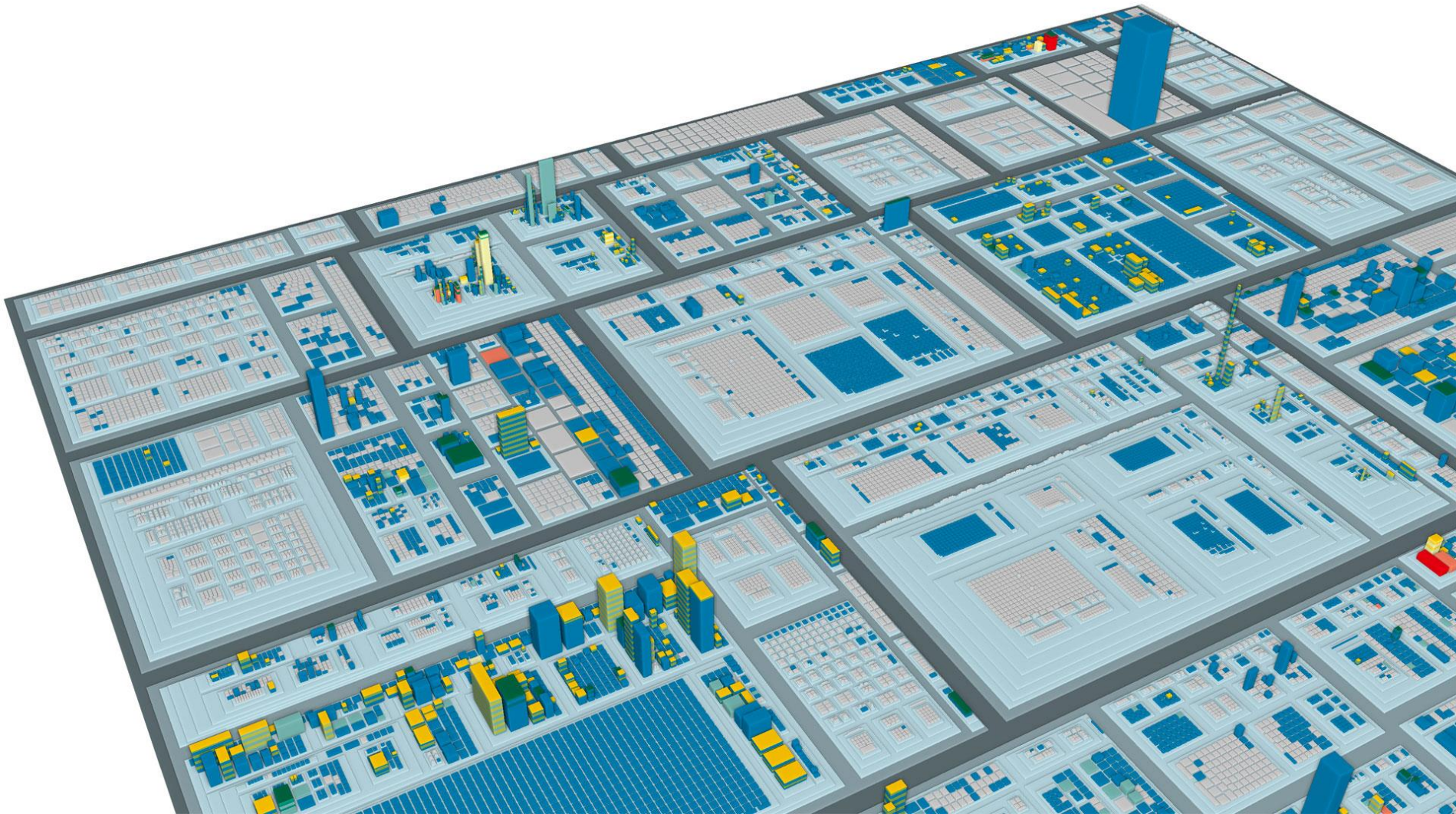
Automated Visual Software Analytics

OpenHPI – Prof. Dr. Jürgen Döllner – 2015



Code Instrumentation

Principles of Code Instrumentation
Static Code Instrumentation
Dynamic Code Instrumentation

Principles of Code Instrumentation

"What do I think of code that isn't commented, unit tested, or instrumented? Not very much. What do I think of a project's chances for success if the discipline to incorporate these elements of development into a project aren't present? Not very good. Four of five projects are over budget, not feature complete, buggy, and lucky if they ever see the light of day, and most projects that are done on fixed bids are money losers. The absence of project discipline is an unacceptable factor in poor software quality, and the remedy is like a ripe piece of low hanging fruit. ***Commenting, unit testing, and instrumenting code is the low hanging fruit of project discipline.****"*

– Paul Kimmel; „Commenting, Testing, and Instrumenting Code ", Dr. Dobb', Jan. 2011

Code Instrumentation

HPI Hasso Plattner Institut

Definition

Code instrumentation refers to methods and techniques that add extra code to a computer program to collect, log, and monitor information about program execution.

Applications

- Code instrumentation is widely used in tools for software profiling, optimization, testing, error detection, memory leak detection, and virtualization.
- Generally, there are two basic categories of code instrumentation techniques: **static and dynamic**.

Code Instrumentation

Static instrumentation techniques

- Add extra code to the source code at compile time
- Range from simple manual coding techniques to automated compiler or assembler-based instrumentation code editing
- Example: manually coded assert macros in C/C++:

```
#include <stdio.h>
#include <assert.h>

void print_number(int* myInt) {
    assert (myInt!=NULL);
    printf ("%d\n",*myInt);
}
```

Dynamic instrumentation techniques

- Add extra code to the binary before or at run-time (editing the binary)
- Can track dynamically linked libraries and indirect branches that are difficult to handle through static instrumentation

Static Code Instrumentation

Characteristics of Static Source Code Instrumentation

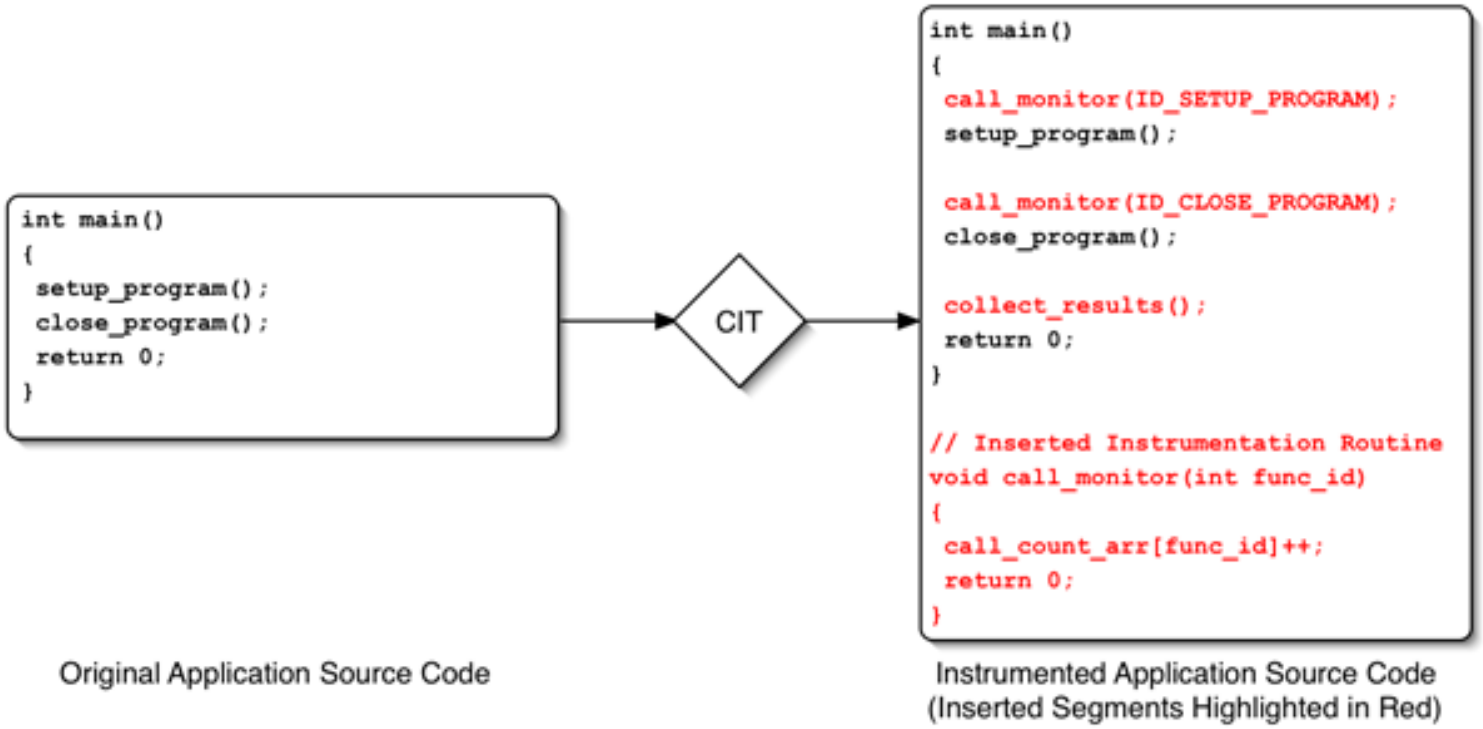
- Adds specific instrumentation source code to the source code of a software system
- Requires full access to the source code and its build environment as the system needs to be recompiled
- Executing the augmented system dumps report data

Examples: Rational Test RealTime (IBM), ATOM (digital)

Static Code Instrumentation

Example: Static Code Instrumentation for Counting Function Calls

Most elementary source code instrumentation by counting the number of function calls



http://www1.cse.wustl.edu/~jain/cse567-06/ftp/sw_monitors1/

Static Code Instrumentation

Example: Static Code Instrumentation by gprof

- gprof is a cross-platform performance analysis tool, collecting a wide array of useful function-level performance statistics from a targeted application.
- Output:
  - Flat Profile: It gives the total execution time spent in each function and its percentage of the total running time. Function call counts are also reported.
  - Textual Call Graph: Shows for each function who called it (parent) and who it called (child subroutines). gprof2dot converts the call graph from gprof into a graphical form.
- Note: Timing values are obtained by statistical sampling; the resulting data is not exact, rather a statistical approximation.

Static Code Instrumentation

Limitations of Static Code Instrumentation

- Increases both source code size and size of application binaries
- Increases execution time of the instrumented applications, which may cause different system behavior
- Cannot instrument external libraries, modules, and subsystems that are linked to the application

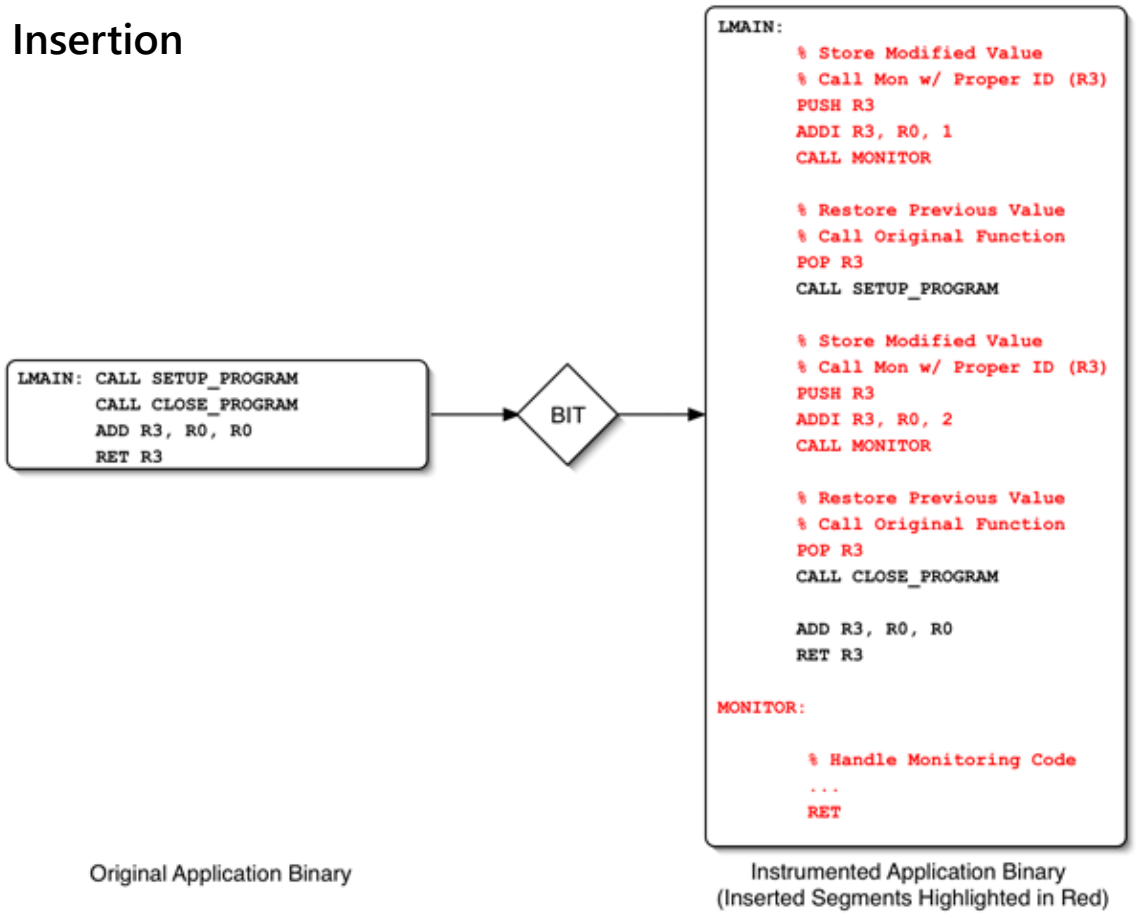
Dynamic Code Instrumentation

Dynamic Code Insertion

- Enables **injecting customized analysis** routines into arbitrary locations within a system binary to record a wide variety of performance data. Alterations can be inserted while the system is running.
- **Instrumentation alterations** can be focused on relevant parts or execution time frames so that highly accurate and focused statistics can be gathered.
- Increases the **breadth of behavior information** – library functions are typically loaded into the same memory space as the instrumented application at startup time and, therefore, appear as just another segment of code within the binary image (does not extend to kernel-level code because it resides in an isolated memory space).
- Example Tools: PIN, Valgrind

Dynamic Code Instrumentation

Dynamic Code Insertion



Original Application Binary

Instrumented Application Binary
(Inserted Segments Highlighted in Red)

http://www1.cse.wustl.edu/~jain/cse567-06/ftp/sw_monitors1/

Dynamic Code Instrumentation

Limitations of Binary Code Insertion

- Increases size of application binaries
- Increases execution time of the instrumented applications, which may cause different system behavior
- A “random” insertion of code into a binary can affect the flow of instructions through a processor pipeline, thus modifying the performance characteristics of the application
- Typically cannot gather statistics at the kernel level
- Analysis and instrumentation routines created with one tool are often incompatible with all others

Dynamic Code Instrumentation

Example: Valgrind

- Instrumentation framework for building dynamic analysis tools
- Examples of Valgrind-based tools:
  - memory error detector
  - thread error detector
  - cache and branch-prediction profiler
  - call-graph generating cache
  - branch-prediction profiler
  - heap profiler

www.valgrind.org

Dynamic Code Instrumentation

Example: Valgrind

Reports

- Use of uninitialized memory
- Accessing memory after it has been freed
- Accessing off the end of a dynamically allocated block
- Accessing inappropriate areas on the stack
- Memory leaks (pointer to dynamically allocated memory that are lost forever)
- Passing uninitialized and/or unaddressable memory to system calls
- Mismatched use of malloc/new/new[] vs. free/delete/delete[]
- Some misuses of POSIX pthreads

Dynamic Code Instrumentation

```
user@ubuntu:~$ valgrind ./memleak_prog
==7265== Memcheck, a memory error detector
==7265== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==7265== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==7265== Command: ./memleak_prog
==7265==
==7265==
==7265== HEAP SUMMARY:
==7265==     in use at exit: 45 bytes in 1 blocks
==7265==   total heap usage: 1 allocs, 0 frees, 45 bytes allocated
==7265==
==7265== LEAK SUMMARY:
==7265==    definitely lost: 45 bytes in 1 blocks
==7265==    indirectly lost: 0 bytes in 0 blocks
==7265==      possibly lost: 0 bytes in 0 blocks
==7265==    still reachable: 0 bytes in 0 blocks
==7265==         suppressed: 0 bytes in 0 blocks
==7265== Rerun with --leak-check=full to see details of leaked memory
==7265==
==7265== For counts of detected and suppressed errors, rerun with: -v
==7265== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
user@ubuntu:~$
```

Example: Valgrind

- Applications run approximately 20-50x slower and use 2.5x+ memory
  - 9 bits for every byte of memory to track validity and addressability
- Plus other overhead (tracking memory block owners, code cache, …)

"There is little reason not [to] use it—use it early, use it often"

– L. A. Tuura; "Using Valgrind", 2002

Summary

Code instrumentation is based on adding extra code to a computer program that aims at collecting, logging, and monitoring data about the program execution. It can observe **functional aspects** (such as function calls, their parameters and returned values) and **data aspects** (such as variable value changes, memory allocation/deallocation, etc.).

Code instrumentation enables to extract **information about the actual system behavior**, derived from and limited to the extent of sample runs. The resulting execution trace data can be analyzed to obtain information about call graphs, test coverage, performance measures, memory consumption and leaks, and other aspect of system behavior.

Code instrumentation serves as a key technology for **software analytics** to obtain information about behavioral aspects of complex software systems and related software quality aspects.

Code Instrumentation

Principles of Code Instrumentation
Static Code Instrumentation
Dynamic Code Instrumentation

Terminology

Code Instrumentation

Code instrumentation refers to methods and techniques that add extra code to a computer program to collect, log, and monitor information about program execution.

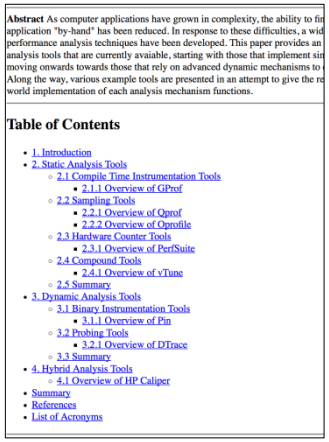
Static Code Instrumentation

Static code instrumentation is a category of code instrumentation that adds extra code to the source code of a software system, that is, instrumentation code is infiltrated at compile time.

Dynamic Code Instrumentation

Dynamic code instrumentation is a category of code instrumentation that adds extra code to the binaries a software system, that is, instrumentation code is infiltrated at run-time.

Further Reading

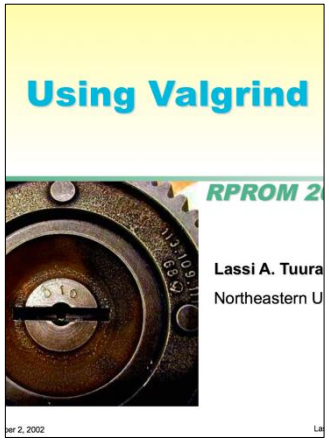


Abstract As computer applications have grown in complexity, the ability to fin application "by-hand" has been reduced. In response to these difficulties, a wid performance analysis techniques have been developed. This paper provides an analysis tools that are currently avaiable, starting with those that implement sir moving onwards towards those that rely on advanced dynamic mechanisms to Along the way, various example tools are presented in an attempt to give the re world implementation of each analysis mechanism functions.

Table of Contents

Justin Thiel; "An Overview of Software Performance Analysis Tools and Techniques: From GProf to Dtrace"

http://www1.cse.wustl.edu/~jain/cse567-06/ftp/sw_monitors1



Lassi A. Tuura; "Using Valgrind", PPT, Northeasten University, Boston, 2002

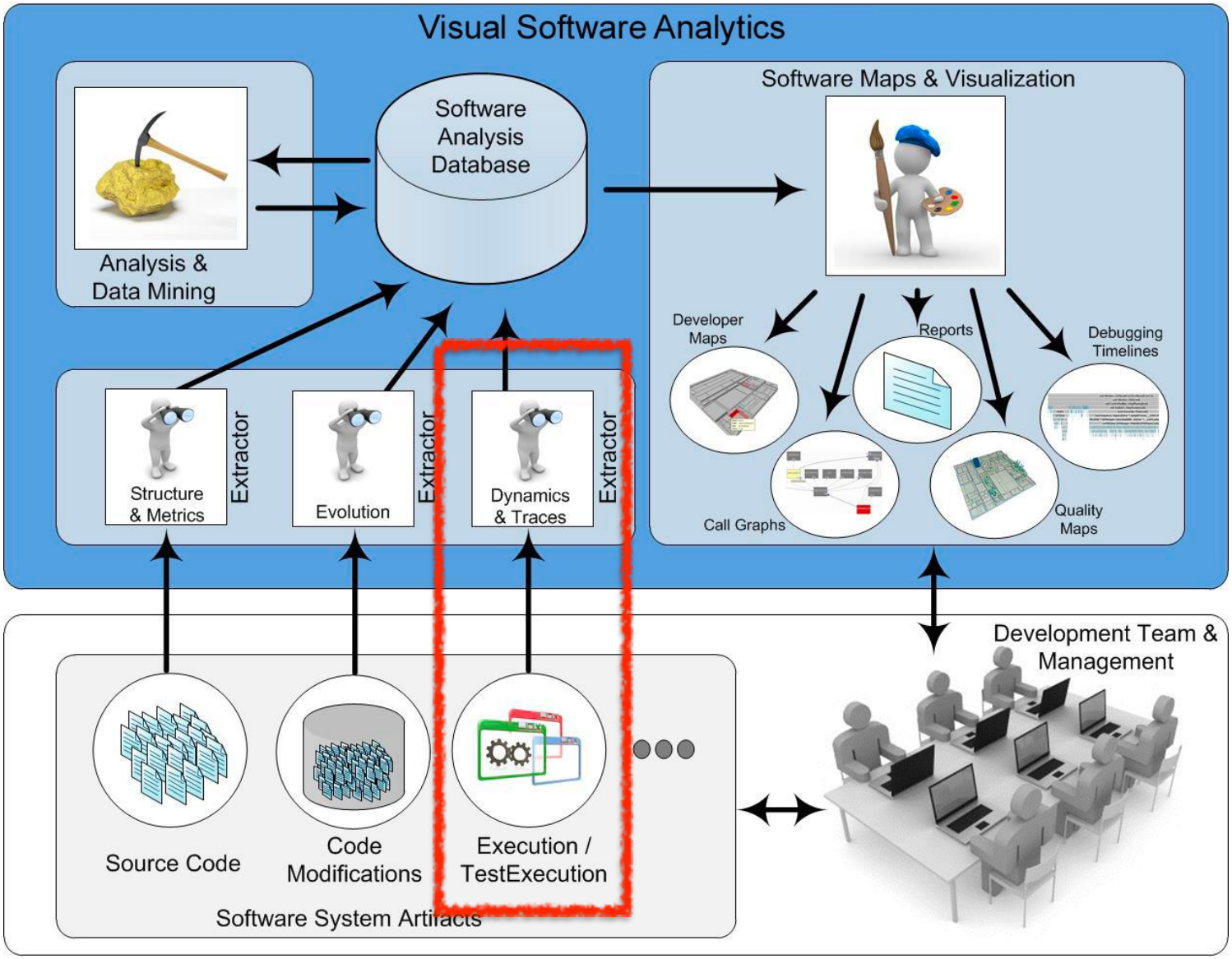
Remarks

- Code instrumentation is an elementary technique to control, monitor, and test complex software systems with respect to the actual system behavior.
- Dynamic code instrumentation tools show a strong technological progress. Most of them support most important programming languages and platforms.
- As a key source of information about system behavior they deliver key information to software analytics such as module or function coverage, test coverage, system and energy performance characteristics.
- The LLVM (Low-Level Virtual Machine) project offers instrumentation tools and evaluation tools (lldb, llvm-cov, lcov, …).

Execution Tracing

Principles of Execution Tracing
Traces and Call Graphs
Logging System Behavior

Applications of Execution Traces



Principles of Execution Tracing

HPI Hasso Plattner Institut

Definition

Execution tracing is a general form of logging information about the execution of a software system during its execution.

Characteristics

- Execution tracing captures **actual system behavior** that is recorded while the system is executed.
- Execution traces reveal what the system was doing while it was executed for the underlying **finite, captured system run**.
- The **granularity of recorded behavior** depends on the applied execution tracing technique (e.g., full code instrumentation, partial logging functions).

Principles of Execution Tracing

Characteristics

- **Execution tracing analysis** is key to
  - investigate the system behavior based on sampled system executions
  - localize errors or other functional deficiencies
  - understand interaction of system functions, components, and subsystems
  - understand interaction between threads
  - understand control flow
  - compute test coverage and profiling information
- Execution tracing
  - is a general-purpose means for **software quality measurements and analysis**
  - is a general source of information of system behavior used by **software analytics**

Traces and Call Graphs

Control Flow

A *control flow* denotes a sequence in which operations are performed during the execution of a computer program.

Understanding the Control Flow

- Execution tracing provides behavior information that can be used to understand the control flow of modules based on and limited to the sample system runs.
- Knowledge of the control flow is essential for most software maintenance tasks and, hence, has an enormous impact on software development effectiveness and behavioral analysis.

Traces and Call Graphs

Trace

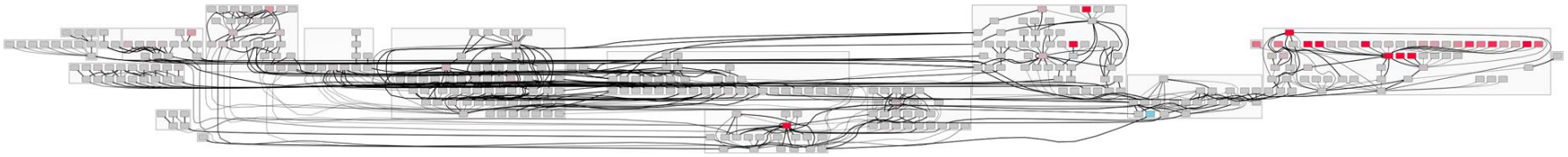
A *trace* is a record of the execution of a computer program that contains all or selected function calls performed and, optionally, the values of parameters passed to and returned by each function.

Traces and Call Graphs

Call Graph

A *call graph* is a hierarchical abstract representation of a sequence of functions, e.g., called during the execution of a system. The graph's nodes correspond to functions (or another entities being called) and calls are represented by edges between nodes.

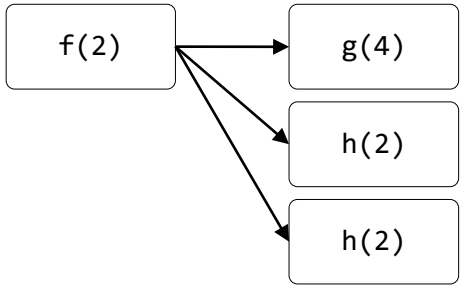
Call graphs with a low number nodes and edges can be depicted by classical node-edge graphs.

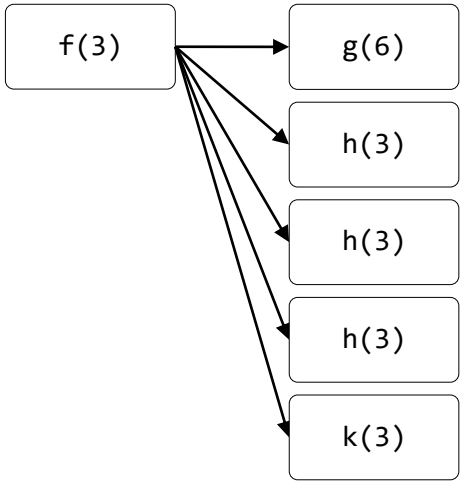


Traces and Call Graphs

Trace Example:

```
01   void f(int i) {
02      g(i * 2);
03      for (int j = 0; j < i; ++j)
04          h(i);
05      if (i > 2) k(i);
06   }
```

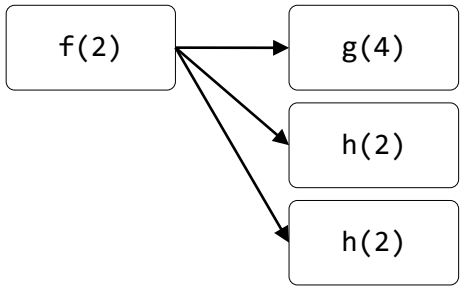


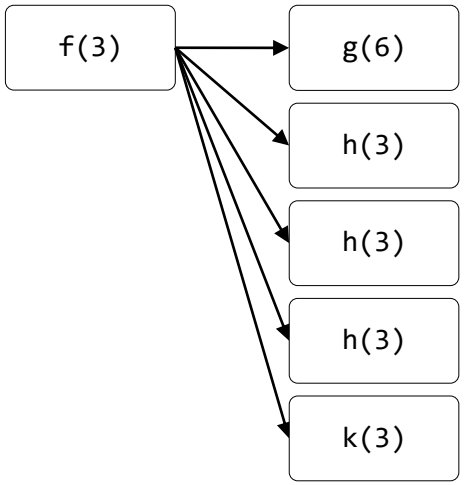


Traces and Call Graphs

Trace Example:

```
01   void f(int i) {
02      g(i * 2);
03      for (int j = 0; j < i; ++j)
04          h(i);
05      if (i > 2) k(i);
06   }
```





A sample execution of **`f(2)`** generates the trace data:

$((t_0, \text{entry f(2)}), (t_1, \text{entry g(4)}), (t_2, \text{exit g()}), (t_3, \text{entry h(2)}),$
$(t_4, \text{exit h()}), (t_5, \text{entry h(2)}), (t_6, \text{exit h()}), (t_7, \text{exit f()}))$

A sample execution of **`f(3)`** generates the trace data:

$((t_0, \text{entry f(3)}), (t_1, \text{entry g(6)}), (t_2, \text{exit g()}), (t_3, \text{entry h(3)}), (t_4, \text{exit h()}), (t_5, \text{entry h(3)}),$
$(t_6, \text{exit h()}), (t_7, \text{entry h(3)}), (t_8, \text{exit h()}), (t_9, \text{entry k(3)}), (t_{10}, \text{exit k()}), (t_{11}, \text{exit f()}))$

Logging System Behavior

Execution Trace Data Characteristics

- Execution trace data of real program runs gets **extremely large** in terms of recorded statements and corresponding storage requirements.
  - Even runs of a few seconds can generate hundreds of millions trace data entries, depending on the detail level at which tracing takes place.
  - The execution of recursive and iterative algorithms is mapped unfolded to trace data.

- Execution trace data needs to be filtered and generalized. Examples:
  - Calls to standard library functions or other black-box modules can be pruned
  - Iterative and recursive calls can be automatically detected and folded

Logging System Behavior

Limitations

- Execution tracing can have a varying degree of **performance decrease** depending on the code instrumentation used.

- Getting execution trace data on a **customer machine** depends on the customer's willingness to install the tracing-enabled version and handing over the trace data.
- Execution tracing data can include **sensitive information** about a product's source code (e.g., if parameters are traced and stored).

- The **information value** of trace data depends on the correlation of execution paths at run-time and the systems' use-cases of interest.

Summary

Execution tracing is a fundamental and powerful method to log system behavior while executing the system. The recorded trace data is the basis for system behavior analysis; for example, the data serves to reconstruct control flow graphs or to compute test coverage.

Execution trace data can store, depending on the tracing technique used, **detailed information of the sequence of instructions executed**, such as function name, time-of-entry, time-of-exit, values of parameters, values of variables and returned values.

Logging system behavior, by its nature, generates **large data sets**. They are not suited to be inspected manually in comparison to tracing used in debugging, which relies on a manual step-by-step inspection. Execution trace data are supposed to be analyzed, evaluated, and visualized by methods of **visual software analytics**.

Execution Tracing

Principles of Execution Tracing
Traces and Call Graphs
Logging System Behavior

Terminology

Execution Tracing

Execution tracing is a general form of logging information about the execution of software system during its execution.

Trace

A *trace* is a record of the execution of a computer program that contains all or selected function calls performed and, optionally, the values of parameters passed to and returned by each function.

Terminology

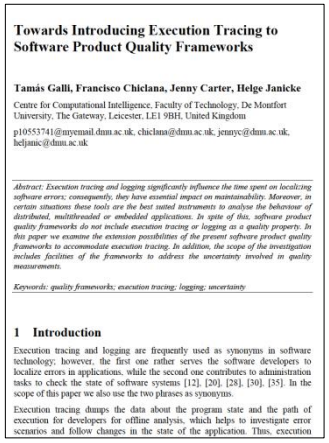
Control Flow

A *control flow* denotes a sequence in which operations are performed during the execution of a computer program.

Call Graph

A *call graph* is a formal representation of the calling sequence of a computer program. A node of a call graph represents a function; an edge represents the invocation of a function call, i.e., it defines a caller-callee relationship.

Further Reading



Towards Introducing Execution Tracing to Software Product Quality Frameworks

Tamás Galli, Francisco Chiclana, Jenny Carter, Helge Janicke

Centre for Computational Intelligence, Faculty of Technology, De Montfort University, The Gateway, Leicester, LE1 9BH, United Kingdom

p10553741@myemail.dmu.ac.uk, chiclana@dmu.ac.uk, jennyc@dmu.ac.uk, heljanic@dmu.ac.uk

Abstract: Execution tracing and logging significantly influence the time spent on localizing software errors; consequently, they have essential impact on maintainability. Moreover, in certain situations these tools are the best suited instruments to analyse the behaviour of distributed, multithreaded or embedded applications. In spite of this, software product quality frameworks do not include execution tracing or logging as a quality property. In this paper we examine the extension possibilities of the present software product quality frameworks to accommodate execution tracing. In addition, the scope of the investigation includes facilities of the frameworks to address the uncertainty involved in quality measurements.

Keywords: quality frameworks; execution tracing; logging; uncertainty

1 Introduction

Execution tracing and logging are frequently used as synonyms in software technology; however, the first one rather serves the software developers to localize errors in applications, while the second one contributes to administration tasks to check the state of software systems [12], [20], [28], [30], [35]. In the scope of this paper we also use the two phrases as synonyms.

Execution tracing dumps the data about the program state and the path of execution for developers for offline analysis, which helps to investigate error scenarios and follow changes in the state of the application. Thus, execution

Tamás Galli, Francisco Chiclana, Jenny Carter, Helge Janicke: "Towards Introducing Execution Tracing to Software Product Quality Frameworks", Acta Polytechnica Hungarica Vol. 11, No. 3, 2014.

Remarks

- Execution tracing is a well suited and essential instrument to analyze the behavior of distributed, multithreaded, and embedded applications; there is almost no chance to understand their control flows by means of static analysis.
- Execution tracing for a representative collection of tests provides key information about performance aspects (profiling), but also about code coverage. It can even be used to analyze specific software properties such as energy consumption.
- To gain insights into execution trace data, the data has to be post-processed by filtering all irrelevant parts (e.g., standard libraries, black boxes), by compressing technical sequences (e.g., operator resolution, type conversions, etc.), and by generalizing highly repeated calls (e.g., recursions, iterations). Commonly, techniques from data mining are adapted for this purpose.

Remarks

- Execution tracing and other logging functionality represent an important and commonly found part of the functionality of any complex software system.
- Approximately up to 25% of source code of complex software systems deal with execution tracing, logging, constraint/parameter checking, assertions, or other forms of capturing and reporting run-time behavior aspects.
- Execution tracing and logging functionality are typically not fully included in product versions of software systems.

Execution Trace Visualization

Principles of Execution Trace Visualization

Icicle Plots

Structure and Activity View Fusion

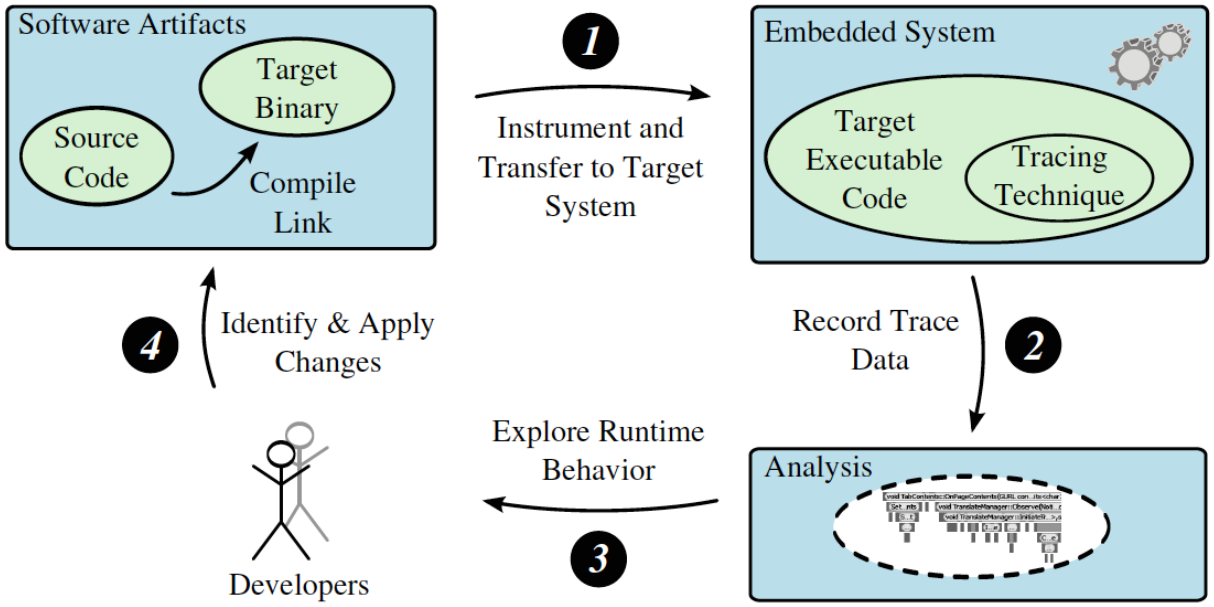
Principles of Execution Trace Visualization

Objectives

- Visualizing execution trace data aims at obtaining insights related to **actual activities and control flows** that take place during system execution.
- Key challenges for execution trace visualization include:
  - **Call graphs** can hardly be visualized by edge-node diagrams due to their size; node-edge diagrams practically cannot be used.
  - The **temporal order** is an essential information dimension but, by its nature, difficult to express by static depictions.
  - The **correlation to components of the hierarchical system implementation** is required in order to improve program understanding, but it is difficult to integrate both aspects in a single visualization technique.

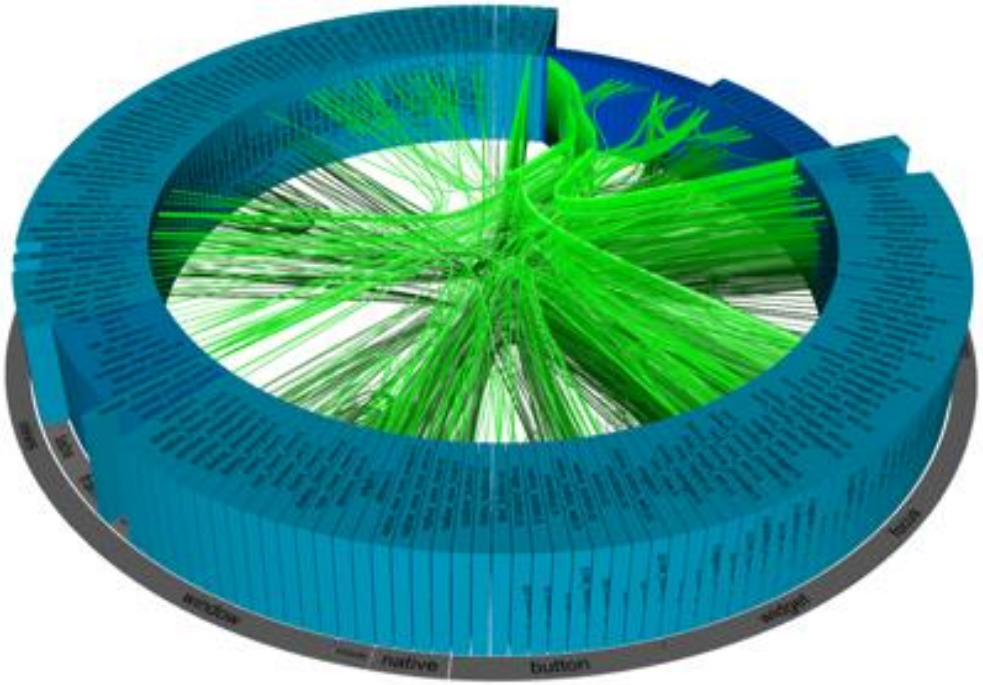
Principles of Execution Trace Visualization

Example: Exploring run-time behavior of embedded systems by execution trace visualization



Example of how tracing can facilitate software-maintenance tasks: (1) instrumentation, (2) trace recording at run-time, (3) visual analysis of trace data – developers gaining insights, (4) applying necessary changes to software artifacts.

Principles of Execution Trace Visualization



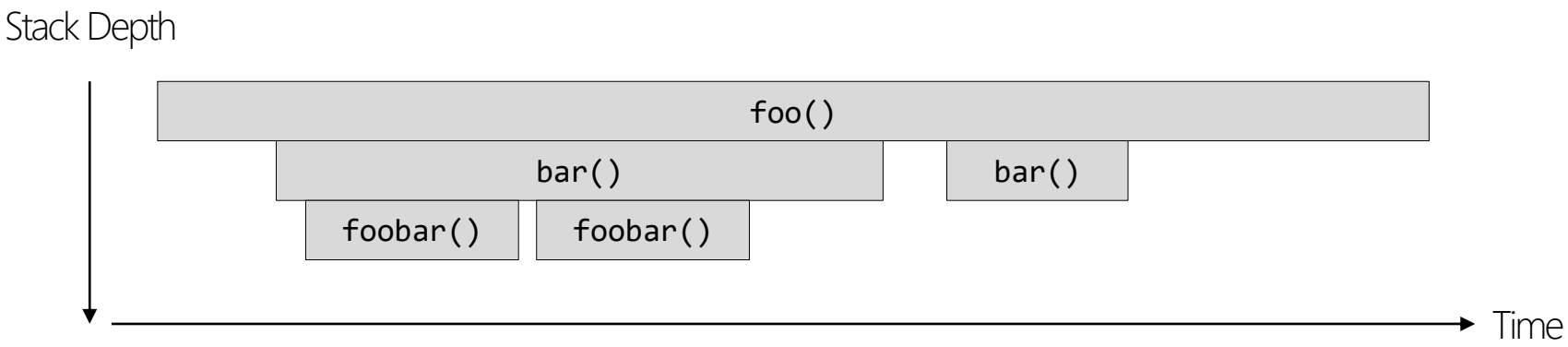
Execution Trace Visualization by Means of a Hierarchical Circular Bundle View

- Calls are represented by edges of the HCBV
- The height of the plane corresponds to the progress of time
- Disadvantages: visual occlusion, clutter, timely ordering difficult to differentiate

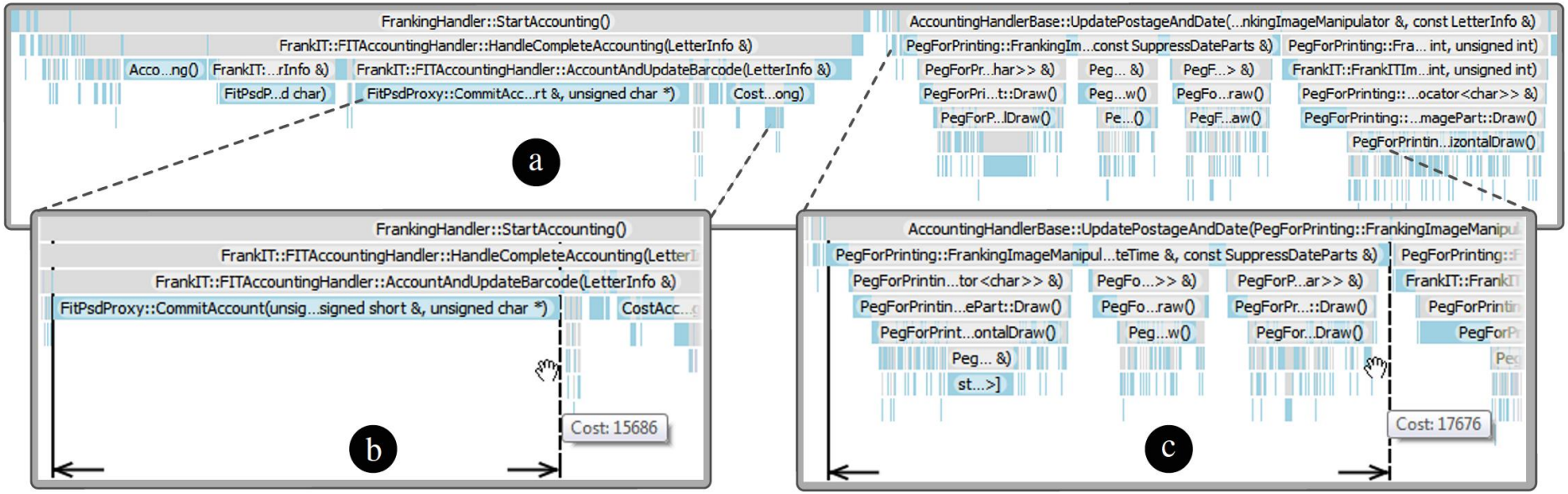
Icicle Plots

Execution Trace Visualization by Icicle Plots

- The call hierarchy is mapped by an icicle plot
- Time progresses from left to right
- Function entry corresponds to the left bar edge and function exit to right bar edge.
- The stack depth corresponds to the call depth



Icicle Plots



Execution Trace Visualization by Icicle Plots

a) Visualization of a thread's trace data captured during execution of a sample system. Stack grows from top to bottom, time progresses from left to right.

b) and c) Detailed views on two phases, measuring execution cost (in µs).

J. Trümper et al., 2012

Structure and Activity View Fusion

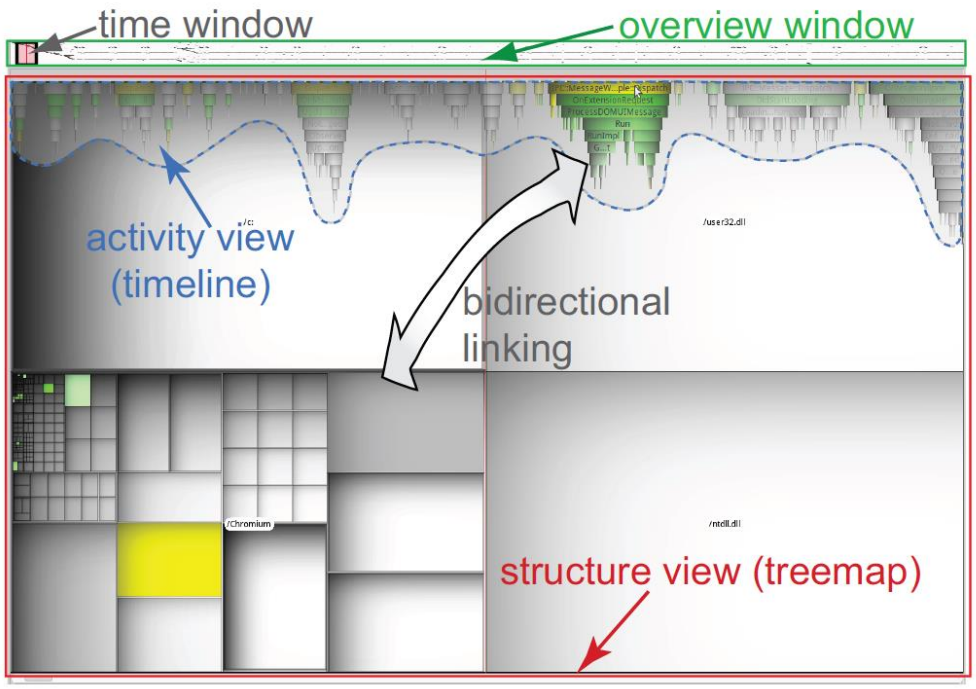
Combined Activity and Structure Visualization by View Fusion

Program comprehension through dynamic analysis requires combining both structure and activity insights.

Correlating structure with activity enables insights to answer questions such as:

- finding high-activity packages
- mapping execution phases to program module structures
- reasoning about performance problems at system component level

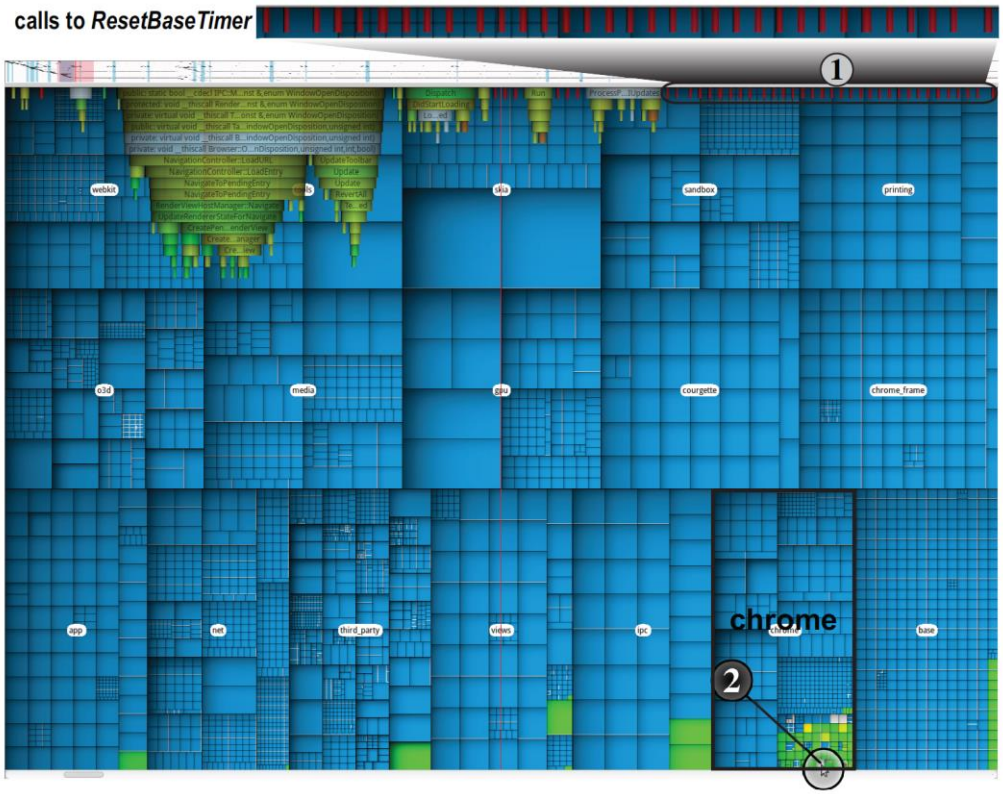
J. Trümper et al., 2012



Structure and Activity View Fusion

Combined Activity and Structure Visualization by View Fusion

- Call count and call duration are often used to find how actively packages take part in the execution of specific functionality.
- Call count mapping: activity view; Call duration mapping: structure view
- Blue treemap cells indicate uninvolved system parts



J. Trümper et al., 2012

Summary

The aim of **visualizing execution trace data** is to understand actual system behavior and actual control flows of a software system during its execution. Visualization of execution trace data has to express the temporal order as a principle information dimension required for comprehending system behavior.

Icicle plots serve as robust, interactive, and scalable visualization technique that allows for exploring and analyzing execution trace data. They can be correlated with structural information of the hierarchical system implementation by fusing both, **activity view and structural view**.

In **software analytics**, execution traces are key information sources of system behavior that are required by manifold tasks such as understanding control flows, application performance, detecting mutated system behavior. For that reason, visualization techniques for execution traces commonly are part of software analytics frameworks.

Execution Trace Visualization

Principles of Execution Trace Visualization

Icicle Plots

Structure and Activity View Fusion

Terminology

Execution Trace Visualization

Execution trace visualization provides interactive depictions of execution trace data; it relies on a visual mapping of call graph data.

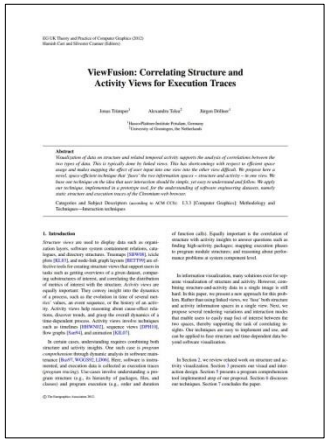
Icicle Plot

An *icicle plot* is a diagram type that depicts nodes of a hierarchy as bars, whereby the horizontal extent typically relates to a node's attribute, and child nodes are nested on the vertical dimension. (When used for execution trace visualization, the extent is mapped to activity time, and the nesting depicts the function call stack.)

Structure and Activity View Fusion

The *structure and activity view fusion* is a specialized type of execution trace visualization that interactively maps both activity information (execution trace data) and correlated structural information (module structure), combined in a single view.

Further Reading



ViewFusion: Correlating Structure and Activity Views for Execution Traces

Jonas Trümper, Alexandru Telea, Jürgen Döllner; "*ViewFusion: Correlating Structure and Activity Views for Execution Traces*", Theory and Practice of Computer Graphics, 2012

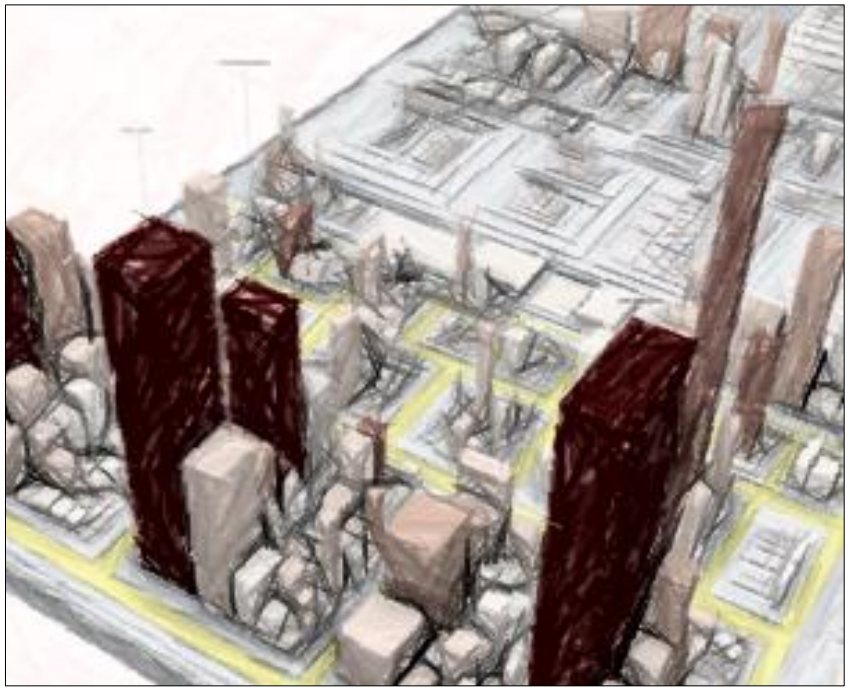
Maintenance of Embedded Systems: Supporting Program Comprehension Using Dynamic Analysis

Jonas Trümper, Stefan Voigt, Jürgen Döllner; "*Maintenance of Embedded Systems: Supporting Program Comprehension Using Dynamic Analysis*", In Proc. IEEE ICSE Workshop on Software Engineering for Embedded Systems (SEES), pages 58-64, 2012

Remarks

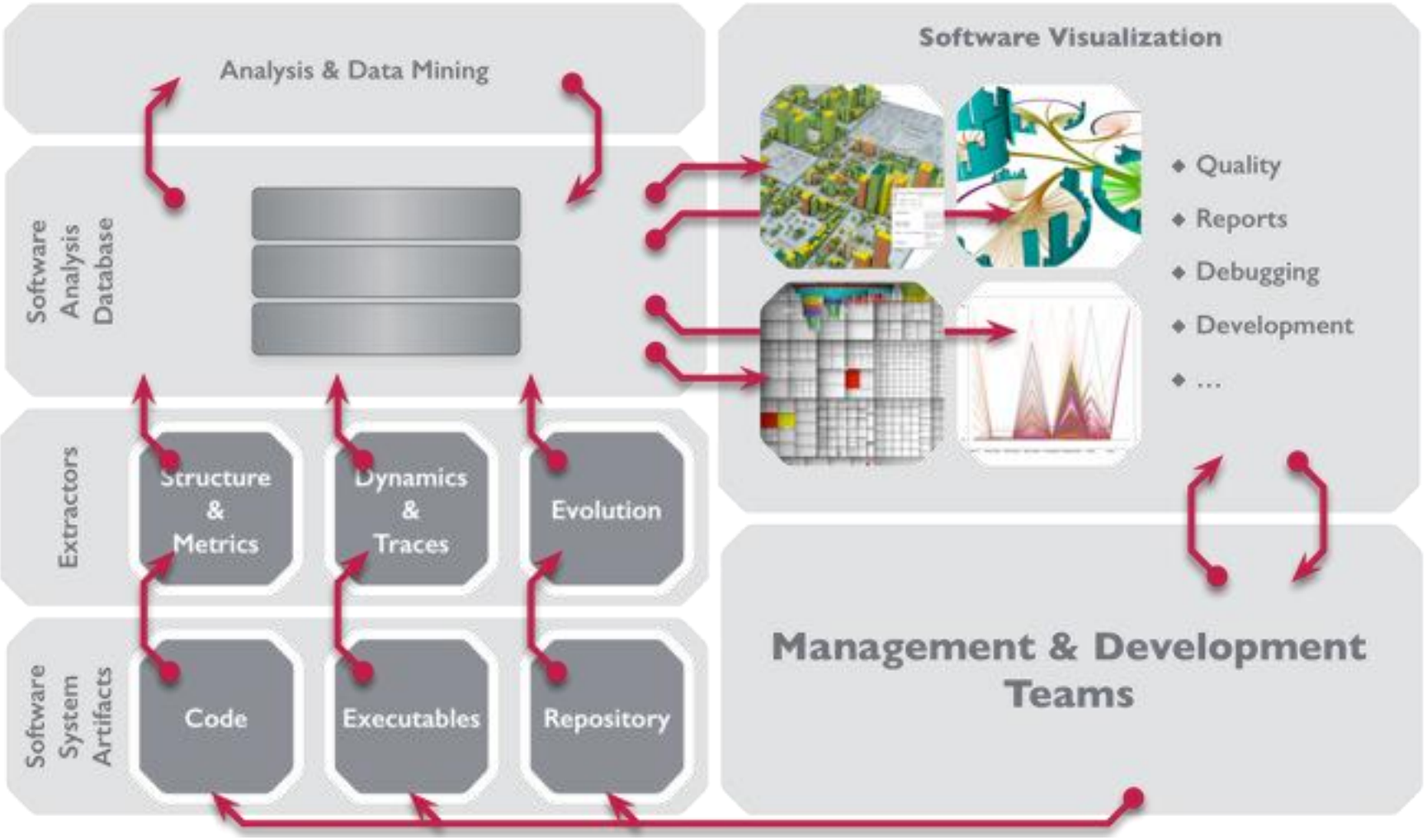
- Visualizing execution trace data represents one big challenge for visual software analytics as execution trace data tends to be large, and time-based data is inherently difficult to visualize by static media.
- Icicle plots, optionally combined with structure views, provide a powerful general-purpose visualization technique to interactively explore execution trace data. They serve as key instruments in software analytics to discover activity-related phenomena.
- Using execution trace visualization for exploration of multi-threaded system behavior is still a challenge (Example: Google Chrome Timeline Visualization)

Software Development Effectiveness

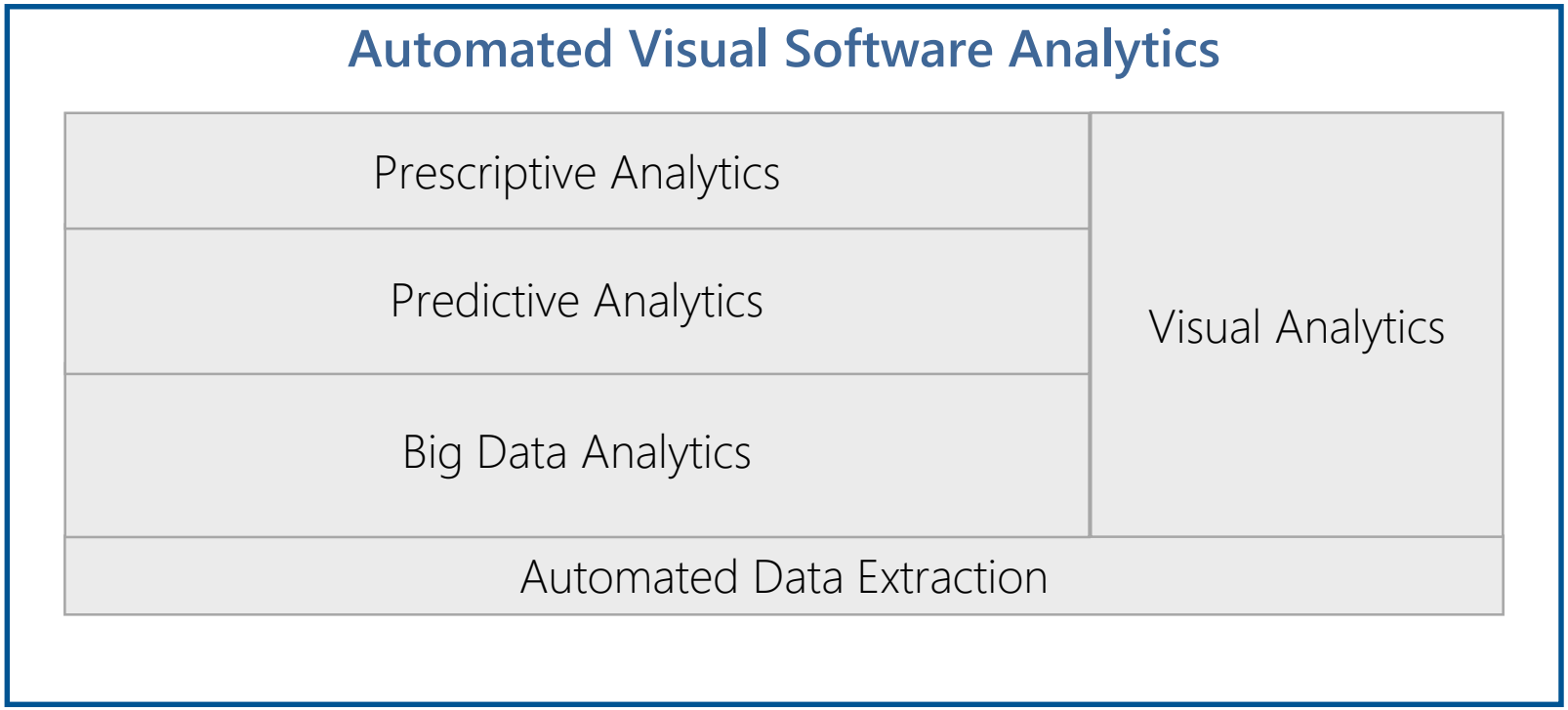


Software Analytics Workflow
Elements of Software Analytics
Impacts of Software Analytics

Software Analytics Workflow



Elements of Software Analytics



Elements of Software Analytics

Automated Data Extraction

Data gathering, extraction, integration, and analysis is largely performed in an automatic way and, thereby, can be efficiently applied to large-scale software engineering projects.

Big Data Analytics

Static system information, dynamic system information, and evolution-related system information –monitored, analyzed, and evaluated over time– are the big data specific to software engineering. Software analytics adapts big data analytics methods and tools to this software engineering data.

Elements of Software Analytics

Visual Analytics

A key conceptual element represents visual analytics, that is, the users and their visual capabilities are actively involved and integrated into software analytics processes and form part of the exploration-analysis-evaluation system.

Predictive Analytics

Prediction within large software engineering data is a key capability of software analytics as it helps to understand, to forecast, and to estimate complex situations by correlating appropriate key information.

Elements of Software Analytics

Prescriptive Analytics

Within the domain of software engineering, we can systematically integrate prescriptive analytics to derive effective actionable insights for professionals.

Software analytics is shaped by its "analytics nature", integrating different analytics disciplines, adapting and extending these to the specific needs of the software engineering domain.

Impacts of Software Analytics

“How does software analytics impact on software development effectiveness?”

Impacts of Software Analytics

“How does software analytics impact on software development effectiveness?”

Software Development Perspective

- Providing tool-based support for designing software transformation processes such as reengineering, refactoring, splitting, merging, outsourcing, …
- Enabling effective comprehension of past and ongoing software development processes and activities in large software projects and teams
- Effective at revealing risks and uncommon phenomena based on correlating key performance indicators and software development data
- Software analytics can be applied as a kind of "software engineering radar" and early warning system for development-related processes and risks

Impacts of Software Analytics

“How does software analytics impact on software development effectiveness?”

Software Quality Perspective

- Revealing, exploring, and monitoring key quality aspects of complex software-based systems in a systematic framework
- Open for project specific, team specific, or domain specific metrics as they are just more information dimensions that can be combined and correlated by forming corresponding software map themes
- Software analytics can be applied as a kind of “software engineering radar” and early warning system for software quality and “health”

Impacts of Software Analytics

“How does software analytics impact on software development effectiveness?”

Management Perspective

- Creating an efficient communication medium for managers, software engineers, and other stakeholders, building a common ground for transparent information provision, monitoring, and decision support
- Providing a systematic way of getting insights into massive, interwoven system and process information, based on “information cartography”
- Enabling a holistic view on software engineering artifacts and processes – correlating non-technical and technical information such as developer activities, development costs, customer relevance, business processes, …

Impacts of Software Analytics

“How does software analytics impact on software development effectiveness?”

Methodology Perspective

- Software analytics concepts, methods, techniques, and tools respect how stakeholders in software engineering actually develop software and manage related processes – they do not postulate specific forms of programming language usage, formal modeling, process designs, etc. and they do not have any assumptions about them.
- Software analytics is a non-normative approach that adapts to deeply optimized, settled projects, teams, and cultures in software engineering.
- Drastically extends the repertoire of methods and tools for large-scale, long-term effective and efficient software developments – a key factor to move from *heuristics-driven software engineering* to *data-driven to software engineering*.

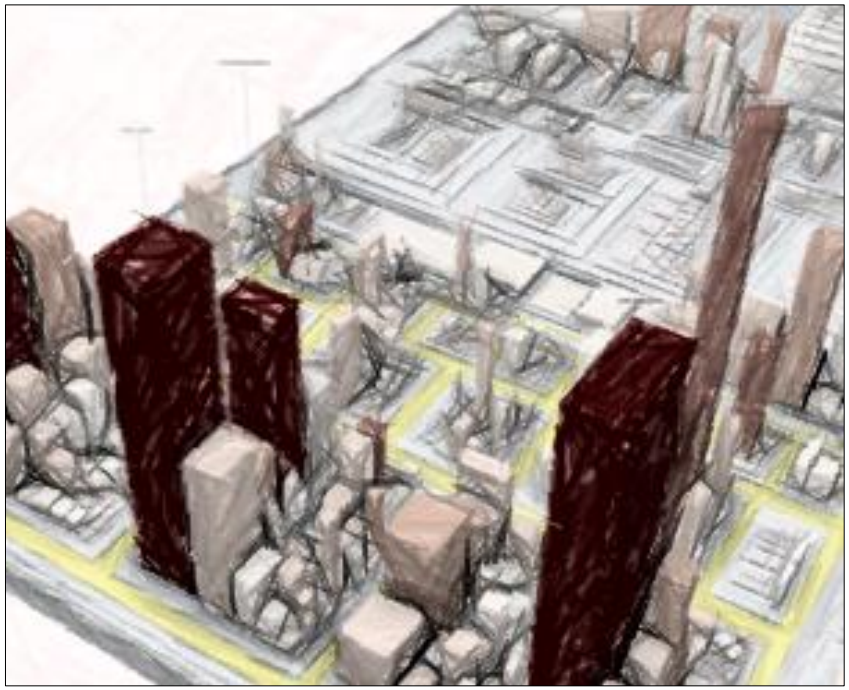
Summary

Software analytics has a major impact on the effectiveness in software development and software engineering. It provides a **holistic view**, taken from the development perspective, quality perspective, management perspective, and methodology perspective.

Software analytics helps to transform **heuristics-driven software engineering into a data-driven discipline** ("analytics nature"), while being adaptive to the specific form and mix of existing software engineering methodologies, techniques, and tools ("non-normative nature").

It offers a kind of **software management cockpit** that helps to reveal, correlate, explore, analyze, and monitor key information sources of software development projects.

Software Development Effectiveness



Software Analytics Workflow
Elements of Software Analytics
Impacts of Software Analytics

Terminology

Software Analytics Elements

Software analytics elements denote combining, integrating, and adapting elements of automated data extraction, big data analytics, visual analytics, prescriptive analytics, and prescriptive analytics with respect to the software engineering and software development.

Holistic View

With *holistic view*, software analytics denotes the support of different perspectives involved in software engineering such as the software development perspective, the software quality perspective, the management perspective, and the methodology perspective.

Further Reading

Monitoring Code Quality and Development Activity by Software Maps

Johannes Bohnet, Jürgen Döllner

Hasso-Plattner-Institute for IT Systems Engineering @ University of Potsdam
Germany

{bohnet, doellner}@hpi.uni-potsdam.de

ABSTRACT

1. INTRODUCTION

Categories and Subject Descriptors

General Terms

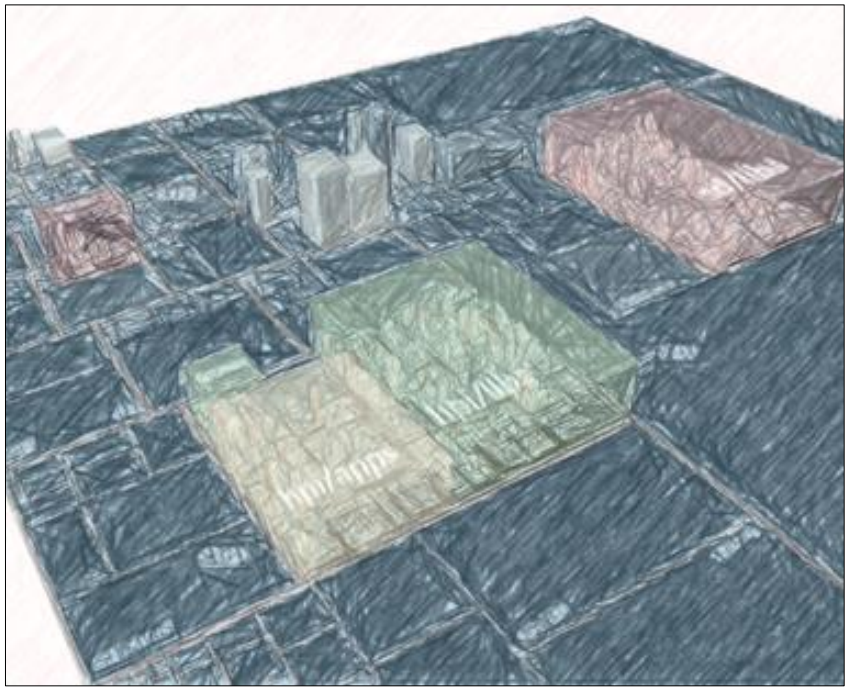
Keywords

Johannes Bohnet, Jürgen Döllner: “Monitoring Code Quality and Development Activity by Software Maps”. IEEE ACM ICSE Workshop on Managing Technical Debt, 9-16, 2011.

Remarks

- Like in a growing number of engineering-based disciplines, the data-driven approaches take the lead. In software engineering, somehow ironically, we do have excellent systems and tools, recording data about almost all aspects of the software development, but all the data is hardly used by means of analytics.
- Software analytics is a game changer. It somehow demystifies software engineering as an engineering discipline rather than an art. Most important, it helps to handle the inherent risks in long-term developments.

Outlook on Software Analytics



Enhancing Visual Analytics
Enhancing Predictive & Prescriptive Analytics
Enhancing the Data Scope

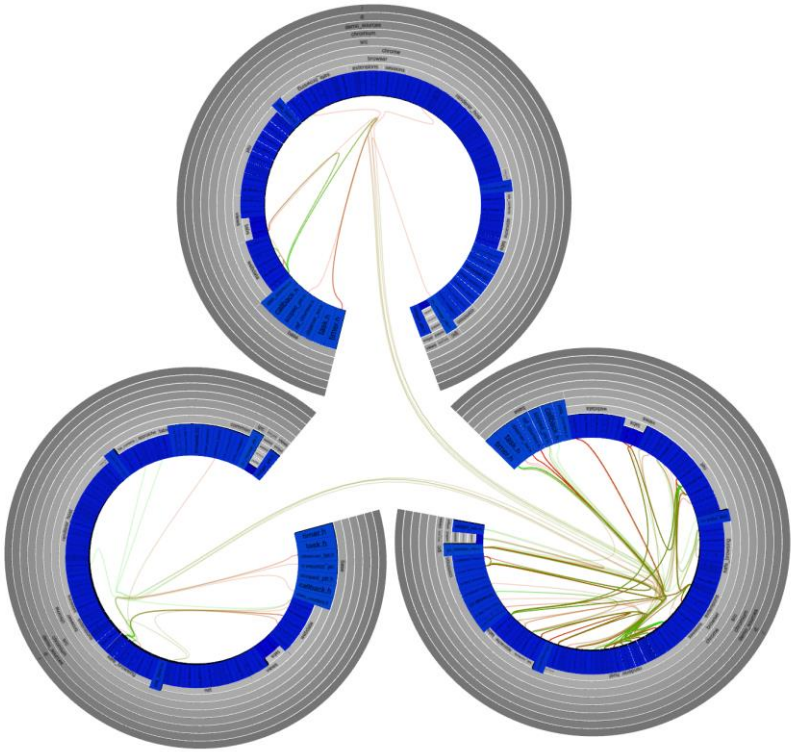
Enhancing Visual Analytics

- **Software visualization and information visualization** are the major scientific disciplines contributing to enhanced visual analytics approaches.
- Goal is to be specific to the characteristics and demands of software engineering.
- Key requirements:
  - large hierarchical data (e.g., for legacy or large-scale systems)
  - large temporal data (e.g., software versions recording the development of more than one decade)
  - effective human perception and recognition

Enhancing Visual Analytics

Example: Multiple Hierarchical Circular Bundle Views

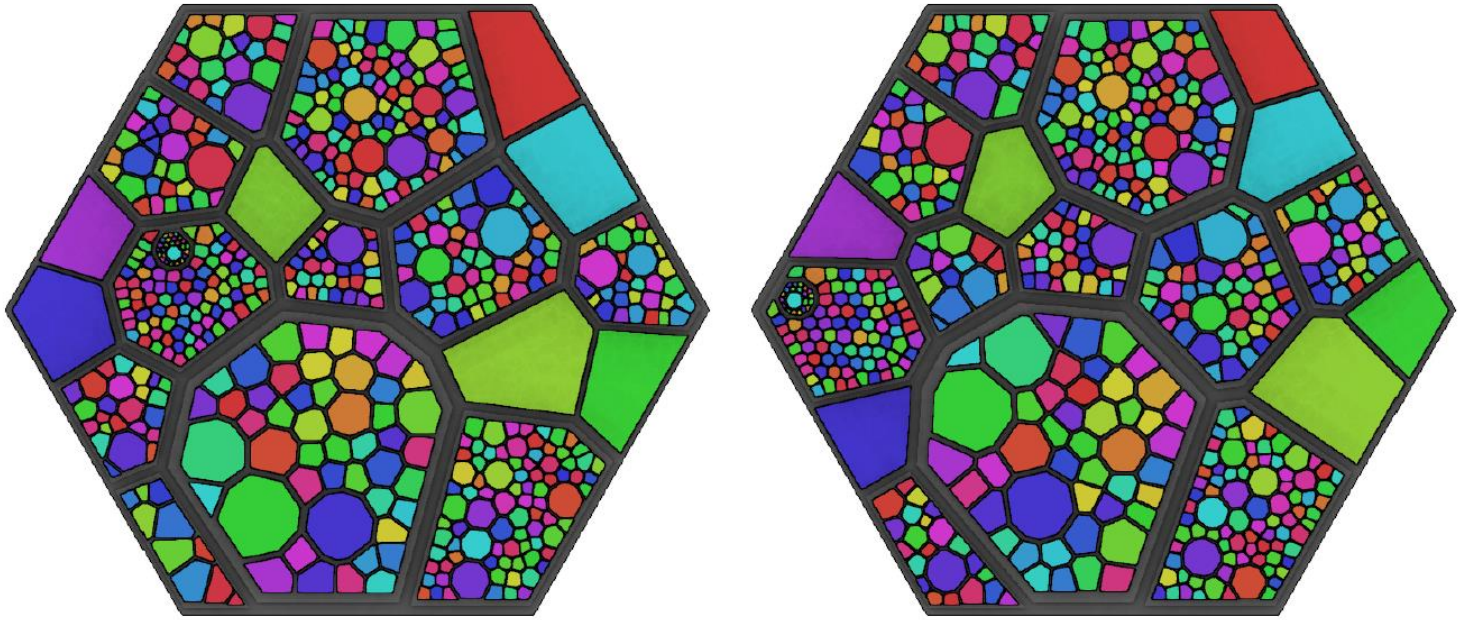
- Visualizing hierarchical system components of multiple systems (or subsystems) together with relations defined among them by means of multiple hierarchical circular bundle views
- Potential applications include:
  - visual redesign of hierarchies of multiple (sub)systems
  - preparation of independent systems that are to merged, subsumed, split, or refactored
  - cross-system comparison of relations among system components, responsibilities, and dependencies



Enhancing Visual Analytics

Example: Voronoi Treemaps for Visualizing Changing Hierarchy Data

- Visualizing hierarchical system components by means of Voronoi treemaps.

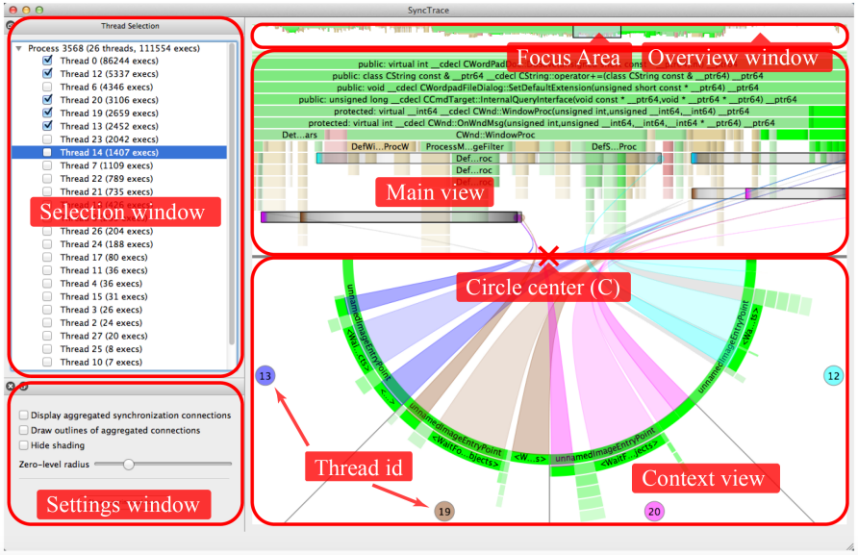


– Sebastian Hahn, Jonas Trümper, Dominik Moritz, Jürgen Döllner: "Visualization of Varying Hierarchies by Stable Layout of Voronoi Treemaps". In Proceedings of the 5th International Conference on Information Visualization Theory and Applications (IVAPP 2014), pages 50-58, 2014. SCITEPRESS Science and Technology Publications.

Enhancing Visual Analytics

Example: Visualization of Multithreaded Interplays

- Visual analytics to support comprehension of thread execution traces and their interplay (e.g., finding usages of blocking operations, such as synchronization and I/O operations, assessing temporal order of such operations).

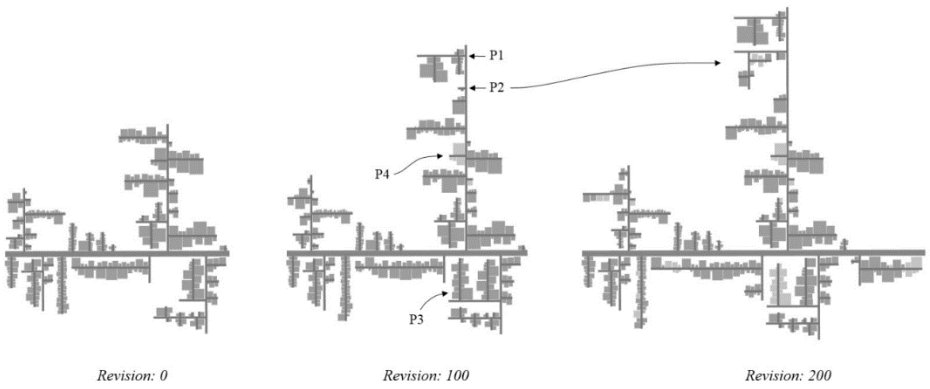


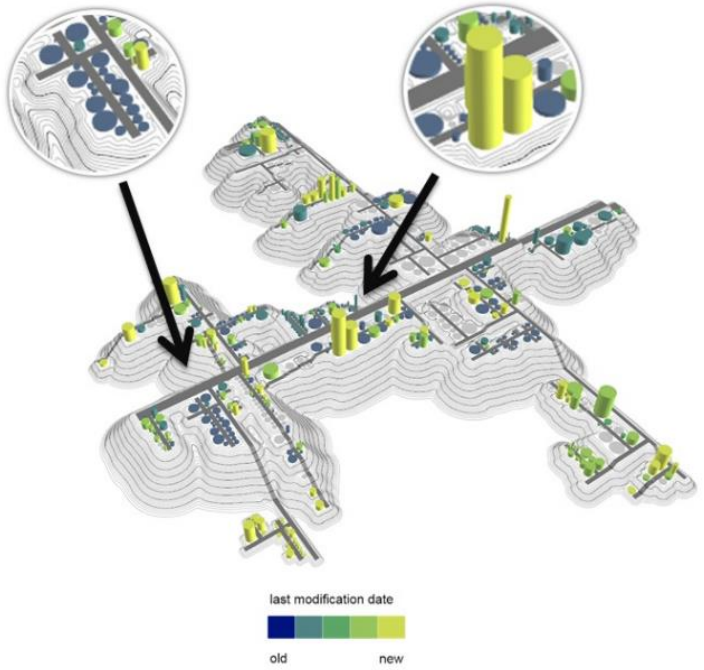
– Benjamin Karran, Jonas Trümper, Jürgen Döllner: "SyncTrace: Visual Thread-Interplay Analysis". Proceedings of the 1st Working Conference on Software Visualization (VISSOFT), pages 10, 9 2013. IEEE Computer Society.

Enhancing Visual Analytics

Example: Representing Development History with Software Cities

- A metaphor-based approach (Software City) that allows for a layout-stable visualization of revisions. A height field is used to depict an orthogonal attribute dimension (creation time).





– F. Steinbrückner: "Consistent Software Cities : supporting comprehension of evolving software systems". PhD Thesis, Brandenburgische Technische Universität Cottbus, Juni 2013

– http://www.tu-cottbus.de/fakultaet1/de/software-systemtechnik/lehrstuhl/personen/frank-steinbrueckner/publikationen-steinbrueckner.html

Enhancing Predictive & Prescriptive Analytics

- A key area of innovation is concerned with advanced methods for **prescriptive analytics specific to software engineering tasks**.

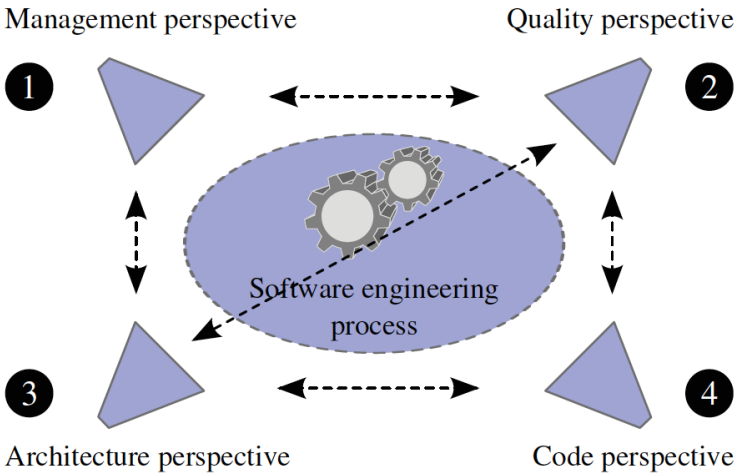
- One category of prescriptive software analytics addresses the specific **needs of the management level** to support decision making at a high level of abstraction.

- Another independent category of prescriptive software analytics addresses the **needs of software development teams** to get assistance in focusing team activity and development hot spots.

Enhancing Predictive & Prescriptive Analytics

Example: Software Analytics as Recommendation Systems

- Extending the collection of software map themes towards the specific needs regarding management, quality, architecture, and code perspectives.



NEEDS OVERVIEW PER PERSPECTIVE.

Perspective	Id	Need description
Management	M1	Condensed, high-level information
	M2	On-demand context information
	M3	Up-to-date overview fact-sheets
	M4	Discuss with other perspectives
	M5	Representation understood by all parties
Quality	Q1	Info. on where to invest testing effort
	Q2	Up-to-date system structure
	Q3	Up-to-date internal module structure
	Q4	Info. on maintainability and change impact
Architecture	A1	Up-to-date system structure
	A2	Up-to-date information on development activity
	A3	Multi-level representation of the system or facts
	A4	Restructuring possibilities
	A5	Change impact assessments
	A6	Change effort assessments

– Jonas Trümper, Jürgen Döllner: "Extending Recommendation Systems with Software Maps". In Proceedings of the 3rd International ICSE Workshop on Recommendation Systems for Software Engineering (RSSE), pages 92-96, 2012. IEEE Computer Society.

Enhancing Predictive & Prescriptive Analytics

Example: Active Management of KPI Objectives

- Defining and monitoring objectives related to software metrics targets or, generally, targets related to key performance indicators using software maps.
- For a given component or subsystem, a target range or corridor can be specified and monitored.
- Predictive analytics helps to estimate whether the actual development activities lead to this goal based on trend analysis. Determine and propose appropriate measure to be taken and adopted by different stakeholders to keep development on track.

– Joint research project with Software Diagnostics Technology

Enhancing Predictive & Prescriptive Analytics

Example: Benchmarking

- Benchmarking across projects having a similar nature
- "How good is my application regarding maintenance compared to other applications?"
- Relate, compare, and learn from the experience and dynamics of other software development projects

– Joint research project with Software Diagnostics Technology

Enhancing the Data Scope

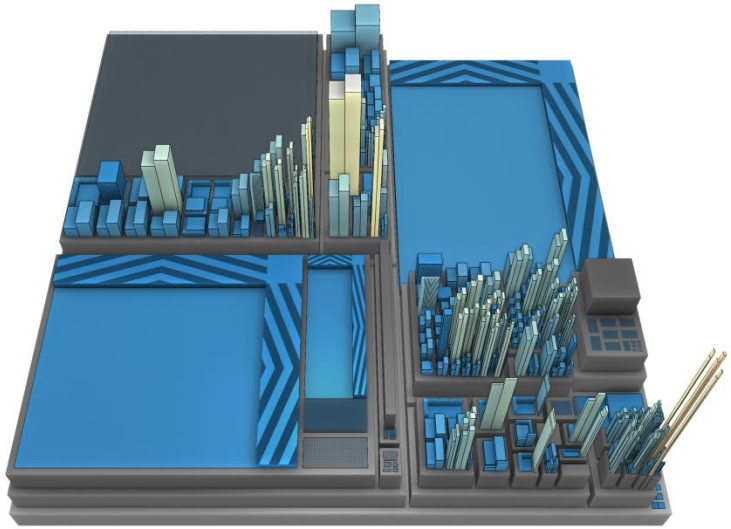
- The data scope considered and used by software analytics can be enhanced towards **financial data and business data**.
- This way, complementary aspects of software projects and their embedding in a company, can be considered by software analytics tasks.

- Software analytics extended by financial and business data can help to support decision making far **beyond software engineering** such as for
  - due-diligence processes,
  - financial statements and technical debts,
  - cost and budget distribution,
  - accrued liabilities inherent to software systems and their components
  - …

Enhancing the Data Scope

Example: Technical Debts & Budget Software Maps

- Extending the scope to financial aspects of software development
- Taking into account developer salary, developer time, and accrued technical debts
- Providing new related means regarding a company's accounting treatment



– Joint research project with Software Diagnostics Technology

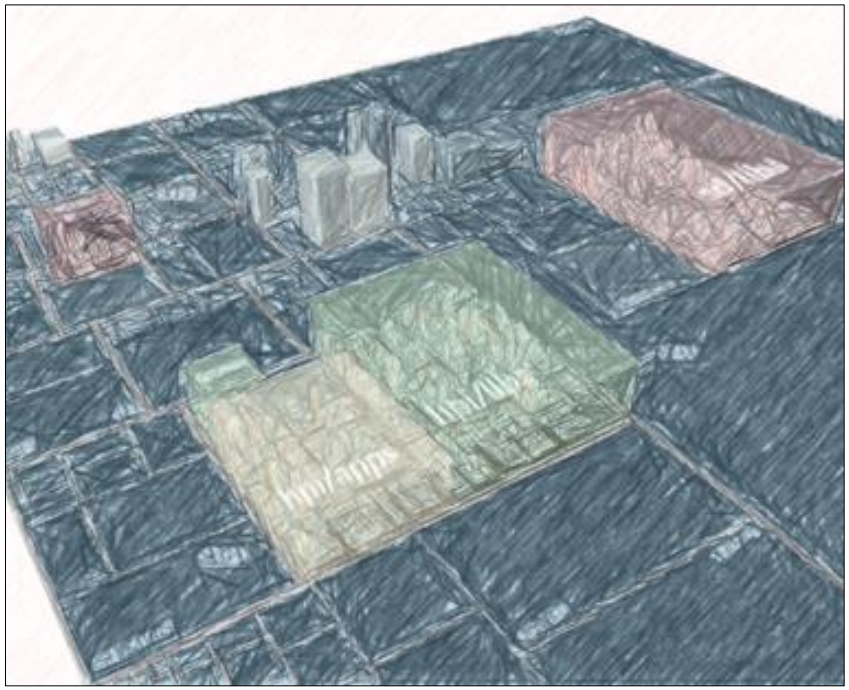
Summary

Software analytics will take a rapid innovative progress in the next years. As a key and "game-changing" technology, it has to broaden its capabilities and functionality in various directions.

Three principal directions can be identified today:

- Enhancing the visual analytics tools by new visualization approaches, in particular, specific to large-scale hierarchical and temporal data.
- Enhancing the predictive and prescriptive software analytics tools to effectively support decision making and actionable insights by means of measurable, specified, monitored, and guided activities for the different stakeholders.
- Enhancing the data scope for software analytics by including financial data and business data, bringing closely together the fields of software analytics and business intelligence.

Outlook on Software Analytics



Enhancing Visual Analytics
Enhancing Predictive & Prescriptive Analytics
Enhancing the Data Scope

Remarks

- We hope you got an overview of what software analytics is about, its strengths, its principles, its techniques, and tools. We hope you got some ideas why and how to apply software analytics within your context.
- All stakeholders - from managers to testers - can better communicate what they consider to be important - a big step forward to more effectiveness and transparency in data-driven software engineering.
- You can expect a boost of productivity; based on our own first empirical observations when applying software analytics in large commercial contexts it can have an impact of about 30%...
- First professional, all-in-one solutions are to come such as from HPI's research partner Software Diagnostics Technology - in a professional context, we recommend such all-in-one solutions instead of a coarse collection of separate tools.