RZ 3581 (# 99591) 02/07/2005
Computer Science 11 pages

Research Report

System Evolution Tracking through Execution Trace Analysis

M. Fischer,‡ J. Oberleitner,‡ H. Gall,† T. Gschwind*

‡Technical University of Vienna
Distributed Systems Group
Information Systems Institute
Vienna, Austria

†Dept. of Informatics
University of Zurich
Switzerland

*IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

System Evolution Tracking through Execution Trace Analysis [*]

M. Fischer and J. Oberleitner
Technical University of Vienna
Distributed Systems Group
Information Systems Institute
Vienna, Austria
{robinson,joe}@infosys.tuwien.ac.at

Harald Gall
University of Zurich
Department of Informatics
s.e.a.l. – software
evolution & architecture lab
{gall}@ifi.unizh.ch

T. Gschwind
IBM Research
Zurich Research Lab
Switzerland
{thomasg}@ieee.org

Abstract

Software evolution analysis is concerned with analysis of artifacts produced during a software systems life-cycle. Execution traces produced from instrumented code reflect a system's actual implementation. This information can be used to recover interaction patterns between different entities such as methods, files, or modules. Some solutions for detection of patterns and their visualization exist, but are limited to small amounts of data and are incapable of comparing data from different versions of a large software system. In this paper, we propose a methodology to analyze and compare the execution traces of different versions of a software system to provide insights into its evolution. We recover high-level module views that facilitate the comprehension of each module's evolution. Our methodology allows us to track the evolution of particular modules and present the findings in three different kinds of visualizations. Based on these graphical representations, the evolution of the concerned modules can be tracked and comprehended much more effectively. Our EvoTrace approach uses standard database technology and instrumentation facilities of development tools, so exchanging data with other analyses is facilitated. Further, we show the applicability of our approach using the Mozilla open source system consisting of about 2 million lines of code.

1 Introduction

Dynamic analyses based on execution traces are used in software testing, software performance analysis, distributed and parallel systems evaluation, and to some extent also in program comprehension and reengineering.

Dynamic information is typically expressed by execution traces that are recorded when instrumenting the source code and storing all events that occur in a given scenario. For that, typical scenarios for an application can be chosen and instrumented. This allows an engineer to focus on important aspects to be validated or comprehended. For a browser software such as Opera or Mozilla, this would be the connection to an http server and the loading of a page.

One challenge with dynamic data is its size: simple scenarios can result in very large execution traces. Because of that, researchers have investigated compression techniques to cope with the size challenge, e.g. [10].

In our previous work, we have investigated scenarios for the Mozilla open source software system to investigate the evolution of its features [6]. We instrumented the code and analyzed the execution traces to find out the relevant functions implementing a particular feature. This was combined with change and bug information to discover all kinds of change dependencies between features. This information was reflected onto the source base structure and visualized for the analyzing engineer.

In [13] we presented an architecture analysis approach that utilizes certain evolution data sources and provides an integrated view. The analysis applies fact extraction and generates specific directed attributed graphs that represent information on accesses, includes, inherits, invokes, and coupling between certain architectural elements. For that purpose, we only considered static information but in many situations it appeared that dynamic information could be supplemented rather beneficially. Especially, since our goal is to point software engineers to locations in a software system that may be critical for maintenance and evolution activities.

Execution traces have been used in program comprehension to facilitate understanding about interactions between building blocks of a software system. Further, they have

[*]The work described in this paper was supported in part by the Austrian Ministry for Infrastructure, Innovation and Technology (BMVIT), the Austrian Industrial Research Promotion Fund (FFF), the European Commission in terms of the EUREKA 2023/ITEA project FAMILIES (http://www.infosys.tuwien.ac.at/Cafe/) and the European Software Foundation under grant number 344.

been used to dynamically discover likely program invariants that must be preserved when modifying and evolving source code [5].

So far their full potential for coarse and fine grained analysis of program evolution has not been exploited. Research work in this area mainly focused on the visualization of execution traces (information mural by Collberg), detection of patterns in the resulting traces for data reduction to overcome the problem of information explosion and their representation as graph.

While in reverse architecting dynamic information such as call graph information is used to get a complete *static* picture of the actual implementation of a software system, execution traces have not been exploited for detailed retrospective software evolution analysis.

Most of the information recorded in execution traces is captured as well by profiling information. Thus, profiling information can be used to generate a call graph or to gain information about the invocation frequency of each method. But patterns of invocation are not recorded, i.e., it is not possible to deduce how frequently a method C was invoked as $A \rightarrow C$ or $B \rightarrow C$.

As a further shortcoming, we identified the impossibility to determine how these invocations are distributed over the execution time, i.e., during which program execution phase the invocation patterns emerge. Reason for the limited data recording capabilities of "traditional" profiling is the information explosion during program execution and the impact on execution time if detailed data are gathered. But for a significant number of software systems and their use cases, this limitations can be neglected if data can be collected using specific test environments.

In contrast to a call graph analysis of a single release of a software system, in retrospective software evolution analysis we are interested in the deltas applied to the software system which describe the changes from one release to another.

We are interested in the occurrence of specific invocation patterns between modules or files and their change when different releases of a software system are compared.

We have seen the need for integrating dynamic information in our previous works and propose an *execution trace analysis approach* to support an engineer in tracking a system's evolution.

The contribution of this paper is *a methodology to exploit program execution traces for retrospective software evolution analysis and provide different visualizations* to support the reasoning about program evolution. For that, we use Mozilla, a multi-million line open source software system, to show the applicability of our *EvoTrace* approach.

The paper is organized as follows: Our *EvoTrace* approach is described in Section 2. In Section 3 we apply it on a large Open Source Software system and present interesting findings. Related work with respect to retrospective software evolution analysis is discussed in Section 4, and finally in Section 5 we draw our conclusions and indicate future work.

2 Methodology

In this section we describe the methodology we use in the *EvoTrace* approach to obtain evolutionary information from execution trace data. *EvoTrace* currently comprises

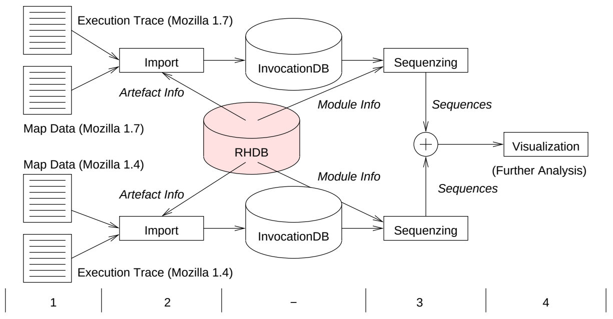


Figure 1. Import and analysis process

three steps as depicted in Figure 1: (1) instrumentation, trace- and map-data generation; (2) import from execution traces; (3) sequencing invocations between modules; and (4) visualization. While the first step is development platform dependent, the subsequent steps of *EvoTrace* use Perl, Java and MySQLwhich are available for a number of OS platforms. Next, we describe the data representation and import process based on our implementation of *EvoTrace*. Though, the process is tailored for our Linuxdevelopment environment, other OSs can be used as well, provided that the required information is available.

The central element is a release history database (RHDB) [7] that contains history information from versioning systems such as CVS and bug tracking data together with links to architectural information.

2.1 Instrumentation, trace- and map-data generation

As noted in [11] there exist three methods to generate traces of method calls: (1) insertion of probes such as prints; (2) modification of the runtime environment such as Java; and (3) debugging to monitor program execution. In the Linux environment the first method is supported by the compiler so only two functions–one for entering and one for exiting–must be implemented. Appropriate calls to these functions are then generated by the compiler. After compiling and linking the application can be tested.

For Mozilla we used a typical scenario in which a web page from our web server was loaded. To avoid user interaction with the application, the application is terminated via an external `QUIT` signal when no more additional events are recorded.

Before the trace information can be used in the further analysis process, the recorded addresses must be mapped onto method- and file-names. This is done with map data generated from the two GNU tools `ldd`and `nm`The first tool, `ldd`, generates a mapping of base addresses for the dynamic linked libraries. These base addresses are required to determine the library for which a call was recorded. The second tool, `nm`, lists symbols from object files with source file name and line number information. Both outputs are written to a map file so the mapping information together with the trace data can be used in the following import process.

2.2 Importing execution traces

Result of the import from the execution traces and map file data are two separate database tables containing the respective traces of each Mozilla release with linkage to existing artifacts in the Release History DB (RHDB). The import works via a Perl script and is divided into two steps: (1) read map file information and try to find corresponding artifacts in the RHDB; and (2) read the execution trace information and add one record in the database for each event in the trace file. After some experiments with the trace data we decided to use the format depicted in Table 1 for the database table which we called *invocation sequence* (`invosequ`). The field sizes are specified in bytes. The trace data generated during execution of the testee, are stored in the four fields: *callee*, *caller*, *type*, and *threadid*. The remaining fields are evaluated during the import from the Perl script:

id: Is a unique identifier assigned during data import to facilitate further analysis.

callee: The code address of the method invoked during program execution from *caller*.

caller: This address determines the exit point of an invocation in the execution trace. While the callee address has a direct mapping to linker addresses, the caller address maps to the code segment between methods and thus is not directly usable. Instead, an application of the caller address lies in the search of corresponding *enter-exit* pairs. These pairs can unambiguously identified within a thread context via the 3-tuple *callee*-, *caller*-address and invocation *level*. Finally, the thread context is used to distinguish between the different execution paths.

type: Each event in the database is marked either with 'e' for enter and 'x' for exit of a method.

threadid: Every event requires information about the corresponding thread context; otherwise traces are intermixed.

Table 1. Record format for trace data

Name	Size	Description
id	4	Unique ID for this event
callee	4	Text segment address of called method
caller	4	Call issuing address
type	1	Method call c or return r
threadid	1	ID of thread context
level	1	Invocation or recursion level
cvsitemid	3	ID of artifact in RHDB

level: The recursion level information is simply derived from the *type*-field by counting enters and exits on a per-thread basis. This information is added to simplify database queries.

cvsitemid: From the import of release history data into our RHDB, a mapping from source files to unique IDs already exists. With the symbol information from object files we are able to map the *callee*-address to the corresponding entry in the RHDB. This information is required to assign the file information to modules.

After importing and linking relevant information, the invocation database is ready to serve queries. In our case study (see Section 3) we give some examples for a quantitative evolution. Next, we describe an analysis algorithm for the detection of interactions between modules based on the invocation sequence data.

2.3 Sequencing

We focus on invocations between different modules. This reduces the amount of information to be displayed and characterizes the communication between modules. Thus we are interested in invocation sequences S_1, S_2 between modules M_a, M_b and their methods a, b, c such as $S_1 = M_a.a(M_b.a(); M_b.b())$ or $S_2 = M_a.a(M_b.a(M_b.b()))$. S_1 exhibits two module switches and S_2 exhibits only one module switch. These invocations are derived from the invocation data stored in the `invosequ` table using the fields `type`, `cvsitemid` and `level`. Since data are not represented as graph in the database, we need to traverse the complete content of the invocation sequence table which is performed by a small Java program. Figure 2 shows the (simplified) Java code which is used to detect the invocations between modules. To reveal the transitions between the different modules a data structure holds information about the invoked modules. For each change of invocation level, the event pairs *(new event in trace, old event on stack)* are compared and a change is checked and recorded via the function `save_diff_module(o,n)`. This function compares the module IDs and counts the transitions of the program flow. Transitions within a method, i.e., on the same invocation level, are recorded with the code in the else-branch. Here, the topmost element of the stack is replaced with the new

```
Event[] events = new Event[MAX_STACK];
events[0] = trace_data();
int cntevent = 1;
while (more_trace_data()) {
  Event n = new Event(trace_data());
  Event o = events[cntevent-1];
  if (n.level > o.level) {
    save_diff_module(o,n);
    events[cntevent++] = n; // save new event
  }
  else if (n.level < o.level) {
    save_diff_module(o,n);
    events[--cntevent-1] = n; // replace old event
  }
  else {
    events[cntevent-1] = n; // replace old event
    o = events[cntevent-2]; // get new old event
    save_diff_modules(o,n);
  }
}
```

Figure 2. Java code for transition detection

event. Then the two elements on the stack are checked for different module IDs. While the if-statements check for invokes and returns such as $M_a.a(...M_b.a();...)$ the default branch detects a series of invocations such as $M_b.b()$ and $M_b.c()$ in $M_a.a(...M_b.a();...M_b.b();...M_b.c();...)$. Every detected transition—i.e., their respective module ID—is written to a separate database table. Next, input data for visualizations are generated from this information.

2.4 Visualization

Currently, the visualization of results is used as substitute for the comparator function until an efficient pattern detection algorithm such as [12] is implemented. As substitution, we combine data from the two versions of execution traces into diagrams so they can be compared visually. One major problem for visualization are the deficiencies of the often used Gantt charts for the presentation of $2 \cdot 10^6$ transitions between modules within the usual viewing range. Consequently, we had to reduce the amount of information. A frequently applied solution is the application of sub-sampling. Since no constraints on the time-slots were given, we decided to use twenty time-slots since it was most appropriate for use in the generated diagrams.

Based on this sub-sampling interval, we counted the module transitions detected in the previous step and generated data sets for three different diagram types: (a) A Gantt chart provides a good view on different phases of the program execution. Different phases such as system initialization or user interface related activities can be distinguished; (b) the "matrix" view emphasizes the quantitative aspect of changes in invocations between modules. The two communication directions between entities are depicted separately; and (c) for a more detailed view on the interaction between modules we use Kiviat diagrams. In this view, the communication between each module is shown on separate axes in the diagram.

The diagrams are generated automatically via a Perl script from the given data sets. Results are depicted in Figures 3, 4, and 5, respectively.

2.5 Optimizations

Next, we discuss some optimizations which we identified during the development of this approach and in relevant literature.

As discussed by Hamou-Lhadj and Lethbridge [11], a limiting factor is the problem of size explosion. Size reduction through pattern matching seems to be the most appropriate solution to this problem.

Deactivation of instrumentation—as sometimes proposed—for certain files to reduce the amount of generated traces would require detailed knowledge about the software system to inspect because otherwise important invocation transitions could be lost. Another drawback of the deactivation solution is the required effort to manually enable or disable instrumentation on a per method basis. To reduce the amount of records the currently separate *enter* and *exit* records can be merged since most of the information is redundant. Aside from the size reduction extra lookups to find a corresponding invocation pair are avoided.

Support for multi-threaded test applications: Instead of using a specific database column to hold thread IDs, writing different thread traces to different database tables could considerably reduce memory consumption.

Standard database technologies support fast access to events of selected modules: If the analysis environment provides sufficient computing power and memory ($\geqq$ 1GB, $\geqq$ 3GHz for a Pentium 4), database tables can bee kept in memory. Thus the execution time for entries in ad hoc queries ranges from fractions of a second to less than one minute depending on whether an index can be used to resolve the query.

Support for detection of sequences and patterns: One field of future work is the detection of invocation patterns. Detected patterns are a prerequisite for the implementation of a fast comparator function of the version related trace data.

Handling multiple versions of execution trace data: In *EvoTrace* we use different database tables to handle the execution traces originating from different version of the test program. Including the version information into the tables would create a large amount of redundant information.

Linkage with existing release history information: This linkage is required to facilitate the architectural evolution analysis process. In recent work [13] we have merged them to information spaces. The traces contain a wealth of information which can be made directly available via database queries for interactive visualization or retrospective evolution analysis.

3 Case study

As in our previous research [6, 7] we continue to use the Mozilla Internet application suite as a representative and challenging case study. Major reasons for that are the already existing RHDB with architectural and evolutionary information and our experiences with Mozilla. Furthermore, results of this work have been integrated into the RHDB to further augment the exploration of the software evolution information space.

The current snapshot of the case study is based on version 1.7 (released 2004-06-18) and version 1.4 (released 2003-07-01) of the Mozilla code base. While version 1.7 is the latest release we have release history data about, the reason for using 1.4 is a problem with the development environment. Releases prior to 1.4 require an outdated version of the GNU compiler collection (GCC), thus earlier releases are not compilable with our current development environment.

Table 2. Selected Mozilla modules and their source code directories

Module	Source Directories
MathML	layout/mathml
New Layout Engine	layout/base, layout/build, layout/html
XPToolkit	content/xul, layout/xul
Document Object Model (DOM)	content/base, content/events, content/html/content, content/html/document, dom
HTML Style System	content/html/style, content/shared
XML	content/xml, expat, extensions/xmlextras
XPCOM	xpcom

For our work on the integration of the architectural and evolutionary information [13] we used a subset of the available Mozilla modules which are related to web content representation and layout. Table 2 lists the selected modules which we will use in this case study again.

3.1 Data collection

Before data collection can start some preparation work of the testee has to been done. Both source code versions of Mozilla are instrumented via the `-finstrument-functions`compiler option provided by GNU compiler collection. This option generates instrumentation code for entry to and exit of functions. Just after function entry and just before function exit, the following profiling functions will be called with the address of the current function and its call site.

```
void __attribute__
        ((__no_instrument_function__))
    __cyg_profile_func_enter
        (void *callee, void *caller)
```

With a similar C function returns from methods can be recorded. To avoid conflicts with the instrumentation function the `__no_instrument_function__` attribute has to be applied. This avoids its recursive invocation. Another source of conflict is Mozilla's thread library `nsprpub`. We require some functions of this library to determine the thread context under which the instrumentation function is executed. For this library we disable instrumentation completely. After this instrumentation work, both programs can be compiled using the same compiler and configuration options.

To avoid interference through user interactions, we implemented a shell script which automatically starts the application with the specified test-parameters and terminates the application after a predefined timeout period. As test-scenario we use a copy of a page of the W3C's *MathML test suite*which we placed on our web server[1] Difference in the resulting execution traces due to network indeterministic can be neglected since the selected modules are not related to network communication. As timeout when the application shall receive the `QUIT` signal we determined one second to be sufficient.

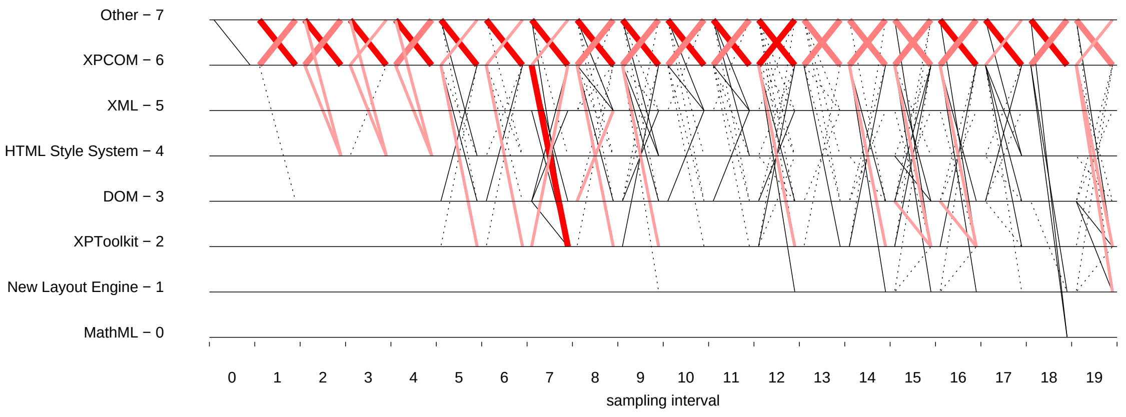
Another source for differences in the resulting traces are changes to Mozilla internal configuration files. To minimize the impact of this source of interference, we used *three test-runs* in a row whereas only the results of the last one are used. To avoid conflicting interactions with the window manager of our test-system, we used a separate X-Window server without any window manager functionality. During test-runs the application window is redirected to this separate server while trace data are stored on the local disk drive.

Currently, we used the C `printf`-function to record each *enter* or *exit* event. The snippet below depicts the data format produced by the instrumentation function:

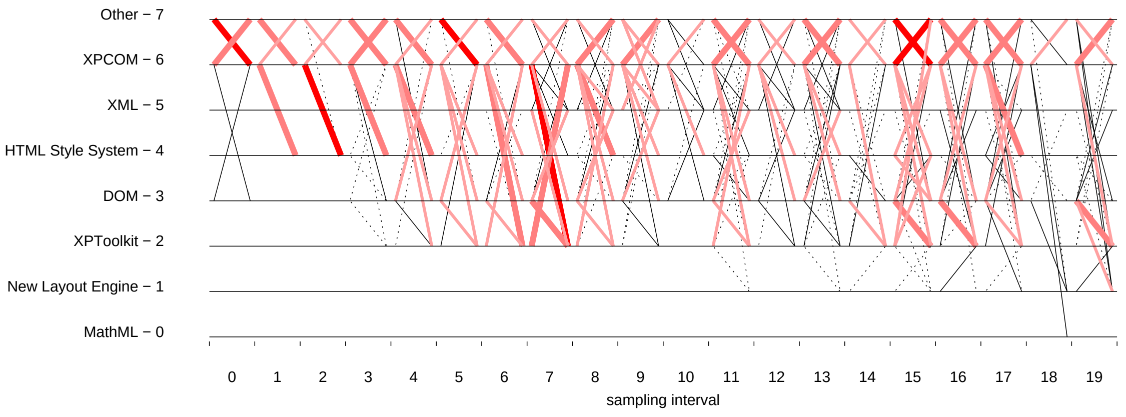
```
e0x8cede20m0x8cedf09t0x8f26548
```

Four types of information are recorded: (1) the event-type (*enter* or *exit*); (2) the callee address; (3) the caller address (starting at the letter 'm'); and (4) the thread context (starting at 't').

[1]http://www.infosys.tuwien.ac.at/staff/mf/testpages/iwpc05/math3.xml



(a) Mozilla 1.4



(b) Mozilla 1.7

Figure 3. Execution trace for Mozilla modules as Gantt diagram

3.2 Post-processing and quantitative results

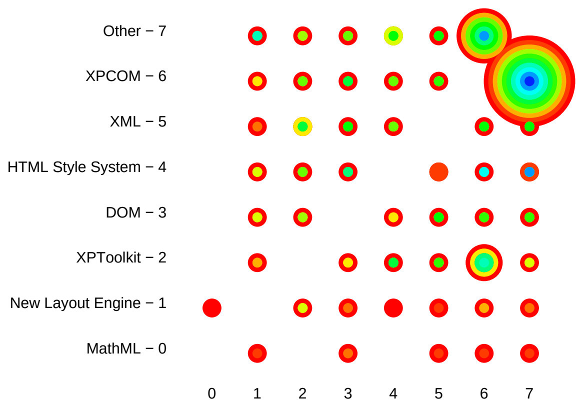
After the import of the raw data into the database via a Perl script, we can obtain first quantitative results with simple SQL queries. The results are listed in Table 3 for both Mozilla versions.

The binary file of version 1.7 is large compared to version 1.4 but the code seems to be leaner and produces less execution trace events. Even though listed as exact numbers, the number of events vary slightly between test-runs since network communication or the OS timing is not deterministic.

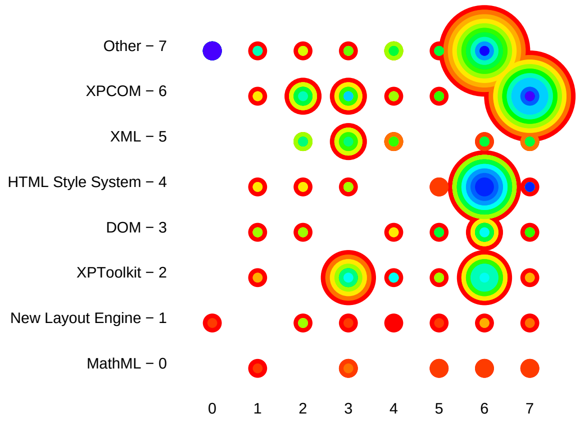
The number of different start addresses of invocations found in the execution traces is given by *callee addresses*. This differs from *number of methods*—number of different methods signatures found in object files—which is based on the symbol information delivered by `nm`. Differences originate from C++ language constructs and internal management tasks of the runtime library. The *caller addresses* lists the number of different addresses from where methods have been invoked. To assign traces to the correct thread context we record the thread ID at each event. Consequently, the *number of threads* gives the total number of different IDs

Table 3. Basic results from trace data

Mozilla	1.4	1.7	Δ%
Binary size	82,109,017	101,012,842	+23
Number of events	23,878,728	18,822,452	-21
Callee addresses	12,077	11,644	-3.6
Caller addresses	41,962	37,011	-12
Number of threads	4	5	+25
Deepest call nesting	153	164	+7.2
Number of methods	11,940	11,563	-3.2
Number of files	868	850	-2.1
Files from modules	403	396	-1.7



(a) Mozilla 1.4



(b) Mozilla 1.7

Figure 4. Interval-based tree-ring invocations for Mozilla modules as matrix view

found.

One aspect not covered by "traditional" profiling is the nesting level. With *deepest nesting* we give the deepest level of invocations found in the execution traces. During the import phase the *callee* address information is combined with the symbol information from object files. Here, the *number of files* represents the number of successful maps to source files. In a post-processing phase, we then identified those files which belong to the modules we are interested in (*files from modules*). This speeds up later data analysis.

3.3 Visualization

After the generation, filtering and first quantitative evaluation of the test-data, we visualized the results for evolution tracking. As described in the previous section, we divided the execution trace into twenty different intervals for subsampling. We have decided to use twenty intervals because the intervals are sufficiently small to distinguish different types of interaction but is high enough to create "readable" visualizations. This interval size is more relevant for the first diagram type we present here than for the other two.

3.3.1 Gantt diagram

One well known form for visualization of execution traces are Gantt diagrams which are well suited to study interactions on a very fine grained level. Since our *EvoTrace* approach is designed to reveal coarse changes in system interaction, we use a "reduced" form of the Gantt diagram type where the invocations are sketched. In Figure 3 this modified diagram type is depicted with the filtered invocation sequences of Mozilla 1.4 (a) and Mozilla 1.7 (b) respectively.

In both diagrams the invocation frequencies between modules are divided into six classes: $> 50\%$, $> 25\%$, $> 10\%$, $> 5\%$, $> 2.5\%$ and $\leqq 2.5\%$ whereas invocations of the last class are not shown. Invocations are depicted as lines with different shapes representing their frequency between modules.

When visually comparing diagrams (a) and (b) the differences in invocation intensity between the modules `Other` and `XPCOM` are significant. This was surprising, since we did not expect such extreme changes. Interesting to see are also the mutual invocations between `Other` and `XPCOM`. But this is an expected result since `Other` contains all other modules we did not explicitly identify.

Roughly, four phases can be distinguished: (α) prelude (time-slots 0-2); (β) user interface–`XPToolkit` is the cross-platform user interface–related activities (slots 3-9); (γ) an intermediary phase (slot 10); and (δ) content related activities including `MathML` (slots 11-19). The main differences are that phase α begins in version 1.7 two time-slots earlier and that the intermediary phase γ can be clearly identified. Remarkable is also the strong communication path in slot 7 from module `XPCOM` to `XPToolkit` which appears in both versions.

3.3.2 Matrix view

To overcome the problem of clutter in our Gantt diagram, we developed a specific matrix view, which supports the visualization of invocations as cross product between modules. Callers are placed on the horizontal axis and callees are placed on the vertical axis. For instance, to find the invocations from `XPCOM` to `HTML Style System` you need to go to column 6 and move up till row 4. Dur-

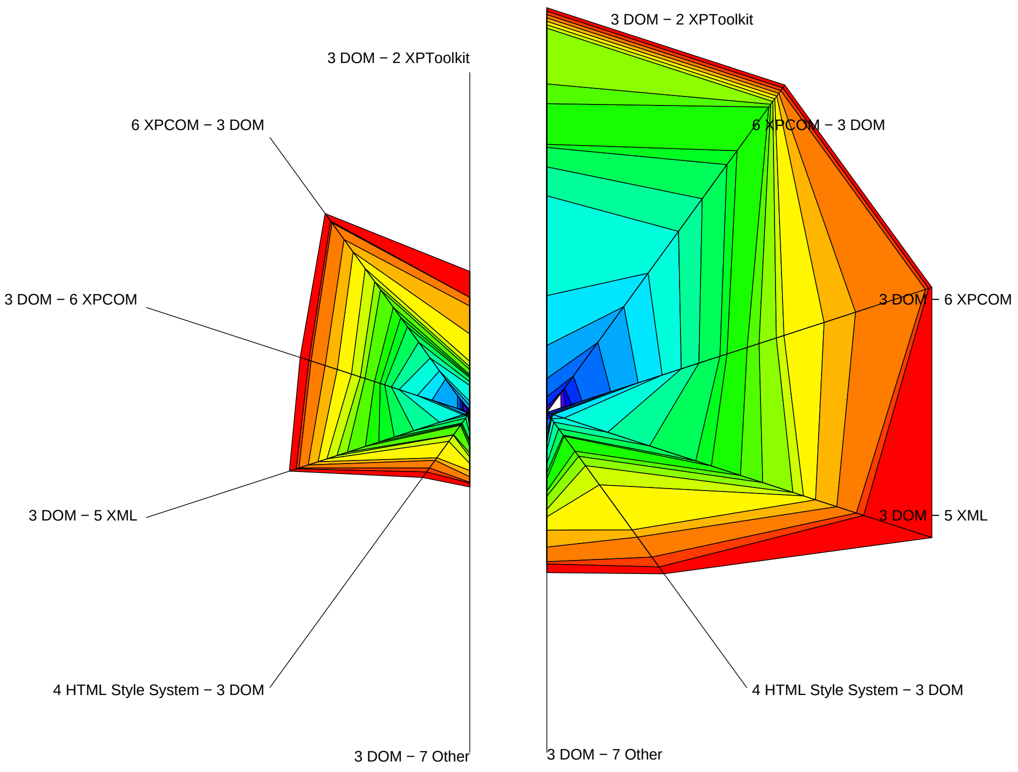


Figure 5. DOM related invocations in Mozilla 1.4 (left) and 1.7 (right)

ing the development of this view we noticed, that presentation quality suffers from the wide spread of invocation frequencies that can differ by an order of magnitude of 5. As solution, we introduced five frequency classes according to the overall maximum number of invocations. Each class has a fixed size so we get data sets with maximum value $\in [0.2, 0.4, 0.6, 0.8, 1.0]$. The data are then scaled to the desired size during diagram generation. As the forth dimension in our visualizations we have the time dimension. We decided to use a *tree-ring scheme* based on the rainbow colors to depict the twenty intervals: blue indicates the first interval (most inner ring) and red indicates the last interval. Since the values are scaled to different maxima—one maximum for each Mozilla version—sizes between both diagrams must be compared to absolute values from the database. A quick comparison indicates that the communication in version 1.7 is more distributed compared to version 1.4 of Mozilla. In contrast to Figure 3 where the changes in the invocation frequency are not directly recognizable, the matrix type view depicted in Figure 4 supports perception of these changes in an intuitive way.

Striking are the high number of invocations between `XPCOM` and `Other` in Figure 4.(a) whereas Figure 4.(b) shows a more distributed characteristic for function invocations in version 1.7. Another interesting result is that communication starts earlier in version 1.7—e.g., `XPCOM - HTML Style System`—compared to the predecessor version (which is also supported by the Gantt diagrams). This can be interpreted in such a way, that the system has been optimized and web pages are now delivered faster to the user.

Next, we give a more detailed view of one selected software module with respect to invocations with other modules.

3.3.3 Detailed module view using Kiviat diagrams

As result of the *EvoTrace* approach, we obtain multidimensional data sets. To overcome some of the limitations of the previous views, we decided to use Kiviat diagrams for a detailed view on the communication between modules. Two diagrams covering a range of 180° each, face by face, allow a quick comparison of specific module data between two releases. Based on the experiences with the wide value range we sorted values in ascending order and limited the result set to the eight most frequent invoked module pairs. Further modifications concern the scaling of the data sets during diagram generation. For data representation we use a 4-dimensional dataset. The actual value of each data point in the diagram is determined by the following scaling formula:

$$v_{k,a,b,s} = \begin{cases} \frac{2}{\max} F & \text{if } 2F < 1 \\ 1 & \text{if } 2F = 1 \\ 1 + 0.01 * s & \text{if } 2F > 1 \end{cases}$$

whereas $\max$ is the overall maximum of F, and

$$F = \sum_{s \leqq n} f_{k,a,b,s}$$

is the cumulated value if invocations between module M_a and M_b for version k of Mozilla over the time-slots $s \leqq n$ ($n \in \{0, 1, ..., 19\}$). Division of the maximum by a constant factor together with the $2F > 1$ branch, reduces the biasing effect of "spikes" diagram drawing.

The resulting diagram for module `DOM` is depicted in Figure 5 whereas the `DOM - XPToolkit`, `XPCOM - DOM`, and `DOM -XPCOM` are reduced in size ($2F > 1$ branch). In contrast to the matrix view, the data sets for the different releases are scaled with a common factor. Thus both sides of of the diagram are directly comparable. Our example graph indicates for the modules `DOM` and `XPToolkit` that the communication has been doubled. Especially during the center period (green area) the increment was substantial. This perception is supported by data from the database where 50,190 invocations for version 1.4 and 105,001 for version 1.7 have been recorded. Further interesting are the delays when communication starts with relevant modules. Examples for early communication (inner blue area) in version 1.4 are `DOM` - `XPCOM` (both directions) and `DOM` - `Other`, respectively. Compared to version 1.7 no significant changes can be found, except for `XPCOM` - `XPToolkit` where communications starts now earlier. As a counter example we refer to the pair `HTML Style System` - `DOM` where communication starts late in both program versions.

Another interesting area of application is the deduction of *uses* relationships. This is facilitated by this diagram type, since results are sorted by frequency and further subdivided than in the Gantt diagram or Matrix view. As depicted in Figure 5 most communication takes place in a single direction between modules. Counter examples are `DOM` - `XPCOM` and `DOM` - `Other`. But `Other` is a virtual module–a superset of modules we did not further break down–and therefore its contribution is not significant.

3.3.4 Discussion

With a traditional database approach large amounts of trace data can be handled efficiently and the database queries are simple to implement and access via standard SQL query interface for third party tools is possible. Another advantage is that storing the program traces in the RHDB supports fast retrieval and detection of a system's interaction patterns without losing context related detail information. During our experiments access speed was not in issue. The detection of invocation sequences of a single trace with more than $19 \cdot 10^6$ events using a Java program and MySQL database on a Pentium 4, 2.8GHz, 1GB takes less than 5 minutes, which we considered reasonably fast. If a speed up for pattern detection is required, the problem can be nicely partitioned via invocation levels.

Though some of the results can be achieved with data from conventional profiling as well, focus of the *EvoTrace* approach is the evaluation of program traces for evolution analysis. Our visualizations provide insights into changes on arbitrary detailed level to track the changes between system releases.

4 Related Work

Most related work we have seen so far either focuses on the generation of evolution traces based static software artifacts such as the number of lines of code of the individual modules of the software system, presenting the age of the code, or the correlation between module changes and programmers [1, 3, 4].

Other reverse engineering approaches try to take the dynamic execution behavior into account and try to infer certain program characteristics based on these traces. For instance, In [9], for instance, Gschwind et al. present an approach that allows to identify how certain features within a program are implemented. This approach is based on execution traces and interactive program queries during the program's runtime. A similar approach is taken by the Smiley system presented in [8]. For this system, Goldman uses wrappers to log the interaction between an application and its external dynamic link libraries (DLL). This work facilitates the understanding of interactions between COTS where no source code is available.

In [14] Wilde et al. describe how runtime profile information can be used to map features onto source code. In earlier work of our group we used these technique as well. The reason we are using execution traces is that when compared with profiling information they contain more relevant information such as thread data or interaction patterns. The work that comes closest to the work presented in this paper was presented by Collberg et al. in [2]. In their paper, they present an approach that takes possible executions into account by analyzing the evolution of the program's call-graph. This is accomplished by generating call graphs for the different versions of the program, merging these call graphs, and finally highlighting the differences between the call graphs.

Analyzing the differences in the call graph, however, still falls short in getting a glimpse of the typical runtime behavior of the program to be analyzed since the call graph does not give any information about how frequently certain functions are being invoked and hence does not give a huge insight into the communication patterns between different parts of the program. This is especially the case if call-back functions are being used which cannot be easily identified on the basis of the call graph.

Other related work was presented by Jerding et al. [12] and Hamou-Lhadj et al. [11]. Both present different but similar techniques to identify patterns and similarities in execution traces. This allows them to compress execution traces and store them in a more compact form. In [11] Hamou-Lhadj et al. provide a survey about other trace exploration tools and techniques. None of these approaches, however, analyze the differences between different execution traces. In the future, however, we plan to use such compression techniques to analyze the differences on a finer grained level and to identify the similarities and differences between the traces of different releases. As we have mentioned previously, currently we divide the execution traces into 20 segments which we assume are similar.

5 Conclusions and Future Work

Dynamic information expressed in execution traces of a software system can be used to understand some evolution aspects. Especially in pointing a software engineer to locations in a system that may be critical for maintenance activities.

Comparing execution traces is a simple but efficient way to gain information about changes in the as-implemented architecture without the need to parse or have direct access to the source code. This information can be used to recover interaction patterns between different entities such as methods, files, or modules.

In this paper, we proposed a methodology to analyze and compare the execution traces of different versions of a software system to provide insights into its evolution. We recover high-level module views that facilitate the comprehension of each module’ s evolution in relation to others. *EvoTrace* allows us to track the evolution of particular modules and present the findings in three different kinds of visualizations: Gantt diagrams, Matrix views and Kiviat diagrams. Based on these graphical representations, we have shown that certain aspects such as invocation structures between modules can be tracked and comprehended much more effectively.

We showed the applicability of our approach using the Mozilla open source system consisting of about 2 MLOC in C/C++. For example, we could determine the evolution from a “dispatcher” oriented communication in version 1.4 to a more direct communication between software modules in version 1.7.

Execution trace data are another cornerstone for software evolution analysis. The properties of execution traces, such as detailed information about “scheduling” data, invocation patterns, call frequency, nesting levels, or threading, can complement results gained via release history and architectural analysis. Further research will have to show how dependencies between these three dimensions can be exploited.

For future work, we plan to integrate existing pattern detection approaches to reduce the amount of information and improve the comparability of different software versions. Further, we want to develop an automatism to compute differences for given execution traces which would allow us a more focused analysis.

References

[1] T. Ball and S. G. Eick. Software Visualization in the Large. *IEEE Computer*, 29(4):33–43, 1996.

[2] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 77–ff. ACM Press, 2003.

[3] S. Eick, J. Steffen, and E. Sumner. Seesoft–A Tool For Visualizing Line Oriented Software Statistics, 1992.

[4] S. G. Eick, T. L. Graves, A. F. Karr, A. Mockus, and P. Schuster. Visualizing Software Changes. *Software Engineering*, 28(4):396–412, 2002.

[5] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[6] M. Fischer and H. Gall. Visualizing Feature Evolution of Large-Scale Software based on Problem and Modification Report Data. *Journal of Software Maintenance and Evolution*, 16(6):385–403, November/December 2004.

[7] M. Fischer, M. Pinzger, and H. Gall. Populating a Release History Database from Version Control and Bug Tracking Systems. In *Proceedings International Conference on Software Maintenance (ICSM’03)*, pages 23–32, September 2003.

[8] N. M. Goldman. Smiley - an interactive tool for monitoring inter-module function calls. In *Proceedings of the 8th International Workshop on Program Comprehension (IWPC’00)*, pages 109–118. IEEE Computer Society, 2000.

[9] T. Gschwind, J. Oberleitner, and M. Pinzger. Using run-time data for program comprehension. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC’03)*, pages 245–250. IEEE Computer Society Press, May 2003.

[10] A. Hamou-Lhadj and T. C. Lethbridge. Compression techniques to simplify the analysis of large execution traces. In *IWPC ’02: Proceedings of the 10th International Workshop on Program Comprehension*, pages 159–160. IEEE Computer Society, June 2002.

[11] A. Hamou-Lhadj and T. C. Lethbridge. A survey of trace exploration tools and techniques. In *CASCON '04: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 42–55. IBM Press, 2004.

[12] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *Proceedings of the 19th International Conference on Software Engineering (ICSE)*, pages 360–371. IEEE Computer Society Press, May 1997.

[13] M. Pinzger, H. Gall, and M. Fischer. Towards an integrated view on architecture and its evolution. In *Proceedings of the Software Evolution through Transformations: Model-based vs. Implementation-level Solutions (SETra'04)*. Elsevier Science Publishers: Utrecht, Netherlands, 2004. to appear.

[14] N. Wilde and M. C. Scully. Software Reconnaissance: Mapping Program Features to Code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, January 1995.