4]. For software engineering to benefit from the techniques presented in this thesis, an IDE integration is thus an important next step. With the presented techniques being applicable to common procedural programming languages, they can be widely used in several common IDEs such as Microsoft Visual Studio.

7.3 Future Research Directions

This work can be extended in several directions: On the one hand, this includes extensions to and new applications for the presented techniques. On the other hand, one promising direction is exploring how the existing techniques can be combined with orthogonal extensions.

Multi-Core and Many-Core Scalability

Multi-threading is typically either applied to (a) keep interactive applications responsive despite the use of blocking operations such as I/O or (b) to speed up computationally intensive operations by splitting them into several independent tasks in a divide-and-conquer manner. In the latter case, multi-threaded software often scales poorly with the number of available cores: When run on multi-core and many-core systems, such software often hits an invisible 'barrier' which keeps it from using all available cores. For instance, it may correctly use all cores in a 4-core system, but keeps using only 4 cores in a 64-core system. One common reason for this barrier is the fact that synchronization is done in a way that significantly limits the maximum possible parallelism. This can be addressed by integrating runtime data related to the target application with scheduling data from the underlying operating system (e.g., workload of cores and assignment of threads to cores). With this, one could analyze how low degrees of concurrency on the application level (i.e., core usage) correlate with scheduling decisions, and vice versa.

[4]`http://www.softwarediagnostics.com/flight-recorder/`, last accessed 10/31/2013

Parallel and Distributed Systems

A generalization of the multi-threading principle is the usage of multiple networked machines (hardware nodes) to form a distributed system. Each node represents an autonomous computing unit that communicates with other nodes using message passing systems. Several techniques exist for the analysis of such distributed systems, but they often treat individual nodes as black boxes, i.e., they don't provide information on the nodes' internal behavior: They stricly analyze messages passed between the nodes. This inherently restricts use of the techniques' to high-level behavior analysis of the distributed systems. However, the cause of a problem that becomes evident in such high-level analysis often is of a low-level nature: For instance, the higher-level insight that a message is not being sent does not explain *why* it is not sent. For this, engineers could use information on the runtime behavior of each node, or on the software systems that define the behavior of a node. Combining existing work on behavioral analysis of distributed systems and behavioral analysis of individual software systems appears to be a promising candidate in coping with this challenge.

Analysis of Correlated Behavior-Evolution

As this thesis has shown, correlating software traces with each other can provide useful insights into software behavior. This can further be applied in analyzing the evolution of such behavior [129], which can yield useful insights in several use cases, including

- **Regression Testing:** Trace comparison as an automated technique could help in detecting deviations in software behavior that are not covered by test assertions. That is, comparing traces from test executions to a trace capturing the 'ground truth' of a behavior (a correct execution) can allow for increased precision of tests: To explicit assertions in unit tests, the trace comparison would add 'implicit' assertions related to control-flow. Moreover, investigating the differences between the two traces can help explain why a test fails.
- **Behavior Evolution:** Correlating software traces with evolution data can support reasoning about the effect of changes on software behavior [6]. This task is especially difficult, since there generally is only an ambiguous mapping between the effect – behavioral variations – and its possible causes: The effect can be induced either by variations in the execution context or by changes to the code base. By comparing software traces captured from different versions of a given functionality, one can analyze the two possible causes in a combined way.

Analysis of Memory Usage

The analysis of software behavior in this thesis focuses on control flow captured as FBTs. As shown in this thesis, by comparing traces from executions based on different execution contexts (including different input data), the effects of data-dependent execution on control flow can be explained at the function level. Nevertheless, the *cause* of these differences can be in the data. In many cases, the analysis of memory usage, such as data flow, can thus provide additional insight into behavioral aspects of software [7, 8]. It can enable us to explain aspects related to data-dependent execution [59] such as those illustrated in Listing 7.1.

Consequently, comparing control flow and combining it with data-flow information can enable us to better explain *why* executions differ, in order to investigate the *cause* of the differences: If we know the value of the variable x in Listing 7.1 above, we can explain *why foo()* was executed instead of *bar()*.

```
    if(x > 0)
        foo();
    else
        bar();
```

Listing 7.1: *Example code snippet for data-dependent execution.*

Moreover, we can optimize memory performance by analyzing memory reads/writes and caching behavior. For instance, we can significantly speed up memory access in systems using memory caches by reducing cache miss rates [57].

Fine-Grained Analysis of Control Flow

The analysis of software behavior using FBTs provides useful insights into control flow, but it also limits the analysis to function calls as atomic events: It abstracts from the fact that functions typically comprise a number of basic blocks that in turn can comprise multiple instructions. This is useful in several respects. For instance, it contributes to the reduction of trace data. That is, traces captured at the basic-block level are typically several orders of magnitude larger than FBTs, not to mention traces capturing individual instructions. Despite their size, they generally enable more detailed analysis of software behavior. Although this requires the means to cope with the resulting massive trace data, their visualization using similar activity views as presented in this thesis is possible: Since they 'only' constitute further hierarchical subdivisions of functions, the trace definition given in this thesis can be extended to capture these subdivisions. The icicle plots currently used in the activity views to depict function calls can be further subdivided to additionally depict basic blocks and instructions. Likewise, aggregating repetitive behavior at these levels can be used to reduce clutter in the icicle plots and help find these patterns.

Formal Evaluation

The techniques put forth in this thesis so far have been evaluated by applying them in the analysis of complex software systems. Here, they proved to be useful in answering real-life questions concerning these systems. A formal evaluation by user studies and controlled experiments, though, is certainly important to provide further insight and can be generalized to a broader level: What are the limitations in terms of visual scalability and usability, and, for instance, by how much do the techniques reduce the effort required to solve specific tasks?

Computational Scalability and Practicability

While the thesis' focus is on *developing* techniques, the prototypes' implementations are not yet fully optimized as to computational scalability. This along with improved ease of use could facilitate their application. Thus, they are important future steps to take for these techniques to be successful in practice.

For instance, computational scalability of several of the techniques presented can be improved by fully leveraging state-of-the-art rendering techniques (e.g., shader programs) available in today's graphics hardware. Nevertheless, for a wide-range application in practice, a tradeoff still has to be made between speed and generality of the implementation, since the typical office desktop-hardware is not (yet) equipped with powerful enough graphics hardware. Fortunately, this is about to change, so that even low-budget graphics hardware is becoming more and more powerful and is already able to execute general-purpose shader programs.

Service-Based Visualization

Provisioning stand-alone applications for software analysis and visualization can be cumbersome in several respects: First, every installation needs access to the relevant system information, which typically requires the analysis information to be synchronized with every installation or even requires that the software analysis portion is implemented for every installation. Second, the visualization itself often poses specific hardware requirements with respect to graphics hardware and power consumption. In particular, mobile devices such as smart phones or tablets often do not provide sufficiently powerful graphics hardware and, even if they do, quickly run out of battery when fully leveraging this power for longer periods. Third, network connectivity can be an issue if the system information is only available remotely. For these and other reasons, provisioning visualization as a service is a way to overcome these challenges: Software analysis and visualization is done on powerful server hardware, and only the resulting visual representation is delivered to the clients. A proof-of-concept protoype for providing 3-dimensional treemaps as a service has already been implemented [298].

References

[1] *IEEE Standard Glossary of Software Engineering Terminology 610.12-1990*, 1990.

[2] Software-Projekt Fiscus: Steuerzahler blechen für IT-Panne. *FOCUS Money Online*, 2004. `http://www.focus.de/finanzen/news/finanzamt-software_aid_84105.html`.

[3] Abuthawabeh, A. and Zeckzer, D. IMMV: An Interactive Multi-Matrix Visualization for Program Comprehension. In *Proceedings of the Working Conference on Software Visualization*. IEEE, 2013.

[4] Adamoli, A. and Hauswirth, M. Trevis: A Context Tree Visualization & Analysis Framework and Its Use for Classifying Performance Failure Reports. In *Proceedings of the International Symposium on Software Visualization*, pages 73–82. ACM, 2010.

[5] Alam, S. and Dugerdil, P. EvoSpaces: 3D Visualization of Software Architecture. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*, pages 500–505. IEEE, 2007.

[6] Alcocer, J. P. S., Bergel, A., Ducasse, S., and Denker, M. Performance Evolution Blueprint: Understanding the Impact of Software Evolution on Performance. In *Proceedings of the Working Conference on Software Visualization*. IEEE, 2013.

[7] Allen, F. E. and Cocke, J. A Program Data Flow Analysis Procedure. *Communications of the ACM*, 19(3):137–147, 1976.

[8] Allen, F. A Basis for Program Optimization. In *Proceedings of the International Federation for Information Processing Congress*, pages 385–390. North-Holland Pub. Co., 1971.

[9] Altman, E., Arnold, M., Fink, S., and Mitchell, N. Performance Analysis of Idle Programs. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, pages 739–753. ACM, 2010.

[10] Amenta, N. and Klingner, J. Case Study: Visualizing Sets of Evolutionary Trees. In *Proceedings of the Symposium on Information Visualization*, pages 71–74. IEEE, 2002.

[11] American Heritage Dictionaries Editors, editor. *The American Heritage Dictionary of the English Language*. Houghton Mifflin Harcourt, 2004.

[12] Antoniol, G. and Gueheneuc, Y.-G. Feature Identification: An Epidemiological Metaphor. *IEEE Transactions on Software Engineering*, 32(9):627–641, 2006.

[13] Artho, C., Havelund, K., and Honiden, S. Visualization of Concurrent Program Executions. In *Proceedings of the International Conference on Computer Software and Applications*, pages 541–546, 2007. IEEE.

[14] AT & T. The Graphviz package, 2010. `http://www.graphviz.org`.

[15] Auber, D. Tulip Visualization System, 2012. `http://tulip.labri.fr`.

[16] Baker, M. J. and Eick, S. G. Space-Filling Software Visualization. *Journal of Visual Languages and Computing*, 6:119–133, 1995.

[17] Balzer, M., Deussen, O., and Lewerentz, C. Voronoi Treemaps for the Visualization of Software Metrics. In *Proceedings of the Symposium on Software Visualization*, pages 165–172. ACM, 2005.

[18] Basili, V. R. Evolving and Packaging Reading Technologies. *Journal of Systems and Software*, 38(1):3–12, 1997.

[19] Battista, G. D., Eades, P., Tamassia, R., and Tollis, I. G. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[20] Beck, F., Petkov, R., and Diehl, S. Visually Exploring Multi-Dimensional Code Couplings. In *Proceedings of the International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–8. IEEE, 2011.

[21] Beck, K. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999.

[22] Bederson, B. B., Shneiderman, B., and Wattenberg, M. Ordered and Quantum Treemaps: Making Effective Use of 2D Space to Display Hierarchies. *ACM Transactions on Graphics*, 21(4):833–854, 2002.

[23] Bedy, M., Carr, S., Huang, X., and Shene, C.-K. A Visualization System for Multithreaded Programming. *SIGCSE Bulletin*, 32(1):1–5, 2000.

[24] Beetz, M. and Grosskreutz, H. Probabilistic Hybrid Action Models for Predicting Concurrent Percept-Driven Robot Behavior. *Journal of Artificial Intelligence Research*, 24(1):799–849, 2005.

[25] Bennett, C., Myers, D., Storey, M.-A., and German, D. Working with 'Monster' Traces: Building a Scalable, Usable Sequence Viewer. In *Proceedings of the International Workshop on Program Comprehension through Dynamic Analysis*, pages 1–5, 2007.

[26] Berger, E. D. Software Needs Seatbelts and Airbags. *Communications of the ACM*, 55(9):48–53, 2012.

[27] Berthold, J. and Loogen, R. Visualizing Parallel Functional Program Runs: Case Studies with the Eden Trace Viewer. In *Proceedings of the International Conference Parallel Computing*, pages 121–128, 2007.

[28] Biggerstaff, T. J., Mitbander, B. G., and Webster, D. The Concept Assignment Problem in Program Understanding. In *Proceedings of the International Conference on Software Engineering*, pages 482–498. IEEE, 1993.

[29] Bladh, T., Carr, D. A., and Scholl, J. Extending Tree-Maps to Three Dimensions: A Comparative Study. In *Proceedings of the Asia-Pacific Conference on Computer-Human Interaction*, pages 50–59. Springer Verlag, 2004.

[30] Blanchette, J. and Summerfield, M. *C++ GUI Programming with Qt4*. Prentice Hall International, 2nd edition, 2008.

[31] Boehm, B. A Spiral Model of Software Development and Enhancement. *SIGSOFT Software Engineering Notes*, 11(4):14–24, 1986.

[32] Bohnet, J. *Visualization of Execution Traces and its Application to Software Maintenance*. PhD thesis, Hasso-Plattner-Institute at the University of Potsdam, Germany, 2010.

[33] Bohnet, J. and Döllner, J. Monitoring Code Quality and Development Activity by Software Maps. In *Proceedings of the International Workshop on Managing Technical Debt*, pages 9–16. IEEE, 2011.

[34] Bohnet, J. and Döllner, J. Visually Exploring Control Flow Graphs to Support Legacy Software Migration. In *Proceedings der Software Engineering Konferenz der Gesellschaft für Informatik*, pages 245–246. GI, 2007.

[35] Bourquin, F. and Keller, R. K. High-Impact Refactoring Based on Architecture Violations. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 149–158. IEEE, 2007.

[36] Bremm, S., von Landesberger, T., Hess, M., Schreck, T., Weil, P., and Hamacherk, K. Interactive Visual Comparison of Multiple Trees. In *Proceedings of the Conference on Visual Analytics Science and Technology*, pages 31–40. IEEE, 2011.

[37] Broberg, M., Lundberg, L., and Grahn, H. Visualization and Performance Prediction of Multithreaded Solaris Programs by Tracing Kernel Threads. *International Parallel Processing Symposium*, 0:407–413, 1999.

[38] Brodbeck, D. and Girardin, L. Visualization of Large-Scale Customer Satisfaction Surveys Using a Parallel Coordinate Tree. In *Symposium on Information Visualization*, pages 197–201. IEEE, 2003.

[39] Brooks, F. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, 1987.

[40] Brown, M. H. and Sedgewick, R. A System for Algorithm Animation. *SIGGRAPH Computer Graphics*, 18(3):177–186, 1984.

[41] Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., Sangwan, R., Seaman, C., Sullivan, K., and Zazworka, N. Managing Technical Debt in Software-Reliant Systems. In *Proceedings of the Workshop on the Future of Software Engineering Research*, pages 47–52. ACM, 2010.

[42] Brunner, R. TSC and Power Management Events on AMD Processors. Technical report, AMD Corporation, 2005.

[43] Burch, M. *Visualizing Static and Dynamic Relations in Information Hierarchies*. PhD thesis, Universität Trier, 2009.

[44] Burch, M., Blascheck, T., Louka, C., and Weiskopf, D. Visualizing Hierarchy Changes by Dynamic Indented Plots. In *Proceedings of the International Conference on Information Visualization Theory and Applications*, pages 91–98. SCITEPRESS, 2014.

[45] Butler, D. M., Almond, J. C., Bergeron, R. D., Brodlie, K. W., and Haber, R. B. Visualization Reference Models. In *Proceedings of the Conference on Visualization*, pages 337–342. IEEE, 1993.

[46] Cairo, A. *The Functional Art: An Introduction to Information Graphics and Visualization.* New Riders, 2013.

[47] Card, S. K., Mackinlay, J. D., and Shneiderman, B. *Readings in Information Visualization: Using Vision to Think.* Morgan Kaufmann Publishers Inc., 1999.

[48] Carr, D. A. Guidelines for Designing Information Visualization Applications. In *Proceedings of the Ericsson Conference on Usability Engineering*, pages 1–7, 1999.

[49] Caserta, P. and Zendra, O. Visualization of the Static Aspects of Software: A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 17(7):913–933, 2011.

[50] Caserta, P., Zendra, O., and Bodénès, D. 3D Hierarchical Edge Bundles to Visualize Relations in a Software City Metaphor. In *International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–8. IEEE, 2011.

[51] Cataldo, M. and Herbsleb, J. Coordination Breakdowns and Their Impact on Development Productivity and Software Failures. *IEEE Transactions on Software Engineering*, 39(3):343–360, 2013.

[52] Chapman, B., Jost, G., and Pas, R. v. d. *Using OpenMP: Portable Shared Memory Parallel Programming.* Scientific and Engineering Computation. MIT Press, 2007.

[53] Charette, R. N. Why Software Fails. *IEEE Spectrum*, 42(9):42–49, 2005.

[54] Chen, C. *Information Visualization: Beyond the Horizon.* Springer Verlag, 2nd edition, 2006.

[55] Chi, E. H. A Taxonomy of Visualization Techniques Using the Data State Reference Model. In *Proceedings of the Symposium on Information Vizualization*, pages 69–76. IEEE, 2000.

[56] Chikofsky, E. J. and Cross II, J. H. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[57] Choudhury, A. N. M. I. and Rosen, P. Abstract Visualization of Runtime Memory Behavior. In *Proceedings of the International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–8. IEEE, 2011.

[58] Choudhury, A. I. *Visualizing Program Memory Behavior Using Memory Reference Traces.* PhD thesis, School of Computing, University of Utah, 2012.

[59] Cleve, H. and Zeller, A. Locating Causes of Program Failures. In *Proceedings of the International Conference on Software Engineering*, pages 342–351. ACM, 2005.

[60] Cockburn, A., Karlson, A., and Bederson, B. B. A Review of Overview+Detail, Zooming, and Focus+Context Interfaces. *ACM Computing Surveys*, 41(1):2:1–2:31, 2009.

[61] Cohn, M. *Succeeding with Agile: Software Development Using Scrum.* Addison-Wesley, 2009.

[62] Coleman, D., Ash, D., Lowther, B., and Oman, P. Using Metrics to Evaluate Software System Maintainability. *IEEE Computer*, 27(8):44–49, 1994.

[63] Conroy, P. Technical Debt: Where Are the Shareholders' Interests? *IEEE Software*, 29(6):88–88, 2012.

[64] Corbi, T. A. Program Understanding: Challenge for the 1990's. *IBM Systems Journal*, 28(2):294–306, 1989.

[65] Cornelissen, B. and Moonen, L. Visualizing Similarities in Execution Traces. In *Proceedings of the Workshop on Program Comprehension through Dynamic Analysis*, pages 6–10. IEEE, 2007.

[66] Cornelissen, B. and Moonen, L. On Large Execution Traces and Trace Abstraction Techniques. *Journal of Software Maintenance and Evolution: Research and Practice*, 00:1–7, 2008.

[67] Cornelissen, B., Holten, D., Zaidman, A., Moonen, L., van Wijk, J. J., and van Deursen, A. Understanding Execution Traces Using Massive Sequence and Circular Bundle Views. In *Proceedings of the International Conference on Program Comprehension*, pages 49–58. IEEE, 2007.

[68] Cornelissen, B., Zaidman, A., Holten, D., Moonen, L., van Deursen, A., and van Wijk, J. J. Execution Trace Analysis through Massive Sequence and Circular Bundle Views. *Journal of Systems and Software*, 81(12):2252–2268, 2008.

[69] Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., and Koschke, R. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[70] Cunningham, W. The WyCash Portfolio Management System. OOPSLA' 92 Experience Report, 1992.

[71] Dang, T. N., Anand, A., and Wilkinson, L. TimeSeer: Scagnostics for High-Dimensional Time Series. *IEEE Transactions on Visualization and Computer Graphics*, 19(3):470–483, 2013.

[72] de Kergommeaux, J. C., Stein, B. D. O., and Bernard, P. Pajé, an Interactive Visualization Tool for Tuning Multi-Threaded Parallel Applications. *Parallel Computing*, 26(10):1253–1274, 2000.

[73] De Pauw, W. and Heisig, S. Visual and Algorithmic Tooling for System Trace Analysis: A Case Study. *SIGOPS Operating Systems Review*, 44(1):97–102, 2010.

[74] De Pauw, W. and Heisig, S. Zinsight: A Visual and Analytic Environment for Exploring Large Event Traces. In *Proceedings of the International Symposium on Software Visualization*, pages 143–152. ACM, 2010.

[75] De Pauw, W., Lorenz, D., Vlissides, J., and Wegman, M. Execution Patterns in Object-Oriented Visualization. In *Proceedings of the Conference on Object-Oriented Technologies and Systems*, pages 219–234. USENIX, 1998.

[76] De Pauw, W., Jensen, E., Mitchell, N., Sevitsky, G., Vlissides, J. M., and Yang, J. Visualizing the Execution of Java Programs. In *Proceeding of the International Seminar on Software Visualization*, pages 151–162. Springer-Verlag, 2002.

[77] Demeyer, S., Ducasse, S., and Nierstrasz, O. *Object Oriented Reengineering Patterns*. Morgan Kaufmann Publishers Inc., 2002.

[78] Diehl, S. *Software Visualization. Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, Berlin, 2007.

[79] d'Ocagne, M. *Coordonnées Parallèles et Axiales: Méthode de Transformation Géométrique et Procédé Nouveau de Calcul Graphique Déduits de la Considération des Coordonnées Parallèles*. Gauthier-Villars, 1885.

[80] Du Bois, K., Sartor, J. B., Eyerman, S., and Eeckhout, L. Bottle Graphs: Visualizing Scalability Bottlenecks in Multi-Threaded Applications. In *Proceedings of the SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, pages 355–372. ACM, 2013.

[81] Dugerdil, P. and Alam, S. Execution Trace Visualization in a 3D Space. In *Proceedings of the International Conference on Information Technology: New Generations*, pages 38–43. IEEE, 2008.

[82] Ebert, J., Riediger, V., and Winter, A. Graph Technology in Reverse Engineering: The TGraph Approach. In *Proceedings of the Workshop Software Reengineering*, volume 126 of *LNI*, pages 67–81. GI, 2008.

[83] Eick, S. G., Graves, T. L., Karr, A. F., Mockus, A., and Schuster, P. Visualizing Software Changes. *IEEE Transactions on Software Engineering*, 28:396–412, 2002.

[84] Eick, S. G., Steffen, J. L., and Sumner, Jr., E. E. Seesoft – A Tool for Visualizing Line Oriented Software Statistics. *IEEE Transactions on Software Engineering*, 18: 419–430, 1999.

[85] Eiglsperger, M., Gutwenger, C., Kaufmann, M., Kupke, J., Jünger, M., Leipert, S., Klein, K., Mutzel, P., and Siebenhaller, M. Automatic Layout of UML Class Diagrams in Orthogonal Style. *Information Visualization*, 3(3):189–208, 2004.

[86] Eisenbarth, T., Koschke, R., and Simon, D. Locating Features in Source Code. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003.

[87] Erlikh, L. Leveraging Legacy System Dollars for E-Business. *IT Professional*, 2(3): 17–23, 2000.

[88] Ernst, M. D. Invited Talk: Static and Dynamic Analysis: Synergy and Duality. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*, pages 35–35. ACM, 2004.

[89] Femmer, H., Broy, N., Zec, M., MacWilliams, A., and Eckl, R. Dynamic Software Visualization with BusyBorg - A Proof of Concept. In *Proceedings of the Conference on Computer Software and Applications Conference*, pages 492–497. IEEE, 2011.

[90] Ferrante, J., Ottenstein, K. J., and Warren, J. D. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[91] Fischmeister, S. and Lam, P. Time-Aware Instrumentation of Embedded Software. *IEEE Transactions on Industrial Informatics*, 6(4):652–663, 2010.

[92] Fittkau, F., Waller, J., Wulf, C., and Hasselbring, W. Live Trace Visualization for Comprehending Large Software Landscapes: The ExplorViz Approach. In *Proceedings of the Working Conference on Software Visualization*. IEEE, 2013.

[93] Fleming, S., Kraemer, E., Stirewalt, R., and Dillon, L. Debugging Concurrent Software: A Study Using Multithreaded Sequence Diagrams. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*, pages 33–40. IEEE, 2010.

[94] Forward, A. and Lethbridge, T. C. The Relevance of Software Documentation, Tools and Technologies: A Survey. In *Proceedings of the Symposium on Document Engineering*, pages 26–33. ACM, 2002.

[95] Foster, A. A Nonlinear Model of Information-Seeking Behavior. *Journal of the American Society for Information Science and Technology*, 55(3):228–237, 2004.

[96] Fowler, M. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley Professional, 2000.

[97] Franklin, M. *The Computer Engineering Handbook: Digital Systems and Applications (Second Edition)*, chapter Multithreading, Multiprocessing, pages 35–51. CRC Press, 2008.

[98] Furnas, G. W. Generalized Fisheye Views. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 16–23. ACM, 1986.

[99] Furnas, G. W. and Zacks, J. Multitrees: Enriching and Reusing Hierarchical Structure. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 330–336. ACM, 1994.

[100] Gait, J. A Probe Effect in Concurrent Programs. *Software – Practice & Experience*, 16:225–233, 1986.

[101] George, B. and Nagpal, P. Optimizing Parallel Applications Using Concurrency Visualizer: A Case Study. Technical report, Microsoft Corporation (Parallel Computing Platform Group), 2010.

[102] Ghoniem, M. and Fekete, J.-D. Animating Treemaps. In *Proceedings of the HCIL Symposium-Workshop on Treemap Implementations and Applications*, 2001.

[103] Giese, H. and Wirtz, G. Visual Modeling of Object-Oriented Distributed Systems. *Journal of Visual Languages and Computing*, 12:183–202, 2001.

[104] Gokpinar, B., Hopp, W., and Iravani, S. The Impact of Misalignment of Organizational Structure and Product Architecture on Quality in Complex Product Development. *Management Science*, 56:468–484, 2010.

[105] Graham, M. and Kennedy, J. Combining Linking and Focusing Techniques for a Multiple Hierarchy Visualisation. In *Proceedings of the International Conference on Information Visualisation*, pages 425–432, 2001.

[106] Graham, M. and Kennedy, J. A Survey of Multiple Tree Visualisation. *Information Visualization*, 9:235–252, 2009.

[107] Graham, M. and Kennedy, J. Exploring Multiple Trees Through DAG Representations. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1294–1301, 2007.

[108] Graham, S. L., Kessler, P. B., and McKusick, M. K. Gprof: A Call Graph Execution Profiler. *SIGPLAN Notes*, 39(4):49–57, 2004.

[109] Greevy, O. and Ducasse, S. Correlating Features and Code Using a Compact Two-Sided Trace Analysis Approach. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 314–323. IEEE, 2005.

[110] Guerra-Gómez, J. A., Pack, M. L., Plaisant, C., and Shneiderman, B. Visualizing Change Over Time Using Dynamic Hierarchies: TreeVersity2 and the StemView. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2566–2575, 2013.

[111] Hamou-Lhadj, A. and Lethbridge, T. Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System. In *Proceedings of the International Conference on Program Comprehension*, pages 181–190. IEEE, 2006.

[112] Hamou-Lhadj, A. *Techniques to Simplify the Analysis of Execution Traces for Program Comprehension.* PhD thesis, University of Ottawa, 2005.

[113] Hao, M. C., Glajchen, D., and Sventek, J. S. SmallSync: A Methodology for Diagnosis Visualization of Distributed Processes on the Web. Technical report, Hewlett Packard, 1998.

[114] Haroz, S. and Whitney, D. How Capacity Limits of Attention Influence Information Visualization Effectiveness. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2402–2410, 2012.

[115] Havre, S., Hetzler, E., Whitney, P., and Nowell, L. ThemeRiver: Visualizing Thematic Changes in Large Document Collections. *IEEE Transactions on Visualization and Computer Graphics*, 8:9–20, 2002.

[116] Healey, C. G., Booth, K. S., and Enns, J. T. High-Speed Visual Estimation Using Preattentive Processing. *ACM Transactions on Human-Computer Interaction*, 3(2): 107–135, 1996.

[117] Heath, M. T. and Etheridge, J. A. Visualizing the Performance of Parallel Programs. *IEEE Software*, 8:29–39, 1991.

[118] Heisenberg, W. Über den Anschaulichen Inhalt der Quantentheoretischen Kinematik und Mechanik. *Zeitschrift für Physik*, 43(3):172–198, 1927. `http://osulibrary.oregonstate.edu/specialcollections/coll/pauling/bond/papers/corr155.1.html`.

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

[120] Hoc, J.-M. and Nguyen-Xuan, A. Language Semantics, Mental Models, and Analogy. *Psychology of Programming*, pages 139–156, 1990.

[121] Holenderski, M., Bril, R. J., and Lukkien, J. J. Grasp: Visualizing the Behavior of Hierarchical Multiprocessor Real-Time Systems. *Journal of Systems Architecture*, 59 (6):307–314, 2013.

[122] Holland, J. H. *Adaptation in Natural and Artificial Systems.* MIT Press, 1992.

[123] Holten, D. Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12: 741–748, 2006.

[124] Holten, D. and Van Wijk, J. J. Visual Comparison of Hierarchically Organized Data. *Computer Graphics Forum*, 27(3):759–766, 2008.

[125] Horwood, P., Wygodny, S., and Zardecki, M. Debugging Multithreaded Applications. *Dr. Dobb's Journal of Software Tools*, 25(3):32, 34–37, 2000.

[126] Hummel, O., Janjic, W., and Atkinson, C. Proposing Software Design Recommendations Based on Component Interface Intersecting. In *Proceedings of the International Workshop on Recommendation Systems for Software Engineering*, pages 64–68. ACM, 2010.

[127] Hunold, S., Hoffmann, R., and Suter, F. Jedule: A Tool for Visualizing Schedules of Parallel Applications. In *International Conference on Parallel Processing Workshops*, pages 169–178, 2010.

[128] Hutchens, D. H. and Basili, V. R. System Structure Analysis: Clustering with Data Bindings. *IEEE Transaction on Software Engineering*, 11:749–757, 1985.

[129] Idris, M., Mehrabian, A., Hamou-Lhadj, A., and Khoury, R. Pattern-Based Trace Correlation Technique to Compare Software Versions. In Kamel, M., Karray, F., and Hagras, H., editors, *Autonomous and Intelligent Systems*, Lecture Notes in Computer Science, pages 159–166. Springer Verlag, 2012.

[130] Javed, W. and Elmqvist, N. Stack Zooming for Multi-Focus Interaction in Time-Series Data Visualization. In *Proceedings of the Pacific Visualization Symposium*, pages 33–40. IEEE, 2010.

[131] Jerding, D. F. and Stasko, J. T. The Information Mural: A Technique for Displaying and Navigating Large Information Spaces. *IEEE Transactions on Visualization and Computer Graphics*, 4:257–271, 1998.

[132] Johnson-Laird, P. N., Legrenzi, P., Girotto, V., Legrenzi, M. S., and Caverni, J. P. Naive Probability: A Mental Model Theory of Extensional Reasoning. *Psychological Review*, 106(1):62–88, 1999.

[133] Jones, J. A., Harrold, M. J., and Stasko, J. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the Interational Conference on Software Engineering*, pages 467–477. IEEE, 2002.

[134] Jones, J. A., Orso, A., and Harrold, M. J. GAMMATELLA: Visualizing Program-Execution Data for Deployed Software. *Information Visualization*, 3(3):173–188, 2004.

[135] Jürgens, T. and Klein, C. Innere Qualität Zahlt Sich Aus: Wie die Deutsche Post die Zukunftsfähigkeit ihrer IT-Systeme Sichert. *OBJEKTspektrum*, 01(01):55–62, 2010.

[136] Kadaba, N. R., Irani, P. P., and Leboe, J. Visualizing Causal Semantics Using Animations. In *IEEE Transactions on Computer Graphics and Applications*, 2007.

[137] Kamiya, T. Agec: An Execution-Semantic Clone Detection Tool. In *Proceedings of the International Conference on Program Comprehension*, pages 227–229. IEEE, 2013.

[138] Karlsson, B. *Beyond the C++ Standard Library.* Addison-Wesley Professional, 2005.

[139] Keller, T. and Tergan, S.-O. *Knowledge and Information Visualization*, chapter Visualizing Knowledge and Information: An Introduction, pages 1–27. LNCS 3426. Springer Verlag, 2005.

[140] Kerren, A. and Stasko, J. T. Algorithm Animation - Introduction. In *Software Visualization*, pages 1–15, 2002.

[141] Kienle, H. *Building Reverse Engineering Tools with Software Components.* PhD thesis, University of Victoria, 2006.

[142] Kim, B.-C., Jun, S.-W., Hwang, D. J., and Jun, Y.-K. Visualizing Potential Deadlocks in Multithreaded Programs. In *Proceedings of the International Conference on Parallel Computing Technologies*, pages 321–330. Springer Verlag, 2009.

[143] Kincaid, R. SignalLens: Focus+Context Applied to Electronic Time Series. *IEEE Transactions on Visualization and Computer Graphics*, 16:900–907, 2010.

[144] Kirk, D. B. and Hwu, W.-M. *Programming Massively Parallel Processors: A Hands-On Approach.* Morgan Kaufmann, 2010.

[145] Kitchin, R. M. Cognitive Maps: What Are They And Why Study Them? *Journal of Environmental Psychology*, 14(1):1–19, 1994.

[146] Ko, A. J., Aung, H., and Myers, B. A. Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks. In *Proceedings of the International Conference on Software Engineering*, pages 126–135. ACM, 2005.

[147] Ko, A. J., DeLine, R., and Venolia, G. Information Needs in Collocated Software Development Teams. In *Proceedings of the International Conference on Software Engineering*, pages 344–353. IEEE, 2007.

[148] Kobayashi, K., Kamimura, M., Yano, K., Kato, K., and Matsuo, A. SArF Map: Visualizing Software Architecture from Feature and Layer Viewpoints. In *Proceedings of the International Conference on Program Comprehension*, pages 43–52. IEEE, 2013.

[149] Kosara, R., Healey, C. G., Interrante, V., Laidlaw, D. H., and Ware, C. User Studies: Why, How, and When? *IEEE Computer Graphics and Applications*, 23(4):20–25, 2003.

[150] Koschke, R. Software Visualization in Software Maintenance, Reverse Engineering, and Re-Engineering: A Research Survey. *Journal of Software Maintenance and Evolution*, 15(2):87–109, 2003.

[151] Koschke, R. Software Visualization for Reverse Engineering. In *Software Visualization*, pages 138–150, 2001.

[152] Kranzlmüller, D. *Software Visualization: From Theory to Practice*, chapter Visualizing Program Behavior with the Event Graph, pages 29–57. Kluwer Academic Publishers, 2003.

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

[154] Kruskal, J. B. and Landwehr, J. M. Icicle Plots: Better Displays for Hierarchical Clustering. *The American Statistician*, 37(2):162–168, 1983.

[155] Lamping, J., Rao, R., and Pirolli, P. A Focus+Context Technique Based on Hyperbolic Geometry for Visualizing Large Hierarchies. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 401–408. ACM/Addison-Wesley, 1995.

[156] Langelier, G., Sahraoui, H., and Poulin, P. Visualization-Based Analysis of Quality for Large-Scale Software Systems. In *Proceedings of the International Conference on Automated Software Engineering*, pages 214–223. IEEE/ACM, 2005.

[157] Lanza, M. and Ducasse, S. Polymetric Views – A Lightweight Visual Approach to Reverse Engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, 2003.

[158] LaToza, T. D. and Myers, B. A. Hard-to-Answer Questions about Code. In *Proceedings of the Workshop on Evaluation and Usability of Programming Languages and Tools.* ACM, 2010.

[159] LaToza, T. D., Venolia, G., and DeLine, R. Maintaining Mental Models: A Study of Developer Work Habits. In *International Conference on Software Engineering*, pages 492–501. IEEE, 2006.

[160] LeBlanc, T. J. and Mellor-Crummey, J. M. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, 36:471–482, 1987.

[161] Lee, B., Robertson, G. G., Czerwinski, M., and Parr, C. S. CandidTree: Visualizing Structural Uncertainty in Similar Hierarchies. In *Proceedings of the International Conference on Human-Computer Interaction*, pages 250–263. Springer Verlag, 2007.

[162] Lehman, M. M. and Belady, L. A., editors. *Program Evolution: Processes of Software Change.* Academic Press Professional, Inc., 1985.

[163] Leon, D., Podgurski, A., and White, L. J. Multivariate Visualization in Observation-Based Testing. In *Proceedings of the International Conference on Software Engineering*, pages 116–125. IEEE, 2000.

[164] Letovsky, S. Cognitive Processes in Program Comprehension. *Journal Systems Software*, 7(4):325–339, 1987.

[165] Lewerentz, C., Simon, F., and Steinbrückner, F. Crococosmos. In *Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 72–76. Springer Verlag, 2002.

[166] Libby, R. Man Versus Model of Man: The Need For a Nonlinear Model. *Organizational Behavior and Human Performance*, 16:23–26, 1976.

[167] Liblit, B. R. *Cooperative Bug Isolation.* PhD thesis, University of California, Berkeley, 2004.

[168] Lienhard, A., Fierz, J., and Nierstrasz, O. Flow-Centric, Back-in-Time Debugging. In *Objects, Components, Models and Patterns*, volume 33, chapter 16, pages 272–288. Springer Verlag, 2009.

[169] Lim, E., Taksande, N., and Seaman, C. A Balancing Act: What Software Practitioners Have to Say about Technical Debt. *IEEE Software*, 29(6):22–27, 2012.

[170] Lindahl, M. Analyzing Real-Time Systems with Hardware Trace. *C++ Users Journal*, 2005.

[171] Liu, D., Marcus, A., Poshyvanyk, D., and Rajlich, V. Feature Location via Information Retrieval Based Filtering of a Single Scenario Execution Trace. In *Proceedings of the International Conference on Automated Software Engineering*, pages 234–243. ACM, 2007.

[172] Lixenfeld, C. Gescheiterte IT-Projekte. *Computerwoche*, 2013. `http://www.computerwoche.de/a/gescheiterte-it-projekte,2546218,2`.

[173] Lommerse, G., Nossin, F., Voinea, L., and Telea, A. The Visual Code Navigator: An Interactive Toolset for Source Code Investigation. In *Proceedings of the Symposium on Information Visualization*, pages 24–31. IEEE, 2005.

[174] Lonnberg, J., Ben-Ari, M., and Malmi, L. Visualising Concurrent Programs with Dynamic Dependence Graphs. In *International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–4. IEEE, 2011.

[175] Maletic, J. I., Marcus, A., and Collard, M. L. A Task Oriented View of Software Visualization. In *Proceedings of the International Workshop on Visualizing Software for Understanding and Analysis*, pages 32–40. IEEE, 2002.

[176] Marcus, A., Feng, L., and Maletic, J. I. 3D Representations for Software Visualization. In *Proceedings of the Symposium on Software Visualization*, pages 27–36. ACM, 2003.

[177] McCabe, T. J. A Complexity Measure. *IEEE Transactions on Software Engineering*, pages 308–320, 1976.

[178] McConnel, S. *Code Complete 2: A Practical Handbook of Software Construction.* Microsoft Press, 2004.

[179] McGee, F. and Dingliana, J. An Empirical Study on the Impact of Edge Bundling on User Comprehension of Graphs. In *Proceedings of the International Working Conference on Advanced Visual Interfaces*, pages 620–627, 2012. ACM.

[180] Mehner, K. JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs. In *Software Visualization*, pages 163–175, 2001.

[181] Mehner, K. *Trace-Based Debugging and Visualisation of Concurrent Java Programs with UML.* PhD thesis, Universität Paderborn, 2005.

[182] Meyer, M. and Wendehals, L. Selective Tracing for Dynamic Analyses. In *Proceedings of the International Workshop on Program Comprehension through Dynamic Analysis*, pages 33–37, 2005.

[183] Mili, R. and Steiner, R. Software Engineering - Introduction. In *Software Visualization*, pages 129–137, 2001.

[184] Moe, J. and Carr, D. Understanding Distributed Systems via Execution Trace Data. In *Proceedings of the International Workshop on Program Comprehension*, pages 60–67. IEEE, 2001.

[185] Mohammadi-Aragh, M. J. and Jankun-Kelly, T. J. MoireTrees: Visualization and Interaction for Multi-Hierarchical Data. In *Proceedings of the Conference on Visualization*, pages 231–238. Eurographics, 2005.

[186] Momotko, M. and Nowicki, B. Visualisation of (Distributed) Process Execution based on Extended BPMN. In *International Workshop on Database and Expert Systems Applications*, pages 280–286. IEEE, 2003.

[187] Monden, A., Nakae, D., Kamiya, T., ichi Sato, S., and ichi Matsumoto, K. Software Quality Analysis by Code Clones in Industrial Legacy Software. In *In Proceedings of the Symposium on Software Metrics*, pages 87–94. IEEE, 2002.

[188] Moreta, S. and Telea, A. Multiscale Visualization of Dynamic Software Logs. In *Proceedings of the Conference on Visualization*, pages 11–18. Eurographics, 2007.

[189] Muelder, C., Gygi, F., and Ma, K.-L. Visual Analysis of Inter-Process Communication for Large-Scale Parallel Computing. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1129–1136, 2009.

[190] Munzner, T., Guimbretière, F., Tasiran, S., Zhang, L., and Zhou, Y. TreeJuxtaposer: Scalable Tree Comparison Using Focus+Context with Guaranteed Visibility. *ACM Transactions on Graphics*, 22(3):453–462, 2003.

[191] Murphy-Hill, E. and Black, A. P. Refactoring Tools: Fitness for Purpose. *IEEE Software*, 25:38–44, 2008.

[192] Myers, B. A. Visual Programming, Programming by Example, And Program Visualization: A Taxonomy. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 59–66. ACM, 1986.

[193] Myers, B. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing*, 1:97–123, 1990.

[194] Myers, B. A. Improving Program Comprehension by Answering Questions. In *Proceedings of the International Conference on Program Comprehension.* IEEE, 2013.

[195] Nagel, W. E., Arnold, A., Weber, M., Hoppe, H.-C., and Solchenbach, K. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 12:69–80, 1996.

[196] Nielson, F., Nielson, H. R., and Hankin, C. *Principles of Program Analysis.* Springer Verlag, 2005.

[197] Noda, K., Kobayashi, T., and Agusa, K. Execution Trace Abstraction Based on Meta Patterns Usage. In *Proceedings of the Working Conference on Reverse Engineering*, pages 167–176, 2012.

[198] Northrop, L. Does Scale Really Matter? - Ultra-Large-Scale Systems Seven Years after the Study. In *Proceedings of the International Conference on Software Engineering*, 2013.

[199] Nutt, G., Griff, A., Mankovich, J., and McWhirter, J. Extensible Parallel Program Performance Visualization. In *Proceedings of the International Symposium on Modeling, Analysis, and Simulation of Computer Systems*, volume 0, pages 205–211, 1995. IEEE Computer Society.

[200] Oklobdzija, V. G., editor. *The Computer Engineering Handbook: Digital Systems and Applications (Second Edition).* CRC Press, 2008.

[201] Parnas, D. L. On the Criteria to be Used In Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[202] Parnas, D. L. Software aging. In *Proceedings of the International Conference on Software Engineering*, pages 279–287, 1994.

[203] Paulisch, F. N. and Tichy, W. F. EDGE: An Extendible Graph Editor. In Newbery Paulisch, F., editor, *The Design of an Extendible Graph Editor*, volume 704 of *Lecture Notes in Computer Science*, pages 133–151. Springer Verlag, 1993.

[204] Pavlopoulou, C. and Young, M. Residual Test Coverage Monitoring. In *Proceedings of the International Conference on Software Engineering*, pages 277–284. ACM, 1999.

[205] Pennington, N. Comprehension Strategies in Programming. In *Empirical Studies of Programmers: Second Workshop*, pages 100–113. Ablex Publishing Corp., 1987.

[206] Petroski, H. *Pushing the Limits: New Adventures in Engineering.* Vintage Books, 2005.

[207] Pigoski, T. M. and April, A. *Software Engineering Body of Knowledge*, chapter Software Maintenance, pages 6.1–6.16. IEEE, 2004.

[208] Poshyvanyk, D., Gueheneuc, Y.-G., Marcus, A., Antoniol, G., and Rajlich, V. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, 2007.

[209] Price, B. A., Baecker, R., and Small, I. S. A Principled Taxonomy of Software Visualization. *Journal of Visual languages and Computing*, 4(3):211–266, 1993.

[210] Price, B. A., Baecker, R., and Small, I. *Software Visualization - Programming as a Multimedia Experience*, chapter An Introduction to Software Visualization, pages 3–28. MIT Press, 1998.

[211] Purchase, H. Which Aesthetic Has the Greatest Effect on Human Understanding? *Lecture Notes in Computer Science*, 1353:248–261, 1997. http://eprints.gla.ac.uk/35804/.

[212] Pérez, J., Crespo, Y., Hoffmann, B., and Mens, T. A Case Study to Evaluate the Suitability of Graph Transformation Tools for Program Refactoring. *International Journal on Software Tools for Technology Transfer*, 12(3-4):183–199, 2010.

[213] Rajlich, V. and Wilde, N. The Role of Concepts in Program Comprehension. In *Proceedings of the International Workshop on Program Comprehension*, pages 271–278. IEEE, 2002.

[214] Reinders, J. *Intel Threading Building Blocks.* O'Reilly & Associates, Inc., 2007.

[215] Reiss, S. and Tarvo, A. Automatic Categorization and Visualization of Lock Behavior. In *Proceedings of the Working Conference on Software Visualization*. IEEE, 2013.

[216] Reiss, S. P. Visualizing the Java Heap. In *Proceedings of the International Conference on Software Engineering*, pages 251–254. ACM, 2010.

[217] Reiss, S. P. and Renieris, M. Encoding Program Executions. In *Proceedings of the International Conference on Software Engineering*, pages 221–230. IEEE, 2001.

[218] Reiss, S. P. and Renieris, M. *Software Visualization: From Theory to Practice*, chapter The BLOOM Software Visualization System, pages 311–357. Kluwer Academic Publishers, 2003.

[219] Renieris, M. and Reiss, S. Fault Localization With Nearest Neighbor Queries. In *In Proceedings of the Conference on Automated Software Engineering*, pages 30–39. IEEE, 2003.

[220] Reniers, D., Voinea, L., Ersoy, O., and Telea, A. A Visual Analytics Toolset for Program Structure, Metrics, and Evolution Comprehension. In *Proceedings of the International Workshop on Academic Software Development Tools and Techniques*. IEEE/ACM, 2010.

[221] Roberts, J. and Zilles, C. TraceVis: An Execution Trace Visualization Tool. In *Proceedings of the Workshop on Modeling, Benchmarking and Simulation*, pages 123–130, 2005.

[222] Roberts, J. C. Exploratory Visualization with Multiple Linked Views. In MacEachren, A., Kraak, M.-J., and Dykes, J., editors, *Exploring Geovisualization*. Amsterdam: Elseviers, 2004.

[223] Roche, J. M. Software Metrics and Measurement Principles. *SIGSOFT Software Engineering Notes*, 19(1):77–85, 1994.

[224] Roman, G.-C. and Cox, K. C. A Taxonomy of Program Visualization Systems. *IEEE Computer*, 26(12):11–24, 1993.

[225] Ronsse, M., De Bosschere, K., Christiaens, M., de Kergommeaux, J. C., and Kranzlmüller, D. Record/Replay for Nondeterministic Program Executions. *Communications of the ACM*, 46:62–67, 2003.

[226] Roock, S. and Lippert, M. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. John Wiley & Sons, Inc., 2006.

[227] Rosenholtz, R., Li, Y., Mansfield, J., and Jin, Z. Feature Congestion: A Measure of Display Clutter. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 761–770. ACM, 2005.

[228] Sander, G. Graph Layout through the VCG Tool. In *Proceedings of the DIMACS International Workshop on Graph Drawing*, pages 194–205. Springer Verlag, 1994.

[229] Sanders, J. and Kandrot, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2011.

[230] Sarkar, M. and Brown, M. H. Graphical Fisheye Views. *Communications of the ACM*, 37(12):73–83, 1994.

[231] Sato, T., Shizuki, B., and Tanaka, J. Support for Understanding GUI Programs by Visualizing Execution Traces Synchronized with Screen Transitions. In *Proceedings of the International Conference on Program Comprehension*, pages 272–275. IEEE, 2008.

[232] Schmidt, D. C., Stal, M., Rohnert, H., and Buschmann, F. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., 2nd edition, 2000.

[233] Schöbel, M. and Polze, A. A Runtime Environment for Online Processing of Operating System Kernel Events. In *Proceedings of the International Workshop on Dynamic Analysis*. ACM, 2009.

[234] Schwanke, R. W. and Hanson, S. J. Using Neural Networks to Modularize Software. *Machine Learning*, 15:137–168, 1994.

[235] Sharma, S. Real-Time Visualization of Concurrent Processes. In *Proceedings of the Joint International Conference on Vector and Parallel Processing*, pages 852–862. Springer-Verlag, 1990.

[236] Shneiderman, B., Bederson, B., and Wattenberg, M. The Treemap 4.0 Visualization System, 2008. `http://www.cs.umd.edu/hcil/treemap`.

[237] Shneiderman, B. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *Proceedings of the Symposium on Visual Languages*, page 336. IEEE, 1996.

[238] Shneiderman, B. and Mayer, R. Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Computer & Information Sciences*, 8(3):219–238, 1979.

[239] Sigovan, C., Muelder, C., and M, K.-L. Visualizing Large-Scale Parallel Communication Traces Using a Particle Animation Technique. In *Proceedings of the Conference on Visualization*. Eurographics, 2013.

[240] Silva, S., Madeira, J., and Santos, B. S. There is More to Color Scales than Meets the Eye: A Review on the Use of Color in Visualization. In *Proceedings of the International Conference Information Visualization*, pages 943–950. IEEE Computer Society, 2007.

[241] Simon, F., Steinbrückner, F., and Lewerentz, C. Metrics Based Refactoring. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 30–38. IEEE, 2001.

[242] Simpson, J. A. and Weiner, E., editors. *The Oxford English Dictionary*. Oxford University Press, xix edition, 1989.

[243] Smith, M. and Munro, M. Providing a User Customisable Tool for Software Visualisation at Runtime. In *Proceedings of the International Conference on Visualization, Imaging and Image Processing*, 2004.

[244] Soloway, E. and Ehrlich, K. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, 10(5):595–609, 1984.

[245] Sosa, M., Eppinger, S., and Rowles, C. The Misalignment of Product Architecture and Organizational Structure in Complex Product Development. *Management Science*, 50:1674–1689, 2004.

[246] Spence, R. *Information Visualization.* Addison-Wesley, 2001.

[247] Spinellis, D. *Code Reading: Open Source Perspective.* Addison-Wesley, 2003.

[248] Stasko, J. and Zhang, E. Focus+Context Display and Navigation Techniques for Enhancing Radial, Space-Filling Hierarchy Visualizations. In *Proceedings of the Symposium on Information Visualization*, pages 57–65. IEEE, 2000.

[249] Stasko, J. T. The PARADE Environment for Visualizing Parallel Program Executions: A Progress Report. Technical report, Georgia Institute of Technology, 1995.

[250] Stasko, J. and Patterson, C. Understanding and Characterizing Software Visualization Systems. In *Proceedings of the Workshop on Visual Languages*, pages 3–10. IEEE, 1992.

[251] Steinbrückner, F. and Lewerentz, C. Understanding Software Evolution with Software Cities. *Information Visualization*, 2012.

[252] Stone, J. M. Debugging Concurrent Processes: A Case Study. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 145–153. ACM, 1988.

[253] Storey, M.-A. D., Fracchia, F. D., and Mueller, H. A. Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualization. In *Proceedings of the International Workshop on Program Comprehension*, pages 17–28. IEEE, 1997.

[254] Storey, M.-A. *Software Visualization: From Theory to Practice*, chapter Designing a Software Exploration Tool Using a Cognitive Framework, pages 113–147. Kluwer Academic Publishers, 2003.

[255] Stroulia, E. and Systä, T. Dynamic Analysis for Reverse Engineering and Program Understanding. *SIGAPP Applied Computing Review*, 10(1):8–17, 2002.

[256] Sumner, W. N., Bao, T., and Zhang, X. Selecting Peers for Execution Comparison. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 309–319. ACM, 2011.

[257] Sutter, H. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3):202–210, 2005. `http://www.gotw.ca/publications/concurrency-ddj.htm`.

[258] Tak, S. and Cockburn, A. Enhanced Spatial Stability with Hilbert and Moore Treemaps. *IEEE Transactions on Visualization and Computer Graphics*, 19(1): 141–148, 2013.

[259] Telea, A. and Ersoy, O. Image-Based Edge Bundles: Simplified Visualization of Large Graphs. In *Proceedings of the Conference on Visualization*, pages 843–852. Eurographics, 2010.

[260] Telea, A. and Auber, D. Code Flows: Visualizing Structural Evolution of Source Code. *Computer Graphics Forum*, 27(3):831–838, 2008.

[261] Telea, A. and Voinea, L. Case Study: Visual Analytics in Software Product Assessments. In *Proceedings of the International Workshop on Visualizing Software for Understanding and Analysis*, pages 65–72. IEEE, 2009.

[262] Termeer, M., Lange, C. F. J., Telea, A., and Chaudron, M. R. V. Visual Exploration of Combined Architectural and Metric Information. In *Proceedings of the International Workshop on Visualizing Software for Understanding and Analysis*, pages 11–16. IEEE, 2005.

[263] Thayer, R., Pyster, A., and Wood, R. Major Issues in Software Engineering Project Management. *IEEE Transactions on Software Engineering*, 7(4):333–342, 1981.

[264] Tilley, S. R. The Canonical Activities of Reverse Engineering. *Annals of Software Engineering*, 9(1-4):249–271, 2000.

[265] Tu, Y. and Shen, H.-W. Visualizing Changes of Hierarchical Data Using Treemaps. *IEEE Transactions on Computer Graphics*, 13(6):1286–1293, 2007.

[266] Tufte, E. R. *The Visual Display of Quantitative Information.* Graphics Press, 2001.

[267] Van Wijk, J. J. and van de Wetering, H. Cushion Treemaps: Visualization of Hierarchical Information. In *Proceedings of the Symposium on Information Visualization*, pages 73–78. IEEE, 1999.

[268] Voigt, S., Bohnet, J., and Döllner, J. Enhancing Structural Views of Software Systems By Dynamic Information. In *Proceedings of International Workshop on Visualizing Software for Understanding and Analysis*, pages 47–50. IEEE, 2009.

[269] Voinea, L., Telea, A., and van Wijk, J. J. CVSscan: Visualization of Code Evolution. In *Proceedings of the Symposium on Software Visualization*, pages 47–56. ACM, 2005.

[270] Voinea, L., Telea, A., and van Wijk, J. J. EZEL: A Visual Tool for Performance Assessment of Peer-to-Peer File-Sharing Networks. In *Proceedings of the Symposium on Information Visualization*, pages 41–48. IEEE, 2004.

[271] von Mayrhauser, A. and Vans, A. M. Program Understanding Behavior During Adaptation of Large Scale Software. In *Proceedings of the International Workshop on Program Comprehension*, pages 164–172. IEEE, 1998.

[272] von Mayrhauser, A. and Vans, A. M. Program Comprehension During Software Maintenance and Evolution. *Computer*, 28(8):44–55, 1995.

[273] von Mayrhauser, A. and Vans, A. M. Program Understanding: Models and Experiments. *Advances in Computers*, 40:1–38, 1995.

[274] Waller, J., Wulf, C., Fittkau, F., Döhring, P., and Hasselbring, W. SynchroVis: 3D Visualization of Monitoring Traces in the City Metaphor for Analyzing Concurrency. In *Proceedings of the Working Conference on Software Visualization.* IEEE, 2013.

[275] Wang, W., Wang, H., Dai, G., and Wang, H. Visualization of Large Hierarchical Data by Circle Packing. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 517–520. ACM, 2006.

[276] Ware, C. *Information Visualization: Perception for Design.* Morgan Kaufmann Publishers, 3rd edition, 2012.

[277] Waters, R. G. and Chikofsky, E. Reverse Engineering: Progress Along Many Dimensions. *Communications of the ACM*, 37(5):22–25, 1994.

[278] Wettel, R. and Lanza, M. Visually Localizing Design Problems With Disharmony Maps. In *Proceedings of the Symposium on Software Visualization*, pages 155–164. ACM, 2008.

[279] Wheeler, K. and Thain, D. Visualizing Massively Multithreaded Applications with ThreadScope. *Concurrency and Computation: Practice and Experience*, 22:45–67, 2009.

[280] Wilde, N., Gomez, J. A., Gust, T., and Strasburg, D. Locating User Functionality in Old Code. In *Proceedings of the Conference on Software Maintenance*, pages 200–205. IEEE, 1992.

[281] Wirth, C., Prahofer, H., and Schatz, R. A Multi-Level Approach for Visualization and Exploration of Reactive Program Behavior. In *Proceedings of the International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–4. IEEE, 2011.

[282] Wirtz, G., Weske, M., and Giese, H. The OCoN Approach to Workflow Modeling in Object-Oriented Systems. *Information Systems Frontiers*, 3(3):357–376, 2001.

[283] Wirtz, G. and Zhang, K. Visual Methods for Parallel and Distributed Programming. *Journal of Visual Languages & Computing*, 12(2):123–125, 2001.

[284] Xie, S., Kraemer, E., Stirewalt, R. E. K., Dillon, L. K., and Fleming, S. D. Design and Evaluation of Extensions to UML Sequence Diagrams for Modeling Multithreaded Interactions. *Information Visualization*, 8(2):120–136, 2009.

[285] Yamaguchi, Y. and Itoh, T. Visualization of Distributed Processes Using "Data Jewelry Box" Algorithm. *Computer Graphics International Conference*, 0:162–169, 2003.

[286] Zaidman, A. *Scalability Solutions for Program Comprehension Through Dynamic Analysis*. PhD thesis, Universiteit Antwerpen, 2006.

[287] Zaki, O., Lusk, E., and Swider, D. Toward Scalable Performance Visualization with Jumpshot. *High Performance Computing Applications*, 13:277–288, 1999.

[288] Zhang, D., Han, S., Dang, Y., Lou, J.-G., Zhang, H., and Xie, T. Software Analytics in Practice. *IEEE Software*, 30(5):30–37, 2013.

[289] Zhang, K., editor. *Software Visualization: From Theory to Practice*. Kluwer Academic Publishers, 2003.

[290] Zhang, K. and Marwaha, G. Visputer - A Graphical Visualization Tool for Parallel Programming. *The Computer Journal*, 38(8):658–669, 1995.

[291] Zhao, C., Zhang, K., Hao, J., and Wong, W. E. Visualizing Multiple Program Executions to Assist Behavior Verification. In *Proceedings of the International Conference on Secure Software Integration and Reliability Improvement*, pages 113–122. IEEE, 2009.

[292] Zhao, J. Multithreaded Dependence Graphs for Concurrent Java Programs. In *Proceedings of International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 13–23. IEEE, 1999.

[293] Zimmermann, M., Perrenoud, F., and Schiper, A. Understanding Concurrent Programming Through Program Animation. In *Proceedings of the Technical Symposium on Computer Science Education*, pages 27–31. ACM, 1988.

Publications

[294] Beck, M., Trümper, J., and Döllner, J. A Visual Analysis and Design Tool for Planning Software Reengineerings. In *Proceedings of the International Workshop on Visualizing Software for Understanding and Analysis*, pages 54–61. IEEE, 2011.

[295] Hahn, S., Trümper, J., and Döllner, J. Visualization of Varying Hierarchies by Stable Layout of Voronoi Treemaps. In *Proceedings of the International Conference on Information Visualization Theory and Applications*, pages 50–58. SCITEPRESS, 2014.

[296] Karran, B., Trümper, J., and Döllner, J. SyncTrace: Visual Thread-Interplay Analysis. In *Proceedings (electronic) of the Working Conference on Software Visualization*, pages 1–8. IEEE, 2013.

[297] Lehmann, C., Trümper, J., and Döllner, J. Interactive Areal Annotations for 3D Treemaps of Large-Scale Software Systems. In *Proceedings (electronic) of the Workshop on Geovisualization*, pages 1–2, 2011.

[298] Limberger, D., Wasty, B. H., Trümper, J., and Döllner, J. Interactive Software Maps for Web-Based Source Code Analysis. In *Proceedings of the International Web3D Conference*. ACM, 2013.

[299] Trümper, J. and Döllner, J. Extending Recommendation Systems with Software Maps. In *Proceedings of the International ICSE Workshop on Recommendation Systems for Software Engineering (RSSE)*, pages 92–96. IEEE, 2012.

[300] Trümper, J., Bohnet, J., and Döllner, J. Understanding Complex Multithreaded Software Systems by Using Trace Visualization. In *Proceedings of the International Symposium on Software Visualization*, pages 133–142. ACM, 2010.

[301] Trümper, J., Bohnet, J., Voigt, S., and 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*, pages 325–330. IEEE, 2010.

[302] Trümper, J., Beck, M., and Döllner, J. A Visual Analysis Approach to Support Perfective Software Maintenance. In *Proceedings of the International Conference on Information Visualisation*, pages 308–315. IEEE, 2012.

[303] Trümper, J., Telea, A., and Döllner, J. ViewFusion: Correlating Structure and Activity Views for Execution Traces. In *Proceedings of the Conference on Theory and Practice of Computer Graphics*, pages 45–52. Eurographics, 2012.

[304] Trümper, J., Voigt, S., and Döllner, J. Maintenance of Embedded Systems: Supporting Program Comprehension Using Dynamic Analysis. In *Proceedings of the*

International Workshop on Software Engineering for Embedded Systems, pages 4396–4402. IEEE, 2012.

[305] Trümper, J., Döllner, J., and Telea, A. Multiscale Visual Comparison of Execution Traces. In *Proceedings of International Conference on Program Comprehension*, pages 53–62. IEEE, 2013.

List of Figures

List of Abbreviations

2D	2-dimensional, page 25
3D	3-dimensional, page 26
3D-CBV	3-Dimensional Circular Bundle View, page 103
D+D-CBV	Drag&Drop Circular Bundle View, page 105
BRec	Building Reconstruction, page 50
CBV	Circular Bundle View, page 6
CTM	Cushion Treemap, page 78
DAG	Directed Acyclic Graph, page 14
ERP	Enterprise Resource Planning, page 112
FBT	Function Boundary Trace, page 13
FP	Francotyp Postalia, page 108
GPU	Graphics Processing Unit, page 42
HEB	Hierarchical Edge Bundling, page 33
IBEB	Image-Based Edge Bundling, page 91
IDE	Integrated Development Environment, page 124
IO	Input/Output, page 5
LOC	Lines of Code, page 1
MDG	Multi-threaded Dependence Graph, page 30
MSV	Massive Sequence View, page 27
PDG	Program Dependence Graph, page 30
SD	Software Diagnostics Technologies, page 114
SLOC	Source Lines of Code, page 50
UML	Unified Modeling Language, page 26
XP	Extreme Programming, page 15

Ehrenwörtliche Erklärung
Declaration of Authenticity

Hiermit versichere ich, dass ich die vorliegende Dissertation ohne Hilfe Dritter und ohne Zuhilfenahme anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe. Die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I hereby declare that the thesis has been written by myself without any external unauthorized help and that it has not been previously presented in this or any similar form to any institution for evaluation.

Potsdam, den 29. Januar 2014

Jonas Trümper