Open access • Proceedings Article • DOI:10.1109/ICPC.2007.39

Understanding Execution Traces Using Massive Sequence and Circular Bundle Views

B. Cornelissen, Danny Holten, Andy Zaidman, Leon Moonen ...+2 more authors

Institutions: Delft University of Technology

Published on: 26 Jun 2007 - International Conference on Program Comprehension

Topics: Software visualization, Program comprehension, Software system, Software development and Software analytics

Understanding Execution Traces Using Massive Sequence and Circular Bundle Views

Bas Cornelissen[1], Danny Holten[2], Andy Zaidman[1],
Leon Moonen[1], Jarke J. van Wijk[2], and Arie van Deursen[3]

[1]*Delft University of Technology – {s.g.m.cornelissen, a.e.zaidman}@tudelft.nl, leon.moonen@computer.org*
[2]*Eindhoven University of Technology – d.h.r.holten@tue.nl, vanwijk@win.tue.nl*
[3]*Delft University of Technology & CWI – arie.vandeursen@tudelft.nl*

Abstract

The use of dynamic information to aid in software understanding is a common practice nowadays. One of the many approaches concerns the comprehension of execution traces. A major issue in this context is scalability: due to the vast amounts of information, it is a very difficult task to successfully find your way through such traces without getting lost. In this paper, we propose the use of a novel trace visualization method based on a massive sequence and circular bundle view, constructed with scalability in mind. By means of three usage scenarios that were conducted on three different software systems, we show how our approach, implemented in a tool called EXTRAVIS*, is applicable to the areas of trace exploration, feature location, and feature comprehension.*

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 [22, 14].

Understanding a system's behavior implies studying existing code, documentation, and other design artifacts in order to gain a level of understanding of the software system that is sufficient for conducting a given maintenance task. This *program understanding* or *program comprehension* process is known to be very time-consuming, and Corbi reports [3] that up to 50% of the time allocated for a maintenance task is spent on gaining knowledge of the software system at hand. 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, or the analysis of data gathered from a running program, has the potential to provide an accurate picture of a software system, among others because it can reveal object identities and occurrences of late binding. However, dynamic approaches are often characterized by enormous amounts of data, which gives rise to scalability issues [27]. Particularly, execution traces from sizeable programs are not easily understood because the efficient visualization of both the structures and the many interrelationships is far from trivial.

In this paper, we present a novel visualization method that allows the visualization of dynamically gathered data from a software system in a condensed way, while still being highly scalable and interactive. We attempt to achieve these goals by presenting two synergistic views: (1) a *circular bundle view* that projects the system's structure in terms of hierarchical elements (and their call relationships) on a circle, and (2) a *massive sequence view* that provides a global overview of the trace. These techniques are implemented in our tool EXTRAVIS (EXecution TRAce VISualizer) that is publicly available for download.[1]

To characterize our approach, we use the framework introduced by Maletic et al. [15]:

1. *Task: Why is the visualization needed?* The amount of trace data that often results from dynamic analysis, calls for an effective visualization. More specifically, we describe how EXTRAVIS is useful for:
   - trace exploration and phase detection,
   - feature location, and
   - feature comprehension.
2. *Audience: Who will use the visualization?* The target audience consists of software developers and reengineers who are faced with understanding (part of) a complex 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

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

15th IEEE International Conference on Program Comprehension (ICPC'07)
0-7695-2860-0/07 $20.00 © 2007 IEEE

these interactions take place. 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.

To assess the usefulness of our approach in the aforementioned contexts, we use the tool to conduct three extensive case studies on an academic, an open source and an industrial software system.

Structure of the paper In Section 2 we provide a detailed description of our visualization approach and tool, along with the requirements. We then present the case studies in Sections 3 through 6. Next, we discuss the advantages and limitations of our approach in Section 7. We cover related work in Section 8, and we summarize our main contributions and future work in Section 9.

2. EXTRAVIS

The goal of EXTRAVIS is to visualize execution traces in order to support program comprehension during various software maintenance tasks. Given an execution trace (or a part thereof), EXTRAVIS presents two synchronized views (see Figure 2 for a screenshot)[2]:

- a *circular view* that shows the system's structural decomposition and the nature of its interactions during a (part of the) trace;
- a *massive sequence view* that provides a concise and navigable overview of the consecutive calls between the system's elements in a chronological order.

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 describe the meta-model used by EXTRAVIS and present the two views that it is based on. The ensuing sections discuss the use of our tool, and illustrate how the combined views can help in conducting various program comprehension tasks.

2.1. Meta-model

The tool 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 information. Input for the tool is provided in the form of RSF files [26].

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

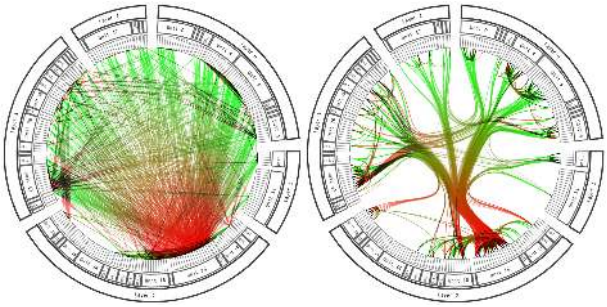

Figure 1. Call relations within a program shown using linear edges (left) and using hierarchical edge bundles (right).

Structural information To visualize the structure of a program, the tool requires a *containment* relation that define the system's structural decomposition, e.g., in terms of package structures or architectural layers.

Basic call relations The second mandatory part of input is a series of *call* relations, which are extracted from an execution trace. The RSF file 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 Optionally, more detailed information on the calling relationships can be added by means of a third input file. This extra data is linked to the basic input on the basis of the aforementioned increment, and deals with object identifiers, runtime parameters and actual return values.

2.2. Circular Bundle View

The circular bundle view offers a detailed visualization of the system's structural entities and their interrelationships. As shown in Figure 1, these relations are depicted by *bundled splines*. Visually bundling relations together helps to reduce visual clutter, and also shows the implicit call relations between parent elements resulting from explicit calls between their respective children. These bundles, called hierarchical edge bundles, were presented by Holten in [9].

The hierarchical elements can be *collapsed* to enable the user to focus on specific parts of the system. Collapsing an element hides all of its child elements and "lifts" the relations pertaining to these child elements to the parent element, providing a straightforward abstraction mechanism. The (un-)collapsing process is fully animated for the user to

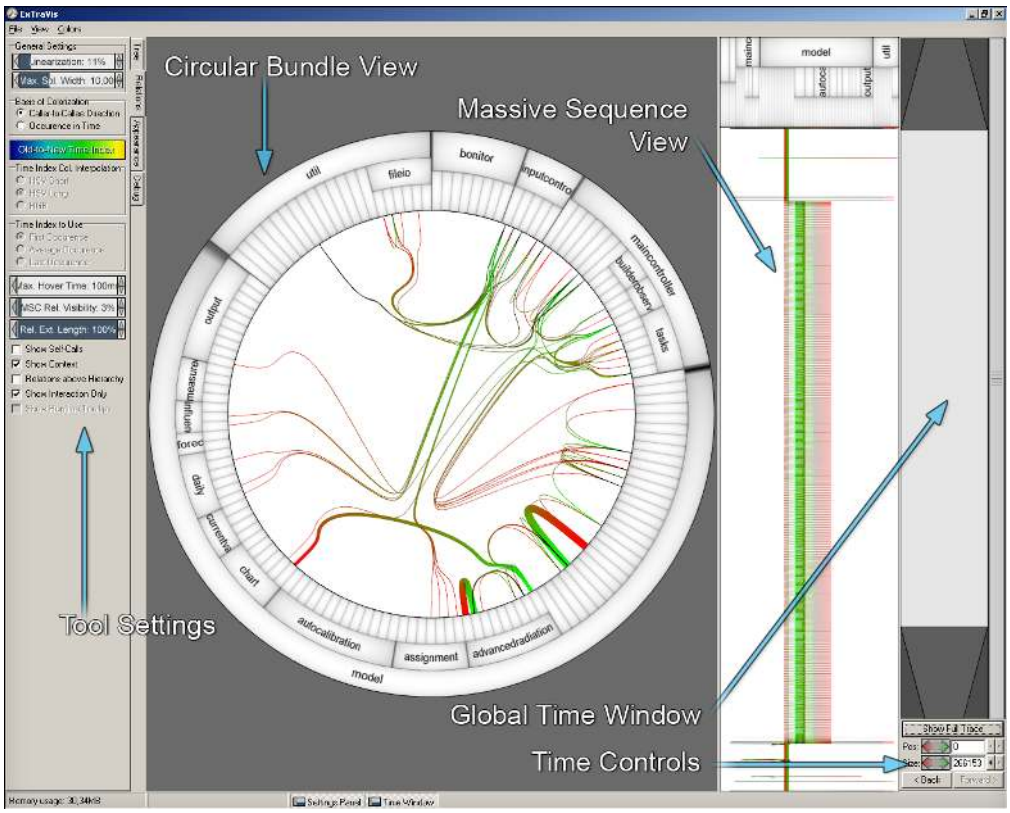

Figure 2. Full view of an entire Cromod trace.

maintain a coherent view of the system, i.e., to facilitate the cognitive linking of the "pre" and "post" view.

Furthermore, textual information related to the underlying source code is provided by means of call highlighting and by providing direct links to the relevant source parts.

In EXTRAVIS, we use hierarchical edge bundles to visualize the dependencies that occur during a selected part of the execution trace. Thus, the circular view provides a snapshot in time that corresponds to the part of the execution trace that is currently being viewed. Splines are drawn between hierarchical elements that communicate with each other. Given (part of) a trace, the *thickness* of a spline indicates the number of calls between two elements.

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 spline indicates its direction. The latter mode colors the relations such that the relations are ordered from least recent (dark) to most recent (light).

2.3. Massive Sequence View

To support users in identifying parts of the trace of interest, EXTRAVIS offers the *massive sequence view*. It provides an overview of (part of) the full execution trace. At the top, the system's structure is shown along the horizontal axis; underneath, there is the visualization of the call relations, which are ordered along the vertical time axis from least recent (top) to most recent (bottom). Again, the directions of the relations are color coded using a gradient (see Figure 2). Additionally, the massive sequence view allows to *zoom in* on parts of the execution trace by allowing the selection of a fragment that needs closer inspection.

The massive sequence view expands upon the concept of the "execution mural" by Jerding et al. [10] in the sense that interaction patterns can be used as abstractions: connecting low-level implementations to higher level design models is potentially useful in program understanding tasks. We provide the user with an additional abstraction mechanism: rather than merely using the visual appearance of patterns, the circular view's collapse mechanism and the lifting of relations results in new, higher level relations which, in turn, correspond to a higher level behavior of the system.

Apart from the coloring aspect, our technique also differs from Jedring's work [10] in that the massive sequence view displays a system's entire structural hierarchy.

3. Case Studies

To illustrate the effectiveness of our techniques, we discuss three specific usage scenarios:

Exploration (Section 4)

- *Context:* System is largely unknown, trace is available. No (or little) up-front knowledge is present.
- *Goal:* Get an initial feeling of how the system works.

Feature location (Section 5)

- *Context:* Features of the software system are known. One or more feature traces, which are manufactured traces that exercise one or more features, are available.
- *Goal:* Knowing which (end user) features are available in a system, the software engineer tries to detect them, i.e., locate their occurrences in the execution trace.

Feature comprehension (Section 6)

- *Context:* A particular feature of the software system has been isolated.
- *Goal:* Understanding how the feature is implemented.

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

3.1. Preparatory Steps

Before we can start to visualize execution traces, we need to generate the necessary input data. We briefly describe the steps needed to collect the inputs.

We make use of a simple Perl tool that derives a system's class decomposition from its directory structure. This results in a parent-child relation that defines the system's structure in terms of classes and (sub-)packages.

As for the dynamic part, we trace a system's execution by monitoring for method invocations and registering the objects that are involved. We achieve this by extending the SDR framework from our earlier work [4]. The associated tracer registers unique objects, method names, information on the call sites (i.e., source filenames and line numbers), runtime parameters and actual return values, and the listener converts these events to RSF.

4. Exploratory Comprehension

Motivation When a system is largely unknown and an execution trace is available, being able to understand the control flow in the trace can be of great help in understanding the system. However, it is a well-known phenomenon that dynamic analysis tends to result in large amounts of data. Due to this "overload", the *exploration* of such traces is by no means a trivial task. To illustrate how we can tackle this issue, we explore an industrial system called CROMOD.

Exercised features For the purpose of exploratory program understanding, we expect to need the following set of features that are incorporated in EXTRAVIS:

- The massive sequence view enables us to visually spot *phases* in the software's execution (similar to Reiss [19]).
- Packages can be *collapsed* to make both the circular and the massive sequence view less densely populated.
- Whereas certain visualizations (e.g., sequence diagrams) would necessitate two-dimensional scrolling, the circular view that we use presents *all* of the current interactions in one concise view.

4.1. Cromod

CROMOD is an industrial Java system that regulates the environmental conditions in greenhouses. The system is built up from 145 classes that are distributed 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 are controlled and, finally, writes its output. The model calculations typically induce massive amounts of interactions, which makes this system an interesting subject for trace visualization.

Setup The trace that results from a typical CROMOD execution contains millions of events, of which a large part can be attributed to logging. For this reason, we have run the program at a log level such that the resulting trace contains roughly 270,000 method and constructor calls, of which the comprehension is still quite a challenge. The trace (100MB) was converted to RSF, and then extended with information on the system's hierarchical decomposition in terms of its package structure.

4.2. Typical Usage Scenario

Loading the trace into EXTRAVIS provides us with the initial view that is shown in Figure 2.

Detecting the major phases The massive sequence view indicates that there are three major "phases" in this execution scenario, characterized by two small beams (first and third phase) and a long segment that appears to be somewhat broader (second phase). At this point in time, we formed our initial hypothesis that these stages concern (1) an input phase, (2) a calculation phase, and (3) an output phase. This proved to be correct upon further examination.

Focusing on the first phase The first phase that we visually discern in the massive sequence view looks like an almost straight vertical "beam". We zoom in on this phase by selecting an interval, and thus reducing the timeframe under consideration. Now, EXTRAVIS only visualizes the interactions within the chosen timeframe. Turning our attention to the circular view (Figure 3), we learn that this

[3]Note that although this experiment involves Java because our toolchain is optimized for Java systems, we have no reason to believe that our technique is not applicable to other (non-object oriented) languages.

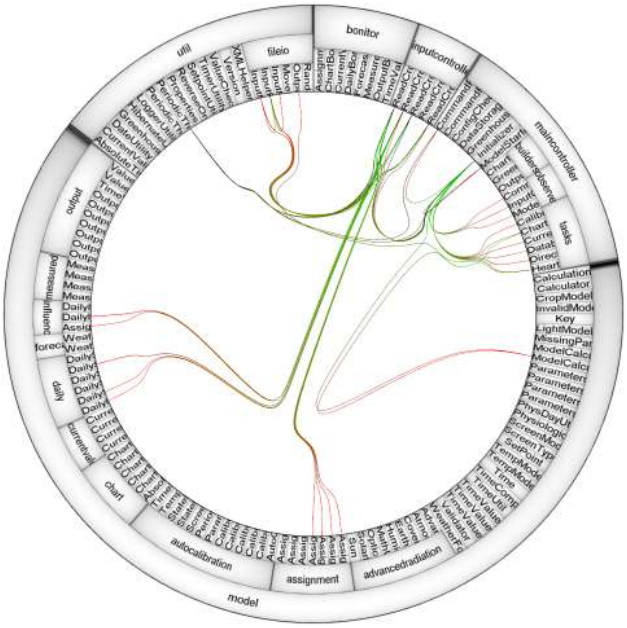

Figure 3. Circular view of Cromod's initialization phase.

first phase merely involves a limited number of classes and packages, of which most pertain to I/O operations.

Understanding what is happening in a phase Double clicking on packages collapses them, which renders the circular view less cluttered and makes the interactions clearer. In this phase it proves useful to collapse the "model" package, which is a relatively large package that is seldomly used at this stage. By means of the edge colors we observe that certain (groups of) classes have high fan-in and fan-out rates and, with respect to the chosen timeframe, the thickness of an edge indicates the number of calls that occurred between the associated elements.

We followed a similar strategy for the second and third phase, the latter being very similar in terms of I/O activity. The second phase was characterized by a number of repeating sub-phases (mainly within the "model" package) that involves the many interactions that make up a model calculation; in particular, it turned out the creation and processing of massive amounts of `Time` objects accounts for the majority of the interactions.

The main lesson learned from this case study is that the identification of phases can help to quickly outline a system's general functionality.

5. Feature Location

Motivation As was briefly mentioned in the introduction, a significant portion of the effort in a maintenance task is spent on determining *where* to start looking within the system, and which parts to focus on. As such, we consider *feature location* [6] to be an important use case, and show how to use our tool to localize features in JHOTDRAW.

Exercised features In addition to the features that were used in the previous section, this case reveals two more:

- The massive sequence view, with its zooming capability, is not merely suitable for phase detection but also for (visually) recognizing patterns, which is a first step towards the location of features.
- In case the zooming process was not satisfactory, it is a matter of pressing the "back" button to return to the original view and redefine a zoom window.

5.1. JHotDraw

JHOTDRAW[4] is a well-known, highly customizable Java framework for graphics editing. It was developed as a "design exercise" and is considered to be well-designed. It comprises roughly 300 classes and 20 packages. Running the program presents the user with a GUI, in which he or she can create drawings that may contain manual sketches, text, predefined figures and such.

Setup To generate a suitable feature trace, we have constructed a user scenario that involves several major features that we hope 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 localization of the "new drawing" and its "insert figure" features easier, we invoked the aforementioned scenario a total of *three* times. However, 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.

5.2. Typical Usage Scenario

Figure 4(a) shows the massive sequence view of the entire execution trace, in which we can immediately observe several recurrent patterns.

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 hypothesis that the blocks concern the initialization of new drawings, and that the subsequent fragments pertain to the figure insertions.

Locating the "insert figure" feature To verify our hypothesis, we take a closer look at the patterns mentioned

[4] `http://www.jhotdraw.org/`

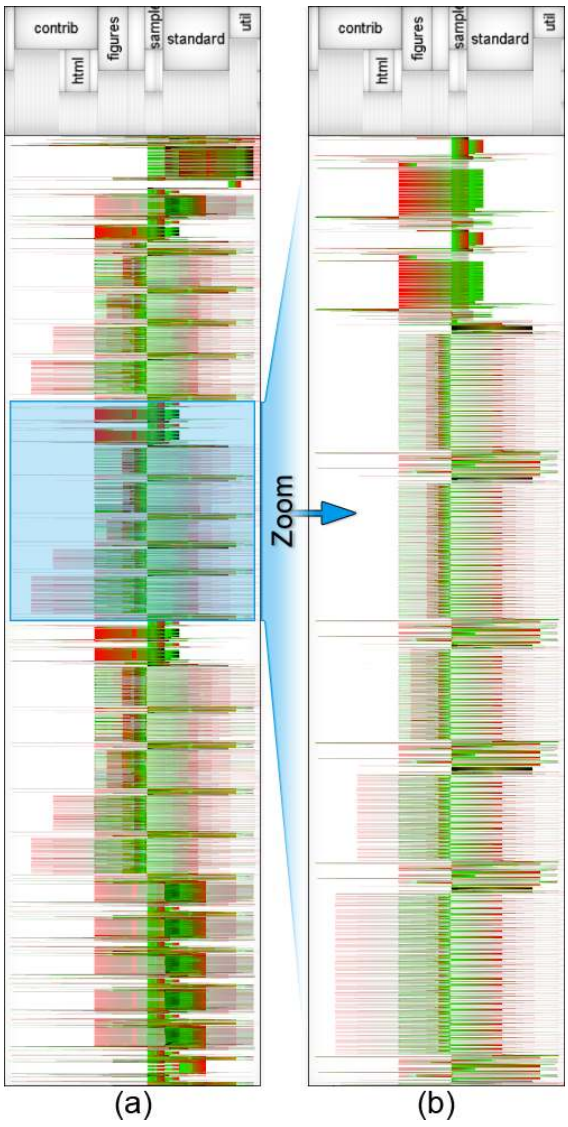

Figure 4. (a) Full trace of the JHotDraw scenario. (b) Zooming in on the "new drawing" feature and the subsequent figure insertions.

above. Figure 4(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 patterns, of which five are very similar. Indeed, these patterns must relate to the figure insertions, as in each pattern there is a fair amount of outgoing calls towards either the "*figures*" package (first three figures) or the "*contrib*" package (last two figures). Upon closer inspection of these packages, our assumption turns out to be correct: `RectangleFigure`, `RoundRectangleFigure`, and `EllipseFigure` are standard figures in JHOTDRAW, whereas `TriangleFigure` and `DiamondFigure` are in the "contrib" package because they were contributed by third parties.

As soon as the feature has been isolated, we can attempt to understand the interactions involved in its implementation. We focus on this activity in the next section.

6. Feature Comprehension

Motivation Once a feature has been located, i.e., when the timeframe of interest has been found, the next step is to understand the feature. *Feature comprehension* is thus concerned with understanding the interactions that take place during a feature invocation: gaining knowledge of a feature's implementation is an important step towards easing maintenance tasks such as change requests.

Exercised features Our tool offers several functionalities to help gain a detailed understanding of trace fragments:

- Highlighting the occurrences of hierarchical elements in the circular view indicates *where* these elements actively participate within the chosen timeframe. This enables the viewer to quickly focus on these locations.
- Once a suitably small timeframe is chosen, switching the display mode from "basic" to "runtime" presents *runtime information* such as objects and actual parameters, rather than classes and formal signatures.
- The user can request a view of the specific parts of the *source code* with a simple right mouse click.
- Switching the circular view into *temporal* mode allows for an accurate view of the chronological ordering of the interactions, and offers a means to systematically step through the execution.

6.1. JPacman

To assess the usefulness of our approach in gaining (detailed) knowledge of features, we have conducted a study on JPACMAN, an academic teaching example at Delft University of Technology. It is a fairly simple Java application that consists of 20 classes and 3,000 lines of code. It is an implementation of the well-known Pacman game, in which there exists a board of 20 by 20 cells containing food items, monsters, and walls. Among the many features in this context are: moving, eating food, bumping into a wall, colliding with a monster, and restarting the game.

Although JPACMAN is a small system, its execution traces typically contain hundreds of thousands of calls, which renders it an appropriate case study for any trace visualization technique.

Setup We obtained a suitable execution trace for JPACMAN by playing a short game and thereby invoking a set of features. These features include (in order of invocation): starting a game, eating food, dying, restarting, and quitting. The trace that resulted from this scenario contains a little over 150,000 calls, of which a great deal is made up of

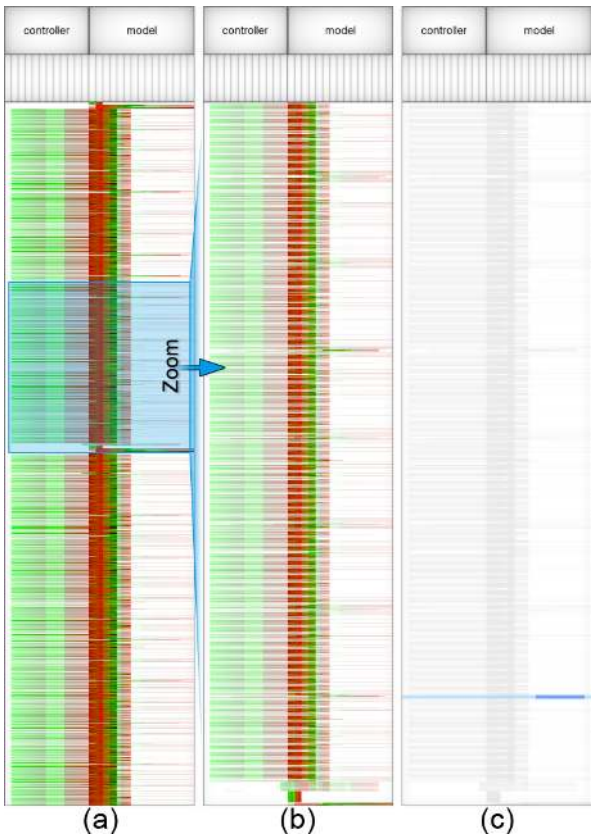

Figure 5. (a) Full trace of the JPacman scenario. (b) Zooming in on the "player dies" feature. (c) Highlighting an interaction.

"noise" because of the many player and monster movements that take place in between the intended feature invocations.

6.2. Typical Usage Scenario

The massive sequence view of the entire JPACMAN trace is displayed in Figure 5(a). To proceed with the inspection of a feature, we must first locate it and then zoom in on it.

Locating the "player dies" feature Our aim is to learn about the feature that is the death of the player. Since in our execution scenario the player's death took place just before the game was restarted, which we expect to be a relatively complex feature, we want to zoom in on a fragment of the execution that precedes the restart phase. The overview allows for the easy detection of this phase: at roughly halfway through the trace there exists a fragment that bears a striking resemblance to the initialization phase, which leads us to believe that this is where the game is restarted. Thus, in the massive sequence view, we select a large interval preceding the restart pattern (Figure 5(a)).

The new view (Figure 5(b)) shows the restart phase at the bottom; however, it is still difficult to spot the collision between the player and a monster, and we do not know on which part to zoom in next. To further narrow down the search space, we have two options.

Narrowing down the timeframe The first option is to consider the circular view and to highlight the `PlayerMove` and `Monster` *classes*. In the massive sequence view, a blue horizontal line (within a grayed-out context) indicates the mutual interactions of these elements within the chosen interval. However, in case of a large system that consists of many (unknown) classes, it might prove difficult to determine which classes to highlight.

Alternatively, we can look at the circular view and, more specifically, inspect the *interactions* pertaining to the `Player` and `PlayerMove` classes. We choose these classes because their names suggest that they are likely to be part of any player movement. We can immediately see a relation between the latter class and a `Monster`, and choose to highlight it. As it turns out, this relation concerns two calls in opposite directions, of which the signatures are `Guest.meetPlayer()` and `PlayerMove.die()`, which indicates that we are on the right track.

Again, by means of a blue line in the massive sequence view, we can see when these interactions take place: Figure 5(c) reveals the collision's exact location. We can now zoom in even further, and obtain a compact visualization of the player-monster collision feature.

Inspecting the feature We switch the circular view to temporal mode, and can see the chronological order of the interactions that took place during this feature. This view allows us to get a quick, yet reasonably complete view on what happens at this stage. For more details we can switch to runtime mode and gradually step through the execution. This way, we observe that the caller of `meetPlayer()` is in fact an object that is a subclass of `Guest`, namely *Monster450* (Figure 6). This turns out to be a double dispatch [1]: the effect of a call to the generic `meetPlayer()` is dependent on the dynamic type of the caller, and in the case of a monster it results in the player's death. The use of this mechanism also sheds light upon the implementation of other types of collisions (e.g., the "eat food" feature), which can be studied in a similar fashion.

7. Discussion

The case studies in Sections 3 through 6 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.

Advantages Most trace visualization tools use UML sequence diagrams (or variants thereof) to display a system's

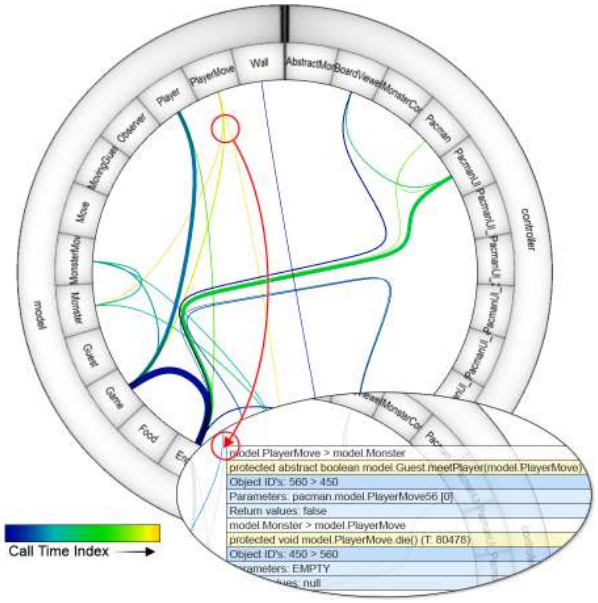

Figure 6. Circular, temporal view of the "player dies" feature that shows runtime information.

structure and the detailed interactions between its components (e.g. De Pauw [17] and Jerding [11]). Although sequence diagrams are very intuitive, they typically become difficult to navigate when the number of components and the time period under consideration become too large: situations where two-dimensional scrolling is necessary to grasp even relatively simple interactions can quickly occur, which easily disorients and confuses the user. EXTRAVIS, on the other hand, uses a scalable circular view that fits on one screen. All of the system's components are hierarchically projected on a circle, and components 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. By 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 components.

Finally, 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 navigation. 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.

Limitations While EXTRAVIS successfully visualizes large execution traces that are normally too difficult to understand, the size of the input trace is still limited. The reason for this is twofold: not only does the tool 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.

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 1. Shneiderman's GUI criteria.

Moreover, while the circular view is a good means to display a series of call relations without the need for scrolling, it can be fairly difficult to grasp the temporal aspect within a time fragment. This is especially the case when considering a rather large amount of calls, as it becomes difficult to discern the various colors and, thus, their chronological order. Our tool's ability to "step" through the fragment, however, partly solves this issue.

Furthermore, threads are currently not supported. Although our tracer does register thread information, the visualization tool is not yet equipped with a means to effectively visualize the interactions between these threads.

Shneiderman Shneiderman introduced seven criteria for assessing the graphical user interfaces of information visualizations [23]. Table 1 outlines how the two synchronized views of EXTRAVIS satisfy each of these seven criteria.

8. 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, whereas in Sections 3 through 6 we have reported on a range of both general and very specific applications of our tool. This section provides a brief overview of the related work in this area.

De Pauw et al. [17] are known for their work on IBM's Jinsight, a tool for visually exploring a program's runtime 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* [18].

Lange et al. [13] report on Program Explorer which, given an execution trace, visualizes a program's interaction graph. This graph can then be studied, and there is support for several filtering techniques to reduce its size. 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 is not very feasible.

Jerding et al. [11] present ISVis, a tool that features two simultaneous views of a trace: a continuous sequence diagram, and a *mural* view that is somewhat similar to our massive sequence view [10]. ISVis' main strength lies in 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 can not contain a large number of structural elements), whereas our tool is centered around a scalable circular view.

AVID [25, 2], a visualization tool by Walker et al., aims at depicting a system's dynamic behavior by having the user define an architecture and then enriching it with runtime information. A form of Reflexion [16] lies at the basis of this process. 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 [21] 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 [20], is a Java front end that visualizes a program's behavior while it is running, rather than analyzing its traces in a postmortem fashion. While the runtime visualization and relatively small overheads render it an attractive tool, it is hard to visualize entire executions.

Greevy et al. [7] present a 3D visualization of the execution of a software system. 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.

Another set of trace visualizations with a variety of purposes can be found in the work of Ducasse et al. [5], who use so-called polymetric views to visualize dynamically collected metrics. Kuhn et al. [12] exploit the correlation between execution traces and signals in time, an approach similar to the one by Zaidman and Demeyer [28]. Hamou-Lhadj et al. [8] apply "use case maps" to visualize behavioral models in a compact fashion. Systä et al. [24] use a variant of sequence diagrams to visualize trace information.

9. Conclusions

Dynamic analysis is generally acknowledged to be a useful means to gain insight about a system's inner workings. The 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 view, shows all the system's hierarchical 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 sequence diagram that provides an interactive overview of an entire trace. The combination of the two views creates a synergy that ensures the easy navigation and study 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 usage scenarios on three different software systems. More specifically, we performed (1) trace exploration, (2) feature location, and (3) feature comprehension. For each of these scenarios, we have presented anecdotal evidence on how our approach helped us gain insight into the software systems under study. Finally, we have reported on the strengths and limitations of our tool and discussed its 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, in three reverse engineering contexts on three distinct software systems: exploratory program comprehension, feature detection, and feature comprehension.

Future work There are many potential directions for future work, both in terms of improving our techniques and in applying them to alternate usage scenarios.

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 much *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 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.

Finally, we want to assess the usefulness of our techniques by conducting an emperical study. For instance, in the context of a large software system, one could think of an experiment that involves EXTRAVIS, a questionnaire, and several test users who are not familiar with the system.

Acknowledgments

This research is sponsored by NWO via the Jacquard Reconstructor project. We would also like to thank West Consulting[5] for their input concerning the CROMOD case.

References

[1] K. Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1996.

[2] 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.

[3] T.A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[4] 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.

[5] 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.

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

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

[8] A. Hamou-Lhadj and T.C. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *Proc. 14th Int. Conf. on Program Comprehension (ICPC)*, pages 181–190. IEEE, 2006.

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

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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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, 1998.

[19] S. P. Reiss. Dynamic detection and visualization of software phases. In *Proc. 3rd ICSE Int. Workshop on Dynamic analysis (WODA)*, pages 1–6, 2005. ACM SIGSOFT Sw. Eng. Notes 30(4).

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

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

[22] 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.

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

[24] 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.

[25] 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.

[26] K. Wong. The Rigi user's manual - version 5.4.4. http://ftp.rigi.csc.uvic.ca/pub/rigi/doc/rigi-5.4.4-manual.pdf (last visited January 30th, 2007), 1998.

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

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

[5] http://www.west.nl/