Execution Trace Analysis through Massive Sequence and Circular Bundle Views

Bas Cornelissen [a] Andy Zaidman [a] Danny Holten [b]
Leon Moonen [a] Arie van Deursen [c] Jarke J. van Wijk [b]

[a] *Delft University of Technology*
{*s.g.m.cornelissen, a.e.zaidman*}*@tudelft.nl, Leon.Moonen@computer.org*

[b] *Eindhoven University of Technology*
d.h.r.holten@tue.nl, vanwijk@win.tue.nl

[c] *Delft University of Technology & CWI*
arie.vandeursen@tudelft.nl

Abstract

An important part of many software maintenance tasks is to gain a sufficient level of understanding of the system at hand. The use of dynamic information to aid in this software understanding process is a common practice nowadays. A major issue in this context is scalability: due to the vast amounts of information, it is a very difficult task to successfully navigate through the dynamic data contained in execution traces without getting lost.

In this paper, we propose the use of two novel trace visualization techniques based on the massive sequence and circular bundle view, which both reflect a strong emphasis on scalability. These techniques have been implemented in a tool called Extravis. By means of distinct usage scenarios that were conducted on three different software systems, we show how our approach is applicable in three typical program comprehension tasks: trace exploration, feature location, and top-down analysis with domain knowledge.

Key words: Program Comprehension, Dynamic Analysis, Execution Traces, Visualization

1 Introduction

Software engineering is a multidisciplinary activity that has many facets to it. In particular, in the context of software maintenance, one of the most daunting tasks is to *understand* the software system at hand. During this task, the software engineer attempts to build a mental map that relates the system's functionality and concepts to its source code [28,22].

Understanding a system's behavior implies studying existing code, documentation, and other design artifacts in order to gain a level of understanding that is sufficient for a given maintenance task. This *program comprehension* process is known to be very time-consuming, and Basili reports that 50 to 60% of the software engineering effort is spent on understanding the software system at hand [2]. Thus, considerable gains in overall efficiency can be obtained if tools are available that facilitate this comprehension process. The greatest challenge for such tools is to create an accurate image of the entities and relations in a system that play a role in a particular task.

Dynamic analysis. Dynamic analysis, or the analysis of data gathered from a running program, has the potential to provide an accurate picture of a software system: e.g., in the context of object-oriented systems it can reveal object identities and occurrences of late binding. Furthermore, through the careful selection of an execution scenario, a goal-driven program comprehension strategy can be followed [37]. The data is obtained through the instrumentation and execution of a system, which (in the case of a post mortem analysis) results in one or more *execution traces* that are to be analyzed.

Challenges & goal. The main issue in the context of dynamic analysis approaches for program comprehension is the enormous amount of data that is collected at run-time, since it gives rise to scalability issues [36]. Particularly in the case of a sizable program, the main challenge for any dynamic analysis based technique is to convey both the program's large structure *and* its many interrelationships to the user, such that the available screen real estate is used efficiently. This is not a trivial task, and straightforward visualizations typically do not suffice because they often require two-dimensional scrolling, thus hindering the comprehension process [35]. An example of such visualizations are UML sequence diagrams which we reconstructed in earlier work [5].

The goal of this paper is the development of new techniques that allow the visualization of dynamically gathered data from a software system in a condensed way, while still being highly scalable and interactive.

Visualization approach. We attempt to achieve our goal by presenting two

synergistic views of a software system. The first view is the *circular bundle view* that projects the system's structural elements on the circumference of a circle and visualizes the call relationships in between. The second view, the *massive sequence view*, provides an interactive, high-level overview of the traced events. These techniques are implemented in our tool EXTRAVIS (EXecution TRAce VISualizer).

To characterize our approach, we use the framework introduced by Maletic et al. [23]:

(1) *Task: Why is the visualization needed?* The amount of trace data that often results from dynamic analysis, calls for a visualization that represents an execution trace in a concise and interactive manner. More specifically, we describe how our approach is useful for three representative program understanding scenarios:
   - trace exploration;
   - feature location; and
   - top-down program comprehension with domain knowledge.

(2) *Audience: Who will use the visualization?* The target audience consists of software developers and re-engineers who are faced with understanding (part of) an unknown software system.

(3) *Target: What low level aspects are visualized?* Our main aim is to represent information pertaining to call relationships, and the chronological order in which these interactions occur. This information is augmented with static data to establish the system's structural decomposition.

(4) *Representation: What form of representation best conveys the target information to the user?* We strive for our visualization to be both intuitive and scalable. To optimize the use of screen real estate, we represent a system's structure in a circular view. Moreover, our massive sequence view presents an interactive overview.

(5) *Medium: Where is the visualization rendered?* The visualization is built up from two synchronized views that are rendered on a single computer screen.

Our approach enables software engineers to quickly gain an understanding of unfamiliar software systems, thus enabling related tasks such as software maintenance to be performed more efficiently. We illustrate this through the application of our tool implementation in the context of trace exploration, feature location, and top-down analysis. These studies involve two open source programs and an industrial system.

With respect to previous work [6], we have made the following extensions. First, we present the results of an elaborate new case study involving a new, larger software system. Second, two existing case studies have been substantially revised and extended. Third, we provide a more thorough discussion of

the benefits and drawbacks of our approach, and of potential validity threats.

Contributions. This paper makes the following contributions.

- A novel approach to execution trace visualization that is based on two linked views: (1) the circular bundle view that displays the structural elements and bundles their call relationships, and (2) the massive sequence view that provides an interactive overview.
- The application of our tool implementation on three distinct software systems in three different program comprehension contexts.

Structure of the paper. In Section 2 we discuss the existing visualization techniques that our approach relies on, while Section 3 provides a detailed description of our own approach. Section 4 introduces our experimental setup and case studies, while Sections 5 through 7 deal with the three case studies that we have performed. Section 8 discusses benefits and drawbacks of our approach and Section 9 outlines related work. Section 10 summarizes our work and provides pointers for future work.

2 Existing work

The two synergistic views that we propose in our approach are based on a number of existing information visualization techniques. In this section we briefly introduce these existing techniques and point out how our approach differs from their original application.

2.1 Circle approach

In order to address the issue of visualizing the large number of structural entities that constitute a software system, we propose to employ a circle approach in which all structural elements are projected on the circumference of the circle.

Hierarchical edge bundles. To depict a system's structural information we use the circle approach from Holten [15], who proposed to project all of a system's structural entities on the outline of a circle and to draw their relations in the middle. The entities are presented in a nested fashion on the circle in order to convey their hierarchical properties, e.g., package structures or architectural layers.

Within the circle the relationships between the structural elements are drawn.

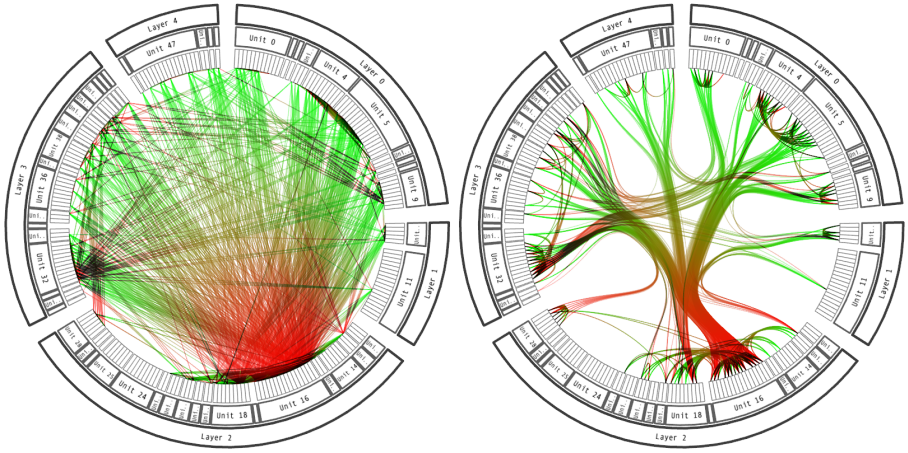


Fig. 1. Call relations within a program shown using linear edges (left) and using hierarchical edge bundles (right).

These relations are depicted by bundled splines (Figure 1): the visual bundling of relations helps to reduce visual clutter, and also shows the implicit relationships that exist between parent elements resulting from explicit calls between their respective children. These *hierarchical edge bundles* were used in [15] to depict static dependencies; we enrich this visualization so that it can show dynamic information.

2.2 Message sequence charts

The technique that we propose to use for the interactive visualization of time-ordered events builds upon the notion of message sequence charts. Message sequence charts are commonly used to visualize a series of chronologically ordered interactions between the entities of a system [3]. Their main advantage is readability: the fact that the events are ordered from top to bottom makes the diagrams intuitive for humans. Scalability, however, is an important limitation for this technique: the charts rapidly become hard to navigate when dealing with too much information, which makes them less suitable for large amounts of entities and interactions.

Information murals. To tackle the scalability issue in visualizing large amounts of dynamic information, Jerding and Stasko proposed the "information mural" [18]. This technique creates a two-dimensional miniature version of the information space and is appropriately scaled to fit on one screen. In this paper, we use a similar technique to visualize large-scale message sequence charts: we show a system's *entire* (nested) structure on the horizontal axis, while plotting the interactions as rectangles along the vertical axis. The rectangles are appropriately colored to indicate the directions of the calls. The purpose of the resulting view is to provide a navigable overview of an execu-

tion trace.

2.3 Visualization criteria

When constructing new techniques to visualize large amounts of data in a comprehensible way, we need criteria that capture "comprehensibility". Taken from the realm of visual programming languages, we discuss two properties that express these criteria and represent a set of important requirements [35]:

Accessibility of related information. It is essential that related information is viewed in close proximity. When considering two objects in a visualization that are not close to each other, there is the psychological claim that these objects are not closely related, or are not dedicated to solving the same problem. Translated to the field of software visualization, we observe two dimensions that pertain to this criterion:

(1) Structural entities that are bound by a parent-child relation should be visualized in close proximity of each other.
(2) Structural entities that participate in the execution in a particular time interval should be visualized in close proximity of each other.

Use of screen real estate. The term "screen real estate" refers to the size of a physical display screen and connotes the fact that screen space is a valuable resource. During information comprehension tasks, it is of great importance to make optimal use of the available screen real estate in order to prevent excessive scrolling and to reduce the effort from the user's part.

Keeping these criteria in mind, any *trace visualization* technique faces a twofold challenge: it must depict (1) a potentially large number of structural entities, all the while keeping related entities relatively close together, and (2) massive amounts of run-time information without confusing the viewer, e.g., preferably without the need for scrolling.

3 Our Approach

The techniques that we described in the previous section have been implemented in a prototype tool called Extravis. Given an execution trace (or part thereof), Extravis presents two linked views:

- the *circular bundle view* that shows the system's structural decomposition and the nature of its interactions during (part of) an execution trace;

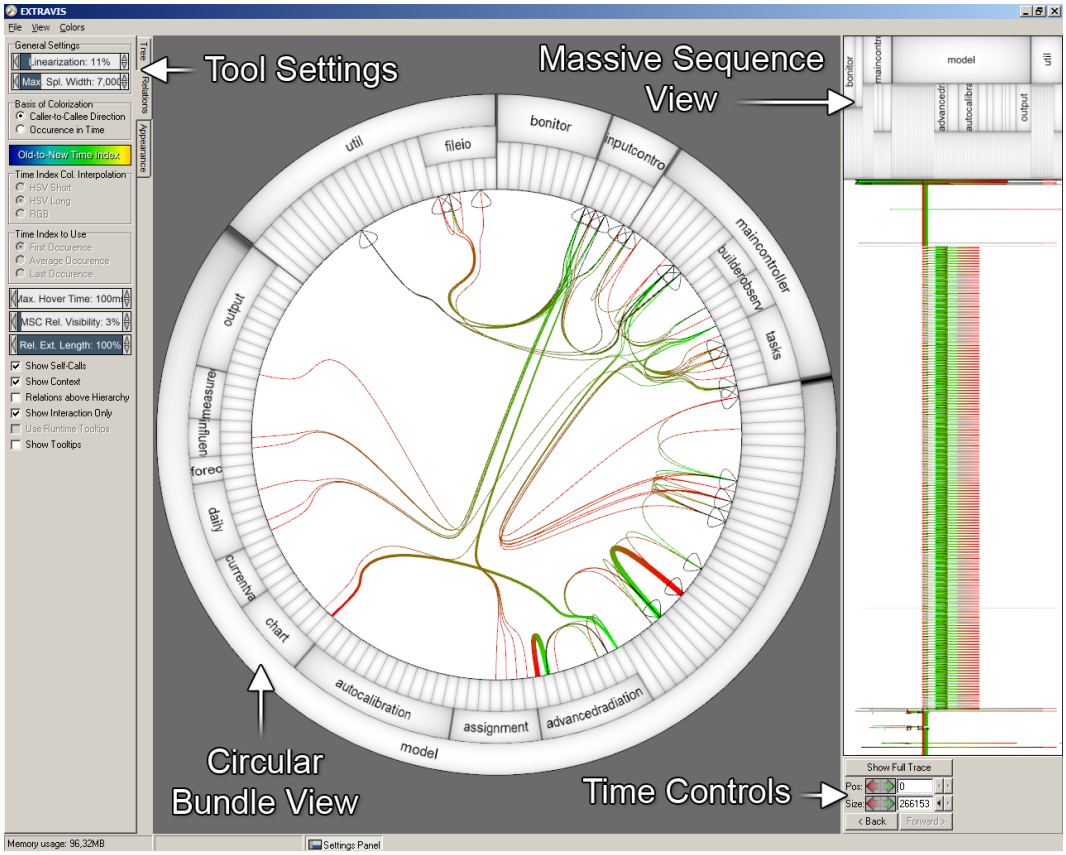


Fig. 2. Full view of an entire Cromod trace.

- the *massive sequence view* that provides a concise and navigable overview of the consecutive calls between the system's entities in a chronological order.

The tool's user interface is depicted in Figure 2, which illustrates the actual views in the context of a large execution trace [1] . Both views offer multiple interaction methods and detailed textual information on demand, and a synchronized mode of operation ensures that changes in the one view are propagated to the other. In this section, we discuss the meta-model used by Extravis and describe the two views in more detail.

3.1 Meta-model

Extravis is based on a meta-model that describes the structural decomposition of the system (a *contains* hierarchy) and a time-stamped call relation. Optionally, additional relations can be supplied which contain more detailed

[1] The figures in this paper are best viewed in color, and are also available in hi-res at `http://www.swerl.tudelft.nl/extravis/`.

information.

Structural information. To visualize the structure of a program, the tool requires a *containment* relation that defines the system's structural decomposition. In this context, one could think of package or directory structures, or architectural layers.

Basic call relations. The second mandatory part of the meta-model is a series of *call* relations which are extracted from an execution trace. The input thus contains information on the caller and callee's classes, the method signatures, and the chronological order of the calls (by means of an increment). Additionally, to link with the source code, the method signatures contain pointers to the source files (if available) and include the relevant line numbers.

Detailed call relations. In case the execution trace is rich in the sense that detailed call information is available, the meta-model also allows the specification of such data as object identifiers, run-time parameters, and actual return values.

3.2 Circular Bundle View

The first of the two views, the circular bundle view, offers a detailed visualization of the system's structural entities and their interrelationships. At the basis of this view lie the techniques that were proposed in Section 2.1:

- The projection of the software system's structural elements on the circumference of a circle, including their hierarchical structuring.
- The visual bundling of the relationships between these elements in the circle.

Furthermore, we have made several enhancements to further facilitate the comprehension process.

First, the high-level structural entities can be *collapsed* to enable focusing on specific parts of the system. As is illustrated in Figure 4, collapsing an element hides all of its child elements and "lifts" the relations pertaining to these child elements to the parent element, thus providing a straightforward abstraction mechanism. The (un-)collapsing process is fully animated for the user to maintain a coherent view of the system, i.e., to facilitate the cognitive linking of the "pre" and "post" view.

Secondly, the circular bundle view provides a snapshot in time that corresponds to the part of the execution trace that is currently being viewed. As

such, the hierarchical edge bundles visualize the interactions occurring during a certain time interval. Edges are drawn between the elements that communicate with each other, and the *thickness* of an edge indicates the number of calls between two elements, thus providing a measure for their degree of coupling. Furthermore, textual information related to the underlying source code is provided by means of call highlighting (i.e., by hovering over an edge) and by providing direct links to the relevant source parts.

Finally, with respect to the coloring, the user can choose from either the *directional* or the *temporal* mode. In the former case, a color gradient along the edge indicates its direction. The latter mode colors the edges such that the calls are ordered from least recent (light) to most recent (dark).

3.3 Massive Sequence View

To support users in navigating through traces and identifying parts of interest, Extravis offers the massive sequence view. Being a derivative of the information mural, it provides an overview of (part of) an execution trace in which the directions of the relations are color coded using a gradient (Figure 2). Additionally, the massive sequence view allows to *zoom in* on parts of the execution trace by the selection of a fragment that needs closer inspection.

3.3.1 Importance-Based Anti-Aliasing

When visualizing information murals and large-scale message sequence charts in particular, it becomes apparent that the amount of available pixel lines on a normal display is not sufficient. Rather than visualizing every event, one typically resorts to such measures as abstraction and linear anti-aliasing [19]. While being useful in maintaining the big picture, certain potentially useful *outlier calls* may not "survive" these measures.

To keep the loss of such calls to a minimum, we propose a new anti-aliasing technique that uses an importance-based blending mode to calculate a pixel line's average color. Calls are weighted depending on the frequency with which they appear within a certain time frame, and calls with small frequencies are emphasized. This technique is called *Importance-Based Anti-Aliasing* (IBAA) and ensures that outlier calls remain visible in the proximity of thousands of calls, thus ensuring that potentially important calls are not missed by the user. A detailed description is provided in [16].

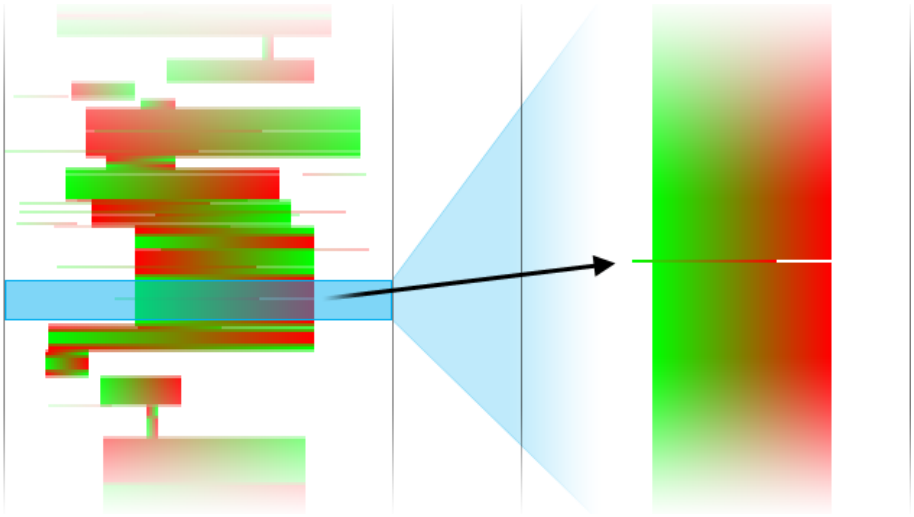


Fig. 3. Using Importance-Based Anti-Aliasing (IBAA) to visualize outlier calls.

3.4 View interaction

An important strength of our approach is the *synergy* between the two views. The views are linked in the sense that user interactions in the one view are visible in the other. This ensures that the user maintains a coherent view of the system during all view interactions.

An example is the collapsing process that was described earlier. Collapsing a structural entity in the circular view hides its interrelationships and aggregates its relations with other entities; this results in an abstraction that is propagated to the massive sequence view, in which the structural hierarchy and the series of calls are modified accordingly. Additionally, the user may zoom in on part of the massive sequence view, thus reducing the time frame under consideration in both views.

Another example that illustrates the usefulness of the linked views concerns highlighting. In the circular view, the viewer can select a structural element, upon which the massive sequence view shows the interactions involving this element by graying out the other calls. Selecting two elements, on the other hand, highlights their mutual interactions. Hovering over a call in either of the two views shows its occurrence(s) in the other view, and spawns a tooltip that describes its nature (e.g., the method signature and call site information).

3.5 Implementation

This section focuses on the technologies that were used to build Extravis.

The front-end of EXTRAVIS is written in Delphi and makes heavy use of OpenGL. For extracting a system's class decomposition from its directory structure, we make use of a simple Perl script. As for the dynamic part, we trace a system's execution by monitoring for (static) method and constructor invocations, and registering the objects that are involved. We achieve this by extending the SDR framework from our earlier work [5], which incorporates a tracer that employs AspectJ [1] for the instrumentation. During an execution, this tracer registers unique objects, method and constructor names, information on the call sites (i.e., source filenames and line numbers), runtime parameters, and actual return values, while a custom-built event listener converts the events to our tool's input format.

EXTRAVIS is available for download and requires a modern Windows PC to run [2]. A modern graphics card is highly recommended in order to fully benefit from the importance-based anti-aliasing.

4 Case Studies

To illustrate the effectiveness of our techniques, we have conducted three distinct case studies. Each of these studies is centered around a specific use case for our techniques, and is representative for program comprehension challenges that are faced by software engineers in everyday life. While in certain cases there is some a priori knowledge regarding the user-level functionality of the system at hand, no implementation details are known in advance.

Trace exploration (Section 5)

- *Context:* The system is largely unknown, but an execution trace is available. No (or little) up-front knowledge is present.
- *Task:* Identify the phases that constitute the system's execution, and study its fan-in and fan-out characteristics.
- *Goal:* Get an initial feeling of how the system works, as a basis for a more focused examination.

Feature location (Section 6)

- *Context:* The user-level functionality of the system is known. The nature of the system is such that the features can be invoked at the user's discretion.
- *Task:* The execution of a scenario in which a set of features is invoked, and the visual detection of these features in the resulting trace.

[2] Available at `http://www.swerl.tudelft.nl/extravis/`.

- *Goal:* The establishment of relations between feature invocations and the corresponding source code elements.

Top-down program comprehension with domain knowledge (Section 7)

- *Context:* The system is unknown, and the user has little control over its functionality since it concerns a batch execution. However, the system's description provides clues as to the behavioral aspects (i.e., the execution phases) that are to be expected.
- *Task:* Using domain knowledge, formulate a hypothesis describing a set of conceptual phases, and validate it through the analysis of a typical trace.
- *Goal:* The use of a top-down approach to gain and refine (detailed) knowledge of a system's inner workings.

Each of these use cases is exemplified by means of a typical usage scenario that involves a medium-scale Java system [3].

5 Case Study 1: Trace Exploration

5.1 Motivation

When a system is largely unknown and an execution trace is available, being able to globally understand the control flow in the trace can be of great help in understanding the system. Particularly in the context of a legacy system that lacks a thorough documentation, any information on the system's inner workings is welcome. However, since execution tracing tends to result in large amounts of data, the *exploration* of such traces is by no means a trivial task. To illustrate how our techniques facilitate this process, we explore an industrial system called CROMOD.

5.2 Cromod

CROMOD is an industrial Java system that predicts the environmental conditions in greenhouses. The system is built up from 145 classes that are dis-

[3] Note that although these experiments involve Java because our tool-chain is optimized for Java systems, we are confident that our technique is applicable to other (non-object-oriented) languages.

tributed across 20 packages. According to the manual, it takes a greenhouse configuration (e.g., four sections, 15 shutters, and 40 lights) and a weather forecast as its input; it then calculates the optimal conditions and determines how certain parameters such as heating, lights, and shutters should be controlled; and then writes the recommended values. Since the calculations are done for a large number of discrete time frames, they typically induce massive amounts of interactions, which makes this system an interesting subject for trace visualization.

5.3 Obtaining the trace

The trace that results from a typical CROMOD execution contains millions of events, of which a large part can be attributed to logging. At the recommendation of the developers, we have run the program at a log level such that the resulting trace contains roughly 270,000 method and constructor calls. This trace captures the essence of the execution scenario, and in terms of size, the visualization and comprehension of this trace remain a challenge. We conclude the setup with the extension of the trace with information on the system's hierarchical decomposition in terms of its package structure.

5.4 Analyzing the trace

Loading the trace into EXTRAVIS provides us with the initial views that are shown in Figure 2.

5.4.1 Studying fan-in and fan-out behavior

The circular view of the trace shows CROMOD's structural decomposition and (the frequency of) the calls that occurred during execution. For example, we can see several edges that are thicker than usual, which suggests that most of the activity is centered around these particular calls. What is also noteworthy is that in the vicinity of certain packages the edges are predominantly colored red, which indicates a high degree of fan-in: the only outgoing calls here seem to be directed toward classes within the package. Examples of such packages are `util.fileio` and `model`: Figure 4 confirms our assumption by collapsing these packages, which results in aggregated relations that are clearly incoming in nature. From this observation, we draw the conclusion that these packages fulfill a library or utility role.

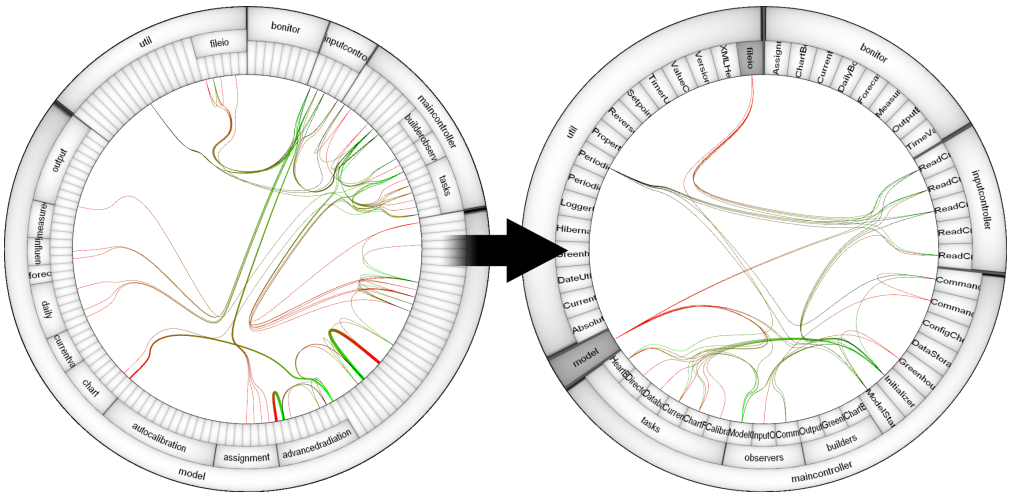


Fig. 4. Collapsing the `model` and `util.fileio` packages in the Cromod trace.

5.4.2 Identifying phases

The massive sequence view indicates that there are three major "phases" in the execution scenario. The first and third phases are characterized by two small beams; in between we observe a long segment that appears to be somewhat broader, and shows a very consistent coloring. At this point in time, we made the hypothesis that the three stages concern (1) an input phase, (2) a calculation phase, and (3) an output phase. We attempt to validate this hypothesis in the following steps.

The first phase that we can visually discern looks like an almost straight vertical "beam". We zoom in on this phase by selecting a suitable interval, thus reducing the time frame under consideration. Now, Extravis only visualizes the interactions within the chosen time frame in both views. Turning our attention to the circular view (Figure 5), we learn that this first phase merely involves a limited number of classes and packages, and judging by the names of the packages and classes involved (e.g., `inputcontroller.ReadCromodForecast` and `fileio.InputFileScanner`), this phase mainly concerns I/O activity with an emphasis on input processing.

A quick glance through the second phase reveals a massive amount of recurrent calls within the `model` package. The phase is mainly made up from the construction of `model.advancedradiation.Sun` objects by `model.advancedradiation.SolarModel`, and the creation of `model.Time` instances by `model.TimeValue`. Indeed, out of the 270,000 events in this execution scenario, 260,000 events concern calls to constructors. From these findings we conclude that this phase is concerned with the Cromod's main functionality, i.e., the model calculations.

Another noteworthy observation in the second phase is the occasional appear-

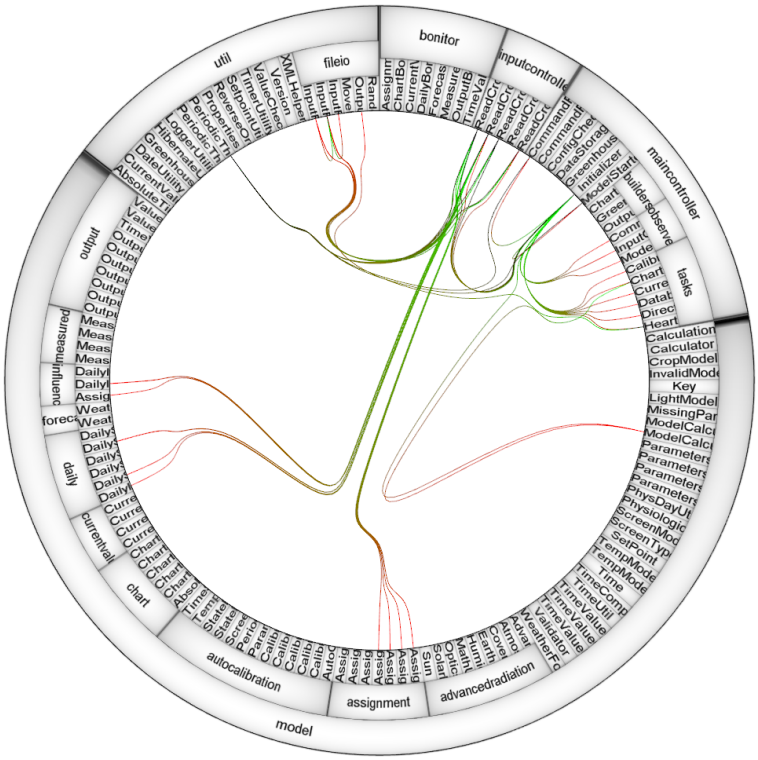


Fig. 5. Circular view of Cromod's initialization phase.

ance of very thin lines in the massive sequence view of Figure 2. A closer look reveals that the aforementioned construction processes are incidentally interleaved with short fragments that involve readers of CROMOD's XML inputs. Focusing on one such fragment (Figure 6), we learn that `util.PeriodicThread.run()` is invoked on `inputcontroller.ReadCromodForecast`, which results in a call to `util.fileio.InputFileScanner.scanForFiles()`. This example illustrates the purpose of our anti-aliasing technique: without the use of IBAA, on a typical display that offers 1,200 pixel lines, and given a trace size of 270,000 calls, the odds of two calls ending up in the resulting massive sequence view would have been less than 1%.

The third phase is similar to the first phase: here we observe interactions between `maincontroller.tasks.CurrentValuesGenerator` and `util.fileio.OutputFile` which again suggests I/O activity, focused on output rather than input. This provides strong evidence for the validation of the hypothesis formulated earlier.

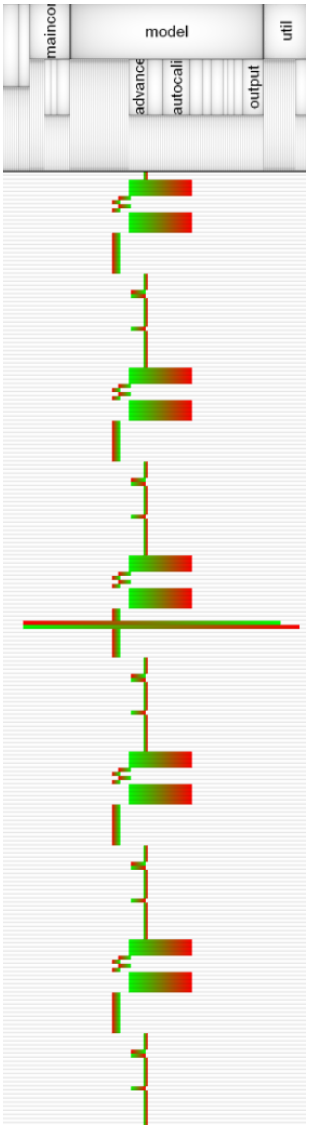


Fig. 6. Zoomed massive sequence view of Cromod's second phase, focusing on a periodic fragment.

5.5 Discussion

In this experiment, we explored a typical execution trace of an industrial system of which we had very little knowledge in advance. Our visualization techniques proved to be very useful in this context, and rapidly helped us gain a certain degree of knowledge of the system.

More specifically, the circular bundle view showed (1) CROMOD's fan-in and fan-out behavior, and (2) the distribution of the events across the system. Moreover, the collapsing of packages aggregates the interrelationships of these packages, rendering both the circular and the massive sequence view easier to read.

The massive sequence view provided an overview of the trace and indicated the existence of three major phases in the execution. We used zooming and call highlighting to learn more about the nature of these phases. The use of IBAA pointed out a series of outlier calls.

6 Case Study 2: Feature Location

6.1 Motivation

A significant portion of the effort in a maintenance task is spent on determining *where* to start looking within the system. In the context of specific features, this process is called *feature location* [34]. Its purpose is to relate a system's features to its source code, which enables an engineer to focus on the code fragments that are relevant to a feature's implementation (e.g., to handle change requests). While research into feature location is often concerned with automatic techniques such as concept analysis [10] and Latent Semantic Indexing [25], we will attempt to *visually* [20] locate certain features in an execution trace. In this context, we consider a feature to be a user-triggerable unit of functionality [11].

6.2 JHotDraw

JHotDraw [4] is a well-known, highly customizable Java framework for graphics editing. It was developed as a "design exercise" and is generally considered to be well-designed. It comprises 344 classes and 21 packages. Running the program presents the user with a GUI in which there is a set of features that may be invoked at the user's discretion, such as opening a file, inserting predefined figures or manual sketches, and adding textual information. While the authors are familiar with JHotDraw's user-level functionality, its inner workings are unknown in advance.

6.3 Obtaining the trace

To generate a suitable feature trace, we use an execution scenario that involves several major features that we want to detect: the creation of a new drawing, and the insertion of *five* different types of figures therein. These figures include rectangles, rounded rectangles, ellipses, triangles, and diamonds. To make the

[4] JHotDraw 6.0, `http://www.jhotdraw.org/`

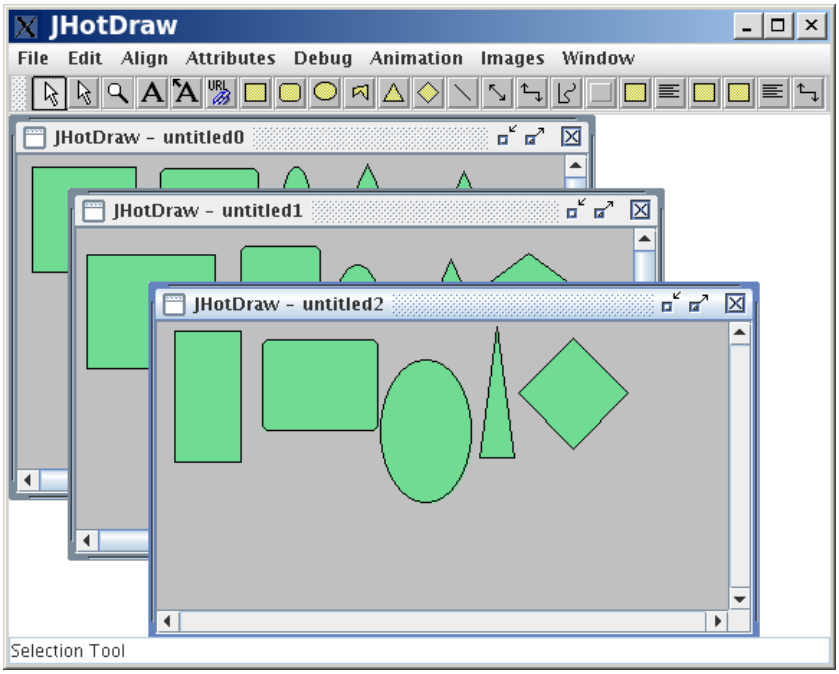


Fig. 7. Execution scenario for JHotDraw, in which five different figures are inserted in three distinct drawings.

localization of the "new drawing" and the "insert figure" features easier, we invoke the aforementioned scenario a total of *three* times (Figure 7).

Since JHotDraw registers all mouse movements, the trace that results from our scenario is bound to contain a lot of noise. We have therefore filtered these mouse events to obtain a trace that is somewhat cleaner. The resulting trace contains a little over 180,000 events.

6.4 Analyzing the trace

Figure 8(a) shows the massive sequence view of the entire execution trace, in which we can immediately observe several recurrent patterns.

6.4.1 Locating the "new drawing" feature

Since in our trace scenario we invoked the "new drawing" feature three times, we are looking for a pattern that has the same number of occurrences. Finding these patterns in the massive sequence view is not very difficult: we can discern three similar blocks, all of which are followed by fragments of roughly the same length. This leads us to the assumption that the blocks concern the initialization of new drawings, and that the subsequent fragments pertain to the figure insertions. As a means of verification, we zoom in on these patterns

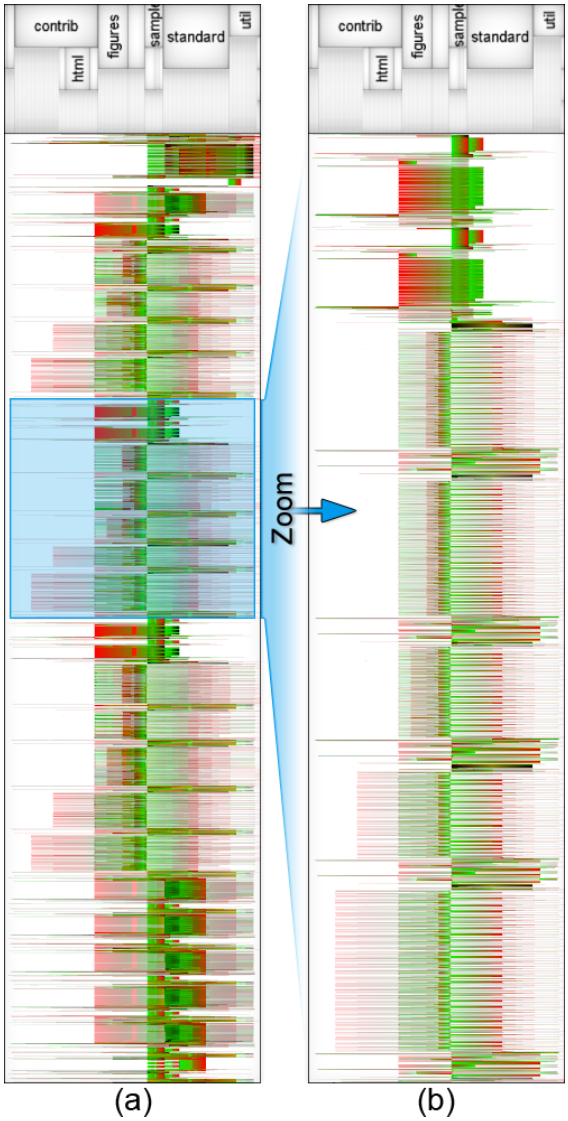
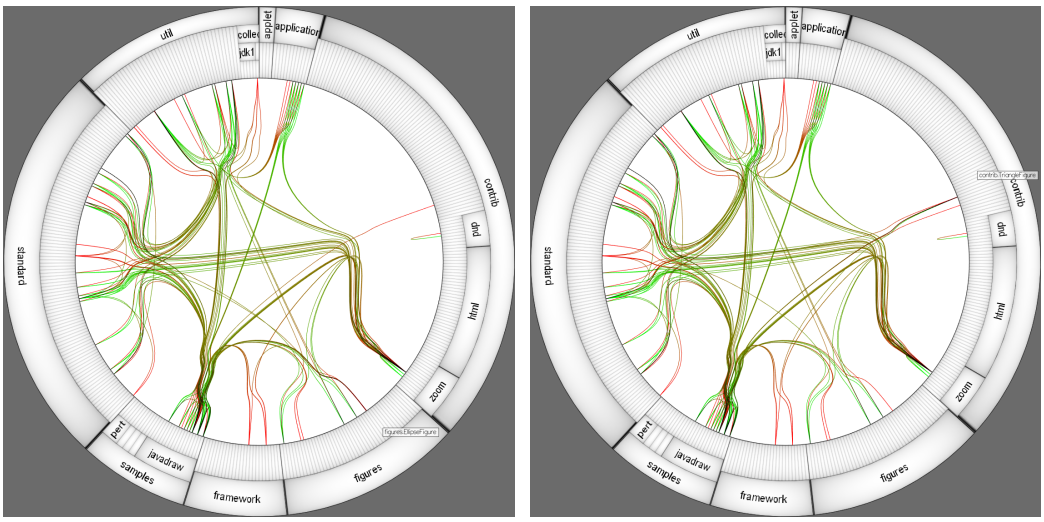


Fig. 8. (a) Full trace of the JHotDraw scenario. (b) Zooming in on the "new drawing" feature and the subsequent figure insertions.

to gather more evidence.

6.4.2 Locating the "insert figure" feature

Figure 8(b) presents a zoomed view of such a fragment, in which we can see the alleged initialization of the drawing in the top fraction. What follows is a series of five patterns that look very similar: a closer look reveals the most prominent difference to be the destinations of certain outgoing calls, which seem to differ especially in the fourth and fifth patterns. These calls are part of an intermitting series and in the latter patterns they are directed toward classes in *different* packages. To determine the identities of these classes, we zoom in on the third and fourth patterns and, in each case, refer to the circular



(a) (b)

Fig. 9. Circular bundle views of two alleged figure drawings, indicating very subtle differences.

view. Comparing the two views points out the differences: at the bottom of Figure 9(a) we observe calls toward three different figure types in the `figures` package (the result of the three figures that are drawn at this point) whereas in the rightmost portion of Figure 9(b) there is also an *additional* call toward `contrib.TriangleFigure`. Repeating this task for the fifth pattern reveals another new figure: `contrib.DiamondFigure`, the fifth figure that was drawn in our scenario. These observations confirm that each of the five fragments concerns a figure insertion.

Taking into consideration that JHotDraw is an open-source project, the division between the figures can be explained as follows: `RectangleFigure`, `RoundRectangleFigure`, and `EllipseFigure` are standard figures in JHotDraw and therefore reside in `figures`, whereas `TriangleFigure` and `DiamondFigure` are in the `contrib` package because they were contributed by third parties.

6.5 Discussion

In this experiment, we have instrumented and executed a medium-sized, GUI-based program according to a scenario that involves certain user-triggerable features. We then visualized the resulting execution trace and pinpointed the locations of these features. The results are promising: choosing an appropriate scenario establishes a strong focus in recognizing the patterns associated with the features, which consequently requires little effort. The feature location process serves as an important first step toward understanding how the

system's features are implemented [34].

Our study has pointed out how the massive sequence view serves as an excellent basis for the (visual) recognition of recurrent patterns. While it does not allow for an easy extraction of more detailed information, selecting a suitable interval requires little effort. Consequently, the circular bundle supports the user in learning more about the events at hand, e.g., by enabling the identification of rather subtle differences in recurrent patterns.

7 Case Study 3: Top-down Program Comprehension with Domain Knowledge

7.1 Motivation

A common situation that developers often find themselves in is when they are not familiar with a specific software system, but that they do have a general knowledge concerning the system's background. Such domain knowledge may have been gained through experience with similar projects in the past. The existence of this up-front domain knowledge means that, a priori, a number of hypotheses about the software system can be formulated and then validated and refined in a top-down fashion [32]. In the process, we form auxiliary hypotheses and receive help from beacons that can direct us. Examples of such beacons are design patterns, and identifiers that have meaningful names. The findings lead to a greater understanding of the system under study, and form a basis for the refinement of the initial hypotheses. The subject system in this experiment is CHECKSTYLE.

7.2 Checkstyle

CHECKSTYLE[5] is an open-source tool that validates Java code. At the basis of this process lies a set of coding standards that can be extended by third parties, and while formerly the focus was on code layout issues, nowadays it also addresses such issues as design problems and bug patterns. The program consists of 23 packages that contain a total of 294 classes and offers both a graphical and a command line user interface. From a user's perspective, its functionality comprises taking a set of coding standards and a Java file as its input, and subsequently presenting a report. Since the system utilizes a batch

[5] Checkstyle 4.3, `http://checkstyle.sourceforge.net/`

execution, unlike in the previous case study we can only control the execution very indirectly.

7.3 Obtaining the trace

The generation of an execution trace is achieved by instrumenting and running CHECKSTYLE from the command line. The input for our scenario consists of an XML file specifying Sun's Java coding conventions [6], and a typical, well-documented Java file that defines one class with 20 methods (300 LOC). Note that while CHECKSTYLE also offers a GUI, in this case study we focus on its command line interface in order to fully concentrate on its core functionality. Executing and tracing this scenario yields an execution trace of nearly 200,000 events.

7.4 Comprehension hypothesis

The main advantage in this case study is the presence of domain knowledge: based on our knowledge of typical source code analysis tools we can speculate about certain properties of CHECKSTYLE. In this experiment, the focus is on its *phases of execution*. In particular, we specify a set of characteristic phases that we expect during the execution of our scenario:

(1) **Initialization.** As with most programs, we expect the initial phase to be concerned with initializations tasks such as command line parsing and input reading.
(2) **AST construction.** Since tools dealing with source code structures typically make use of Abstract Syntax Trees (ASTs), we anticipate that CHECKSTYLE exhibits a similar behavior. The first step in such approaches is the creation of an AST.
(3) **AST traversal.** Once the AST has been generated, the standard procedure for programs in this context is to traverse its nodes and to take action when necessary.
(4) **Report generation.** We expect the final phase to involve the generation of a report and its presentation to the user.

The hypothesis can be considered as a definition of *conceptual* phases. The aim in this experiment is to map these phases onto CHECKSTYLE's actual execution.

[6] Code Conventions for the Java Programming Language, `http://java.sun.com/docs/codeconv/`

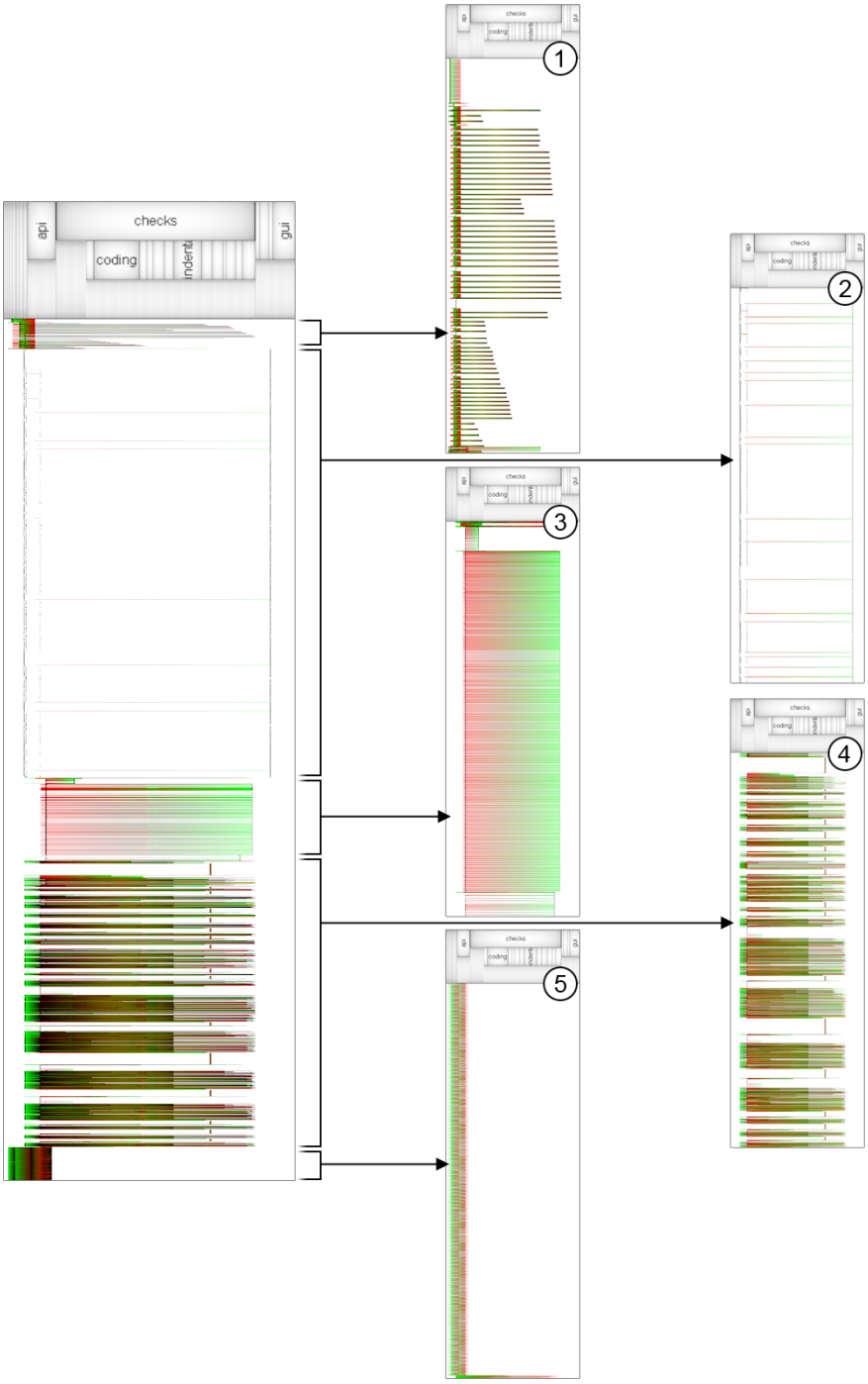


Fig. 10. Massive sequence view of the entire Checkstyle trace (left), and zoomed views of each of its five phases (right).

7.5 Analyzing the trace

The leftmost view in Figure 10 shows the massive sequence view for the full execution trace. Based on the initial view (i.e., at the highest level of granularity) we can roughly discern *five* major phases. For each of these phases, we

report and interpret our findings in the next sections.

7.5.1 First phase

Based on the initial massive sequence view, we choose to zoom in on the first phase (events 1 through 6,400). As it turns out, we are actually dealing with *two* subphases: this is demonstrated by the first of the zoomed views in Figure 10.

First subphase. A quick glance through the first subphase reveals a strong activity among a limited set of objects, being `ConfigLoader` and `DefaultConfiguration`.

Second subphase. The second subphase appears to be more interesting: here we witness an interleaving of sequences of similar calls. The main differences between these sequences are the objects on the receiving end, all of which seem to be located within the `checks` package. This is where the circular view proves helpful: in its temporal mode, browsing through the sequences reveals how all of the checks are processed one by one. This is illustrated in Figure 11, in which the calls are shown in a yellow-to-black (old-to-recent) fashion. Indeed, upon highlighting these calls, the tooltips tell us that the calls pertain to interactions between `DefaultConfiguration` and such checks as `checks.naming.PackageNameCheck` and `checks.header.HeaderCheck`. As a side note, we suspect the reason for the seemingly clockwise trend of the calls in Figure 11 to be the alphabetic order in which the checks are processed, which corresponds to the similarly alphabetically ordered packages and classes in the circular view.

Interpretation. From the many interactions involving configuration classes, we conclude that we are indeed dealing with initialization and configuration tasks. As such, we map CHECKSTYLE's first phase onto the "initialization" phase in our hypothesis.

7.5.2 Second phase

The second major phase that we consider is a series of 91,000 events, of which the largest part displays a very local behavior (second zoomed view in Figure 10). Referring to the circular view in this interval, we learn that most of the activity is concentrated in the `grammars` package, and in `grammars.GeneratedJavaLexer` in particular. To determine what caused this chain of events, we focus on the transition between this phase and the previous phase (Figure 12). Browsing through a limited set of events while scanning for events that have interesting identifier names, leads us to a call that bears

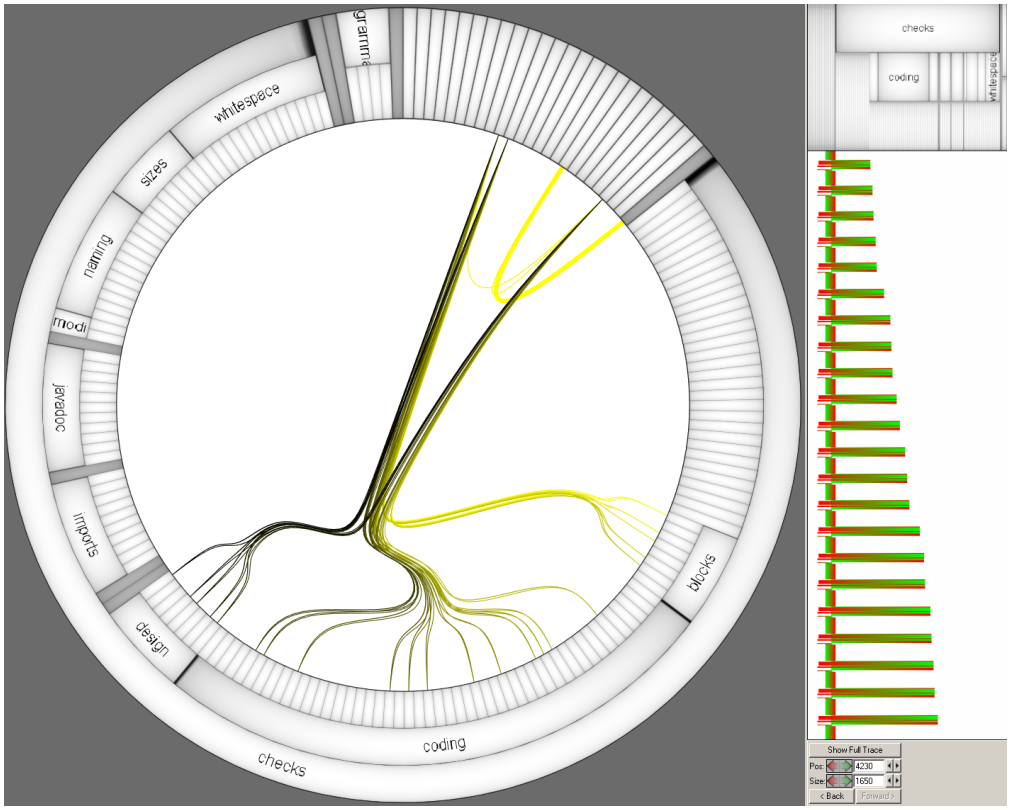


Fig. 11. Full view of a recurrent sequence in Checkstyle's first phase, in which the circular view indicates the receiving classes and the order in which they are processed.

the signature `api.DetailAST TreeWalker.parse(api.FileContents)`.

Interpretation. The signature of the aforementioned call suggests that this phase is concerned with parsing the input file and building an AST. The reason that we are not witnessing the explicit creation of the tree (i.e., node creations) is assumably because we have not instrumented external libraries such as the ANTLR Parser Generator. In conclusion, this phase maps seamlessly onto the "AST construction" phase in our hypothesis.

7.5.3 Third phase

The next phase is a sequence that is characterized by a very consistent shape and coloring, which indicates a great degree of similarity between its 18,000 calls. By examining the earliest calls, we learn that the signatures of the initial calls are `void TreeWalker.walk(api.DetailAST, api.FileContents)` and `void TreeWalker.notifyBegin(api.DetailAST, api.FileContents)`. However, as pointed out by the zoomed view of this third phase in Figure 10, we are actually dealing with roughly *three* subphases: two relatively short parts, and a much longer one in between.

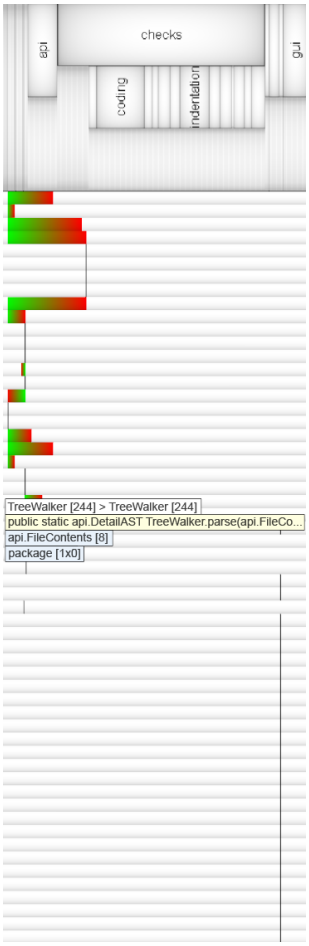


Fig. 12. Focusing on the transition between Checkstyle's first and second phase. The tooltips provide information on the call sites.

First subphase. This subphase consists of 1,300 events and starts off with a series of double calls, in which the `TreeWalker` repeatedly invokes `void api.Check.setFileContents(api.FileContents)` and `void api.Check.beginTree(api.DetailAST)` on a series of different `Check` subclass instances. Most of these calls lead to no further interactions, with the exception of `checks.TodoCommentCheck` (very short, broad "box" at the beginning of the zoomed view) and `checks.GenericIllegalRegexpCheck` (somewhat longer box).

Second subphase. In the second subphase, the receiver of the aforementioned calls is `checks.whitespace.TabCharacterCheck`. The result is the involvement of this class in no less than 15,000 similar interactions, constituting most of the events in this phase. Judging by the method names and the return values that are being passed (e.g., `getLines()` and `getTabWidth()`), we are witnessing the processing of tab characters in the input Java file.

Third subphase. The third subphase is initially similar to the first sub-

phase: more `Check` subclasses receive double calls. Furthermore, in the case of `checks.sizes.LineLengthCheck` the result is a total of 1,100 comparable calls of a line processing nature.

Interpretation. The rather meaningful names of the first few calls in this phase lead to the initial assumption that we are witnessing the traversal of the AST. However, this does not fully explain the peculiar shape of the massive sequence view in this stage. Class highlighting provides the answer as it reveals that `api.DetailAST` is completely absent in the interactions, which means that the aforementioned checks are not reliant on ASTs to fulfill their tasks. This can be explained by the fact that such formatting-oriented checks merely require a *lexical* analysis of the input file. In terms of our hypothesis, we conclude that this phase is *not* covered.

7.5.4 Fourth phase

The fourth zoomed view in Figure 10 shows CHECKSTYLE's fourth major phase, amounting to a total of 78,000 events. The phase is initiated by a call with the signature `boolean TreeWalker.useRecursiveAlgorithm()`, which returns `false`. The next call is `void TreeWalker.processIter(api.DetailAST)`. By highlighting the third event – `void TreeWalker.notifyVisit(api.DetailAST)` – we learn that this particular call frequently occurs throughout the entire phase.

Another observation is the interleaving of two types of activities: a recurrent pattern with interactions spreading across a broad range of classes in the `checks` package, and an intermitting series of 16 vertical "beams". By highlighting the calls that constitute these beams and by referring to the circular view, we learn that these interactions are exclusively concerned with classes in the `checks.javadoc` package.

Interpretation. Judging by the source code associated with `boolean TreeWalker.useRecursiveAlgorithm()`, the decision of the program to use an iterative algorithm rather than a recursive algorithm can be attributed to CHECKSTYLE's configuration during the execution. Furthermore, the frequent calls to `void TreeWalker.notifyVisit(api.DetailAST)` (and the ensuing interactions involving `Check` subclasses) lead to believe that the "Visitor" design pattern [12] is used during the AST traversal. Finally, we discovered that 16 out of 20 method definitions in our input file are preceded by Javadoc entries, which we presume accounts for the 16 Javadoc call sequences that we encountered.

Based on our observations in this phase, we conclude that this phase maps to the "AST traversal" phase in our hypothesis.

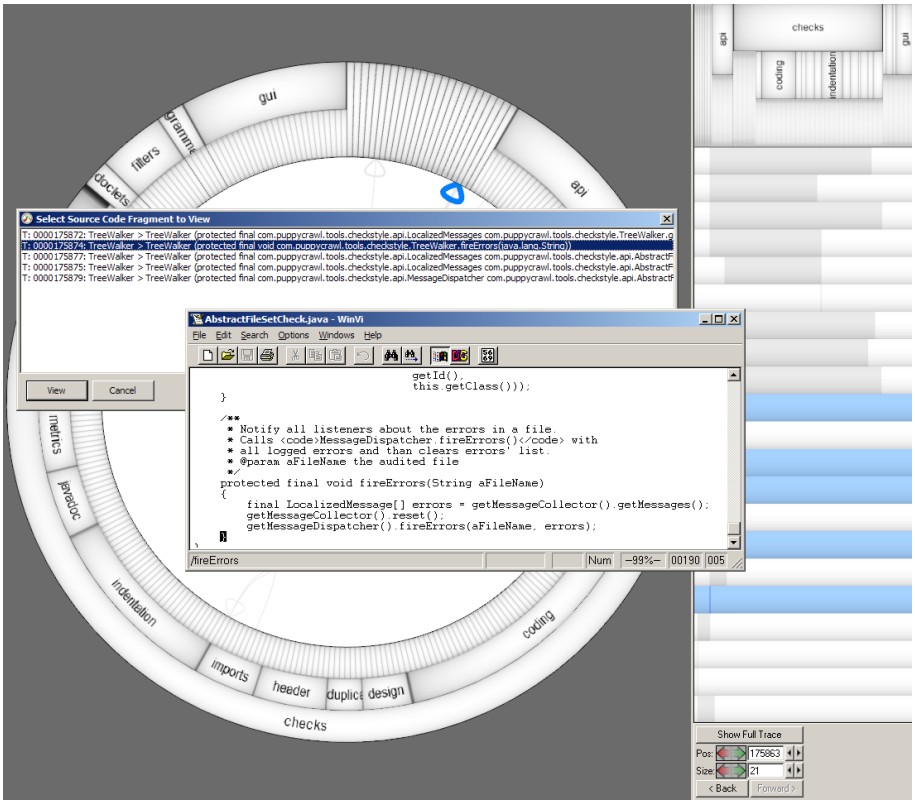


Fig. 13. Looking up associated source code fragments.

7.5.5 Fifth phase

The fifth and final phase as seen in the last zoomed view in Figure 10 contains nearly 15,000 events and actually starts with a main subphase and finishes with a small endphase.

First subphase. The initiating call here is `void TreeWalker.fireErrors(java.lang.String)`, with the actual parameter being "Game.java", our input Java file. Since a brief glance through the ensuing interactions does not prove very meaningful, we double-click on the initial call to view the source code at this call site (Figure 13), in which the Javadoc entry states "Notify all listeners about the errors in a file".

Second subphase. CHECKSTYLE's final phase starts off with `Checker` invoking the `void api.FileSetCheck.destroy()` method on `TreeWalker`. What follows is a series of `api.Check.destroy()` calls toward each of the `Check` subclasses; near the end, the most noteworthy event is `DefaultLogger.closeStreams()` as it precedes the writing of CHECKSTYLE's output to the shell.

Interpretation. The source code associated with

Phase	#Calls	Notable calls	Prominent classes	Description	Maps to
1	6.4K	-	`ConfigurationLoader` `DefaultConfiguration` `checks.*`	Interactions between various configuration classes and checks.	Initialization
2	91K	`TreeWalker.parse()`	`grammars.` `GeneratedJavaLexer` `TreeWalker`	Local activity in the `grammars` package, presumably involving external libraries.	AST construction
3	18K	`TreeWalker.walk()` `TreeWalker.notifyBegin()`	`TreeWalker` `FileContents` 4 `Check`-subclasses	Non-AST related activities involving four specific `check` subclasses.	-
4	60K	`TreeWalker.processIter()` `TreeWalker.notifyVisit()`	`TreeWalker` `api.DetailAST` remaining `Checks`	Interleaving between various `check`-related events and Javadoc processing.	AST traversal
5	8K	`TreeWalker.fireErrors()` `TreeWalker.destroy()`	`TreeWalker` `checks.*`	Processing of errors and termination of the program.	Report generation

Table 1
Results of the Checkstyle experiment.

`void TreeWalker.fireErrors(java.lang.String)` leads us to believe that the first subphase is responsible for presenting the accumulated errors to the user, whereas the second subphase handles the program's termination. This phase maps perfectly onto the "report generation" phase in our hypothesis.

7.6 Discussion

In this case study we have focused on a medium-sized software system that, while being unknown to us in advance, has a functionality that we are familiar with. We have used this domain knowledge to formulate a hypothesis that specifies a set of conceptual phases in a typical execution scenario. Table 1 summarizes the results.

The experiment was quite successful: with the exception of CHECKSTYLE's third phase, each of the five phases that we discerned through the use of EXTRAVIS could be mapped onto a conceptual phase. In particular, the massive sequence view allows to rapidly identify the essential events, i.e., calls that are responsible for phase transitions. The circular bundle view is an aid when a more detailed visualizations of certain call sequences are required, and offers an easy link to the system's source code. While the third phase came unexpected, in retrospect it is perfectly understandable that certain aspects of an input file's source code are treated differently since the associated checkers

have no need of ASTs.

The experiment took only a few hours, and illustrates how domain knowledge and a top-down approach can lead to a significant level of understanding of the system under study. Moreover, additional (sub-)phases can be observed by means of zooming, which in turn can be used to refine initial hypotheses in an iterative fashion (cf. Reflexion models [24]) until a sufficient level of understanding has been obtained for the task at hand.

8 Discussion

The case studies in Sections 5 through 7 have pointed out a series of potential applications of our approach in the context of understanding large execution traces and, by extension, understanding software systems. This section lists a number of important characteristics of our techniques and discusses both the advantages and limitations.

8.1 Advantages

Common trace visualization tools use UML sequence diagrams (or variants thereof) to display a system's structure and the detailed interactions between its components (e.g. De Pauw et al. [7] and Jerding et al. [19]). Although sequence diagrams are very intuitive, they typically become difficult to navigate when the number of components or the time period under consideration become too large: situations where two-dimensional scrolling is necessary to grasp even relatively simple functionalities can rapidly occur, which easily disorients and confuses the user. Extravis, on the other hand, uses a scalable circular view that fits on a single screen. All of the system's components are hierarchically projected on a circle, and entities that are of no immediate interest can be collapsed, which improves readability and ensures that the user is not overwhelmed by too much information.

Moreover, the calling relationships between elements are visualized using bundling, which greatly improves the overall readability in case of many simultaneous relations. Through the use of colors, there is the ability to either (1) show these relationships in a chronological order, or (2) indicate the fan-in and fan-out behavior of the various entities.

The massive sequence view, which provides a concise overview of an entire execution trace, allows the user to easily zoom in on parts of the trace. This reduces the time period under consideration in *both* views and eases the navi-

gation. Another benefit of this view is that it is easy to recognize patterns and phases on the macroscopic level and, by use of zooming, on the fine-grained level as well.

Finally, our techniques are aimed at the optimal use of screen real estate. The observation that a circular representation does not fit on a standard (rectangular) screen is valid; however, it is a matter of positioning the tool controls and settings in the unused space for the screen to be optimally used. Such improvements could be included in future versions of Extravis.

8.2 Limitations

While our techniques effectively visualize large execution traces that are normally too difficult to understand, the size of the input trace is limited in terms of our prototype tool. The reason for this is twofold: not only does Extravis require a substantial amount of *computational* resources – i.e., memory to keep track of all elements and relations, and CPU cycles to perform calculations, counts etc. – but visualizing large systems also requires a considerable amount of *screen real estate*. The latter problem exists because not all events can be visualized in the massive sequence view in a non-ambiguous fashion in case there are more events than there are horizontal pixel lines. It must be noted, however, that Extravis is not necessarily a stand-alone tool; it could well be used as part of a tool chain, e.g., after some abstraction phase.

Moreover, while the circular bundle view is a useful means to display certain characteristics of a program without the need for scrolling, it can be fairly difficult to grasp the temporal aspect. When considering a small time frame (e.g., 50 calls), the circular approach's temporal mode does not make for a visualization that is as readable and intuitive as a sequence diagram, since it requires the interpretation of colors rather than a top-to-bottom reading. In other words, while the display of the system's *entire* structure is useful in such applications as fan-in and fan-out analysis, this information is not always needed.

Furthermore, threads are currently not supported. Although our tracer does register thread information, we do not yet have an effective technique to show the interactions between these threads. In the context of multithreaded systems it may prove useful to effectively visualize these interactions, as threads typically convey important information on the (interleaving of) distinct processes within these systems.

Criterion	Extravis implementation
Overview	Massive sequence view
Zooming	Zooming in the massive sequence view
Filtering	Collapsing of elements
Details-on-demand	Highlighting of elements / relations
Relate	Circular view (with bundling)
History	Forward / back buttons
Extract	Save / load current state

Table 2
Shneiderman's GUI criteria.

8.3 Shneiderman criteria

Shneiderman introduced seven criteria for assessing the graphical user interfaces of information visualizations [30]. Table 2 outlines how the two synchronized views of Extravis satisfy each of these seven criteria.

8.4 Threats to validity

The case studies that we have presented are representative for real-life situations that software developers encounter on a daily basis. The trace exploration, feature location, and top-down analysis scenarios that we used to study and understand the subject software systems are realistic and, as was mentioned in the motivational sections of the studies, occur in various contexts. Nevertheless, there are a number of aspects in which our experiments may differ from real-world situations. We now address those factors that we feel are the most influential.

First, in our experiments we have occasionally relied on identifiers having meaningful names. In Checkstyle, for example, we often used the method's signatures to get an indication of the intended functionality. It must be noted that the presence of meaningful identifier names is by no means a guarantee in everyday software systems.

Secondly, with respect to our feature location study, we have stated that our definition of a feature is a user-triggerable unit of functionality. While this is a common assumption in this problem area (e.g., [11]), our visual form of feature location is more difficult if the features at hand can *not* be invoked directly. When considering Cromod, for example, it is hard to control the execution of its distinct features because it concerns a batch execution based on a set

of complex input files. In other words, the applicability of our techniques in feature location tasks depends on the nature of the system's execution.

Finally, the initial traces in two of the case studies were inexplicably huge. Closer inspection revealed that these traces contained massive amounts of events that can be attributed to non-functional requirements, such as logging (e.g., the CROMOD case) or registering mouse events (e.g., in JHOTDRAW). Assuming that mouse movements and logging are not particularly interesting in grasping a system's general functionality, we carefully filtered out these particular events in a preprocessing step, so as to prevent the traces and the resulting visualizations from becoming unnecessarily complex. It should be noted that this task is rather delicate, and in performing similar experiments one must be careful not to accidentally filter any events that pertain to functionalities that the user considers to be relevant.

9 Related Work

Research into trace visualization has resulted in various techniques and tools over the years. Most related articles are concerned with explaining the visualization tools and techniques by example; in contrast, we have reported on the use of our techniques in several real-world scenarios.

De Pauw et al. [7] are known for their work on IBM's Jinsight, a tool for visually exploring a program's run-time behavior. Many features of this prototype tool have since found their way into Eclipse as plug-ins, more specifically, the *Test & Performance Tools Platform* (TPTP). Though being useful for program comprehension purposes, scalability remains worrisome. To this end, the authors have introduced the *execution pattern notation* [8], which unfolds the graph from a typical sequence diagram (or any variant of a Jacobson interaction diagram [17]) into tree structures. This layout emphasizes the progression of time and not so much the thread of control.

Lange and Nakamura [21] report on Program Explorer, a trace visualization tool targeted at C++ software. Several views are available, of which the *class graph* plays a central role. Through such abstractions as merging, pruning, and slicing, the tool attempts to reduce the search space when studying execution traces; however, the degree of automation of these abstractions is unclear. Furthermore, the tool does not offer a comprehensive view of all the packages and classes that are involved, and selecting a trace interval for detailed viewing does not seem feasible.

Jerding et al. [19] present ISVis, a tool that features two simultaneous views of a trace: a continuous sequence diagram, and a *mural* view that is similar

to our massive sequence view. ISVis' main strength lies in automatic pattern detection, which allows to summarize common execution patterns, and reduces the size of the trace considerably. Our approach differs from ISVis in that the latter deals from the perspective of sequence diagrams (which cannot contain a large number of structural elements), whereas our tool is centered around a scalable circular view.

AVID, a visualization tool by Walker et al. [33,4], aims at exploring a system's behavior by manually defining a high-level model of a system and then enriching it with trace data collected during the system's execution. This is a manual step that involves multiple iterations, thus incrementally improving the user's comprehension of the system. At the basis of this operation lies the Reflexion process [24]. Although there is support for the (sampling-based) selection of a scenario fragment, the tool faces a significant scalability issue as scenarios still induce a potentially large amount of trace data that cannot be directly visualized.

Reiss and Renieris [27] note that execution traces are typically too large to visualize directly and therefore propose to select, compact, and encode the trace data.

Jive, also by Reiss [26], is a Java front-end that visualizes a program's behavior while it is running, rather than analyzing its traces in a post mortem fashion. While the run-time visualization and relatively small overheads render it an attractive tool, it is hard to visualize entire executions. It does, however, provide a view on the classes that are active during a specific phase of the software's execution, and it also allows to perform a rudimentary performance analysis.

Systä et al. [31] present Shimba, an environment that uses sequence diagrams to visualize interactions between user-specified components. Pattern recognition is applied to cope with the scalability problems that are often associated with these diagrams. However, only the interactions between elements that were manually specified by the user are shown, as viewing all components of a large system in a sequence diagram is not feasible due to scalability issues.

Richner and Ducasse [29] propose to use their Collaboration Browser to reconstruct the various object collaborations and roles in software systems. This is achieved by selecting a class and then specifying queries to learn more about the interactions in which this class is involved. Iteratively studying the results of these queries and refining or adding new queries leads to a deeper understanding of the subject software system.

Ducasse et al. [9] use polymetric views to visualize certain metrics that are collected at run-time, resulting in a significant reduction of information. Their approach is mainly aimed at recognizing those entities in a system that are

actively allocating new objects, or that are frequently calling other classes. Although their approach works *offline*, there are similarities with the way in which Reiss projects dynamic metrics in his Jive-tool [26].

Kuhn et al. correlate feature traces with the help of "signals in time": they visualize traces as signals, of which the amplitude is determined on the basis of the stack depth at points during the execution [20]. The idea is that similar traces exhibit comparable sequences of amplitude values, and that these similarities can be visually detected. Their work focuses solely on feature location and not so much on more general program comprehension. Similar work by Zaidman and Demeyer [38] uses the relative frequency of method executions to compare regions in traces, as opposed to using stack depths.

Greevy et al. [13] present a 3D visualization of a software system's execution. The visualization metaphor that they use to display large amounts of dynamic information is that of growing towers, with towers becoming taller as more instances of a type are created. The authors aim to (1) determine which parts of the system are actively involved in a particular (feature) scenario execution, and (2) identify patterns of activity that are shared among different features of the system.

Hamou-Lhadj et al. [14] report on a technique to recover behavioral design models from execution traces. Starting with a complete trace, they determine which classes are utility classes, or classes having a high level of fan-in and low (or non-existent) fan-out. Once these classes are removed from the trace, the resulting trace is visualized in the Use Case Map (UCM) notation. UCMs provide a compact and hierarchical view of the main responsibilities per class combined with architectural components. However, UCMs do not provide a global overview of the application, are not easily navigable, and are more targeted toward understanding very specific parts of a system.

10 Conclusions

Dynamic analysis is generally acknowledged to be a useful means to gain insight about a system's inner workings. A major drawback of dynamic analysis is the huge amounts of trace data that are collected and need to be analyzed. As such, designing an effective trace visualization that (1) is able to cope with these huge amounts of data, and (2) does not confuse the viewer, remains a challenge.

The solution that we propose to tackle this scalability issue is centered around two synchronized views of an execution trace. The first view, which we call the circular bundle view, shows all the system's structural elements (e.g., classes

and packages) and their dynamic calling relationships in a bundled fashion. The second view, the massive sequence view, shows a large-scale message sequence chart that uses a new anti-aliasing technique and that provides an interactive overview of an entire trace. The linking of the two views creates a synergy that ensures the easy navigation and analysis of large execution traces. Our approach is implemented in a publicly available tool called EXTRAVIS.

To illustrate the broad range of potential usage contexts of our approach, we conducted three typical usage scenarios on three different software systems. More specifically, we performed (1) trace exploration, (2) feature location, and (3) top-down program comprehension with domain knowledge. For each of these scenarios, we have presented anecdotal evidence on how our approach helped us to gain different levels of understanding of the software systems under study. Finally, we have reported on the strengths and limitations of our tool, discussed the threats to validity in our case studies, and outlined the added value over related work.

To summarize, our contributions in this paper are:

- A novel approach to visualizing execution traces that employs two synchronized views, namely (1) a circular bundle view for displaying the structural elements and bundling their call relationships, and (2) a massive sequence view that provides an interactive overview.
- The application of our tool, based on this approach, on three distinct software systems in three program comprehension contexts: trace exploration, feature detection, and top-down analysis.

10.1 Future work

There are many potential directions for future work, primarily in terms of improving our techniques and subjecting them to more thorough evaluations.

Among the *improvements* is to facilitate the comparison of execution traces: for example, observing two traces side by side (and thereby detecting correlations) might make feature location considerably easier.

Furthermore, we want to investigate the role of threads in our visualization, and come up with techniques to effectively display both the threads and their interactions.

Future applications include not only the visualization of larger execution traces, but also the detection of outliers. Outlier detection concerns the revelation of call relationships that are not allowed to exist for some reason, e.g., because the elements at hand belong to non-contiguous layers. The circular

view, with its ability to show relations from entire traces in a bundled fashion, provides an excellent basis for the detection of such relationships.

With respect to further *evaluations*, we plan to assess the usefulness of our techniques through empirical studies. Specifically, in the context of a software system with large traces, one could think of a controlled experiment that involves EXTRAVIS, a series of maintenance tasks, and several test users who are not familiar with the system. These tasks would have to be performed by the subjects, part of whom have access to our tool whereas others have not. The results of such an experiment will provide valuable information with respect to the practical applicability of our techniques. Furthermore, we could provide part of the users with access to different tools, such as the ones discussed in Section 9, as this allows to determine the effectiveness of our approach with respect to those of others. Finally, the proposed techniques are subject to evaluations of their own. Such experiments, e.g., an empirical study of importance-based anti-aliasing, could focus on the user's perception of events that are normally difficult to discern.

Acknowledgments

This research is sponsored by NWO via the Jacquard Reconstructor project. We would also like to thank West Consulting [7] for their input concerning the CROMOD case.

References

[1] AspectJ: The AspectJ project at Eclipse.org, `http://www.eclipse.org/aspectj/`.

[2] V. R. Basili. Evolving and packaging reading technologies. *Journal of Systems and Software*, 38(1):3–12, 1997.

[3] L. C. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Trans. Software Eng.*, 32(9):642–663, 2006.

[4] A. Chan, R. Holmes, G. C. Murphy, and A. T. T. Ying. Scaling an object-oriented system execution visualizer through sampling. In *Proc. 11th Int. Workshop on Program Comprehension (IWPC)*, pages 237–244. IEEE, 2003.

[7] `http://www.west.nl/`

[5] B. Cornelissen, A. van Deursen, L. Moonen, and A. Zaidman. Visualizing testsuites to aid in software understanding. In *Proc. 11th European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 213–222. IEEE, 2007.

[6] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proc. 15th Int. Conf. on Program Comprehension (ICPC)*, pages 49–58. IEEE, 2007.

[7] W. De Pauw, R. Helm, D. Kimelman, and J. M. Vlissides. Visualizing the behavior of object-oriented systems. In *Proc. 8th Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 326–337. ACM, 1993.

[8] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proc. 4th USENIX Conf. on Object-Oriented Technologies and Systems (COOTS)*, pages 219–234. USENIX, 1998.

[9] S. Ducasse, M. Lanza, and R. Bertuli. High-level polymetric views of condensed run-time information. In *Proc. 8th European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 309–318. IEEE, 2004.

[10] T. Eisenbarth, R. Koschke, and D. Simon. Feature-driven program understanding using concept analysis of execution traces. In *Proc. 9th Int. Workshop on Program Comprehension (IWPC)*, pages 300–309. IEEE, 2001.

[11] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Trans. Software Eng.*, 29(3):210–224, 2003.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[13] O. Greevy, M. Lanza, and C. Wysseier. Visualizing live software systems in 3D. In *Proc. Symp. on Software Visualization (SOFTVIS)*, pages 47–56. ACM, 2006.

[14] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. C. Lethbridge. Recovering behavioral design models from execution traces. In *Proc. 9th European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 112–121. IEEE, 2005.

[15] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Trans. on Visualization and Computer Graphics*, 12(5):741–748, 2006.

[16] D. Holten, B. Cornelissen, and J. J. van Wijk. Visualizing execution traces using hierarchical edge bundles. In *Proc. 4th Int. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 47–54. IEEE, 2007.

[17] I. Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.

[18] D. F. Jerding and J. T. Stasko. The information mural: A technique for displaying and navigating large information spaces. *IEEE Trans. on Visualization and Computer Graphics*, 4(3):257–271, 1998.

[19] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *Proc. 19th Int. Conf. on Software Engineering (ICSE)*, pages 360–370. ACM, 1997.

[20] A. Kuhn and O. Greevy. Exploiting the analogy between traces and signal processing. In *Proc. 22nd Int. Conf. on Software Maintenance (ICSM)*, pages 320–329. IEEE, 2006.

[21] D. B. Lange and Y. Nakamura. Object-oriented program tracing and visualization. *IEEE Computer*, 30(5):63–70, 1997.

[22] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proc. 28th Int. Conf. on Software Engineering (ICSE)*, pages 492–501. ACM, 2006.

[23] J. I. Maletic, A. Marcus, and M. L. Collard. A task oriented view of software visualization. In *Proc. 1st Int. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 32–40. IEEE, 2002.

[24] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *Proc. 3rd SIGSOFT symposium on Foundations of Software Engineering (FSE)*, pages 18–28. ACM, 1995.

[25] D. Poshyvanyk, Y-.G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Software Eng.*, 33(6):420–432, 2007.

[26] S. P. Reiss. Visualizing Java in action. In *Proc. Symp. on Software Visualization (SOFTVIS)*, pages 57–65. ACM, 2003.

[27] S. P. Reiss and M. Renieris. Encoding program executions. In *Proc. 23rd Int. Conf. on Software Engineering (ICSE)*, pages 221–230. ACM, 2001.

[28] M. Renieris and S. P. Reiss. ALMOST: Exploring program traces. In *Proc. Workshop on New Paradigms in Information Visualization and Manipulation*, pages 70–77. ACM, 1999.

[29] T. Richner and S. Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proc. 18th Int. Conf. on Software Maintenance (ICSM)*, pages 34–43. IEEE, 2002.

[30] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proc. Symp. on Visual Languages (VL)*, pages 336–343. IEEE, 1996.

[31] T. Systä, K. Koskimies, and H. Müller. Shimba - an environment for reverse engineering Java software systems. *Software - Practice and Experience*, 31(4):371–394, 2001.

[32] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55, 1995.

[33] R. J. Walker, G. C. Murphy, B. N. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *Proc. Conf. on Object-Oriented Programming Systems, Languages & Applications (OOPSLA)*, pages 271–283. ACM, 1998.

[34] N. Wilde, J. A. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *Proc. 8th Int. Conf. on Software Maintenance (ICSM)*, pages 200–205. IEEE, 1992.

[35] S. Yang, M. M. Burnett, E. DeKoven, and M. Zloof. Representation design benchmarks: a design-time aid for VPL navigable static representations. *J. Visual Lang. & Computing*, 8(5-6):563–599, 1997.

[36] A. Zaidman. *Scalability Solutions for Program Comprehension through Dynamic Analysis*. PhD thesis, University of Antwerp, 2006.

[37] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *Proc. 9th Conf. on Software Maintenance and Reengineering (CSMR)*, pages 134–142. IEEE, 2005.

[38] A. Zaidman and S. Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *Proc. 8th European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 329–338. IEEE, 2004.