Software Visualization Techniques for the Representation and Exploration of Execution Traces with a Focus on Program Comprehension Tasks

Heidar Pirzadeh and Abdelwahab Hamou-Lhadj

The Software Behaviour Analysis Lab

Department of Electrical and Computer Engineering
Concordia University Montréal, Québec, Canada

{s_pirzad, abdelw}@ece.concordia.ca

Abstract

The analysis of execution traces can reveal important information about the behaviour of software. This information can in turn be used to help with a variety of software engineering applications including software maintenance, performance analysis, and software security. Traces, however, tend to be extremely large. Various visualization techniques have been proposed to help software engineers browse large traces in an effective manner. These techniques include the use of graphs, UML-like diagrams, metaphors, space-filling methods, and many more. Also, existing tools employ several interaction features. However, it is still not clear how these features and the visualization techniques compare to each other, what their advantages and limitations are, and how they can be better characterized to enable their reuse, and to prevent reinventing what already exists. The objective of this paper is to answer these questions through the study of the visualization and interaction features supported by over 26 trace analysis tools. We also contrast the proposed techniques with traditional literature on software visualization to better understand their effectiveness and possible shortcomings. We finally hope that the discussions presented in this paper can set the ground for a stronger and perhaps more theoretical field of trace visualization and comprehension.

Keywords: Software Visualization, Trace Analysis and Visualization, Dynamic Analysis, Software Engineering

1. Introduction

The analysis of software systems is an important activity in a variety of software engineering fields including software maintenance, software performance, software security, and any other area where some understanding of the system is required before applying the task at hand. This is particularly important for those systems that suffer from poor documentation (if it exists at all), and for which key
developers, knowledgeable of the system’s design, have moved to new projects or companies.

Software analysis typically involves two main techniques: Static analysis and dynamic analysis. Static analysis focuses on exploring the structure of the system by analysing its source code while dynamic analysis, the focus of this paper, aims to provide insight into the system’s behavioural properties. Both approaches, however, aim at helping software engineers understand the system’s components and their relations from different point of views. They can be used independently but can also be combined, and in which case, they lead to best results. Dynamic analysis techniques rely extensively on the study of execution traces generated from executing the system under study. As such, traces can be collected to contain only the information needed to understand a particular aspect of the system. In addition, system execution can be driven by specific input data which provides a powerful mechanism for relating program inputs, outputs, and behaviour [Bal99].

Traces, however, have been known to be difficult to work with due to the excessive amount of information they contain. Executing even a relatively medium-sized system can result in traces containing millions of events. Existing trace analysis tools tackle this problem by supporting a variety of visualization techniques and features so that software engineers can explore large traces effectively and efficiently despite their size being massive. The purpose of this paper is to describe and classify these techniques, their advantages and limitations, and set the ground for the way ahead. We achieve this by studying the techniques embedded in 26 trace analysis tools that span over several years (from 1991 to 2008) of research in the field of trace analysis and exploration.

There are currently some published summaries of the state of the art of trace analysis tools for program comprehension but they are either limited in scope (e.g., [PRW03]) or focus on other aspects of trace analysis such as trace abstraction techniques (e.g., [HL04]). The work we present in the present paper has the more focused goal of highlighting and classifying visualization techniques and features that enable software engineers explore and understand lengthy traces.

We believe that the content of this paper can be used as reference work for researchers in the field of trace visualization and analysis with a focus on program comprehension tasks who wish to build on related work and uncover opportunities for future research directions. It can also be used as a powerful guide for the design and evaluation of this class of software engineering tools. The study can also be a reliable reference for software engineers in search of better trace analysis tools, and for tool developers to build on the state of the art.
2. Background and Related Work

2.1. Overview of the Concept of Execution Traces

The study of execution traces has been the topic of many software engineering areas including software performance, software maintenance and evolution, reverse engineering, and most recently software security.

There are various types of traces that can be generated including traces of routine calls, inter-process communication, and executed statements. In fact, one can trace just about any aspect of the system that is deemed helpful to accomplish the task at hand. However, despite the various types of traces most trace analysis tools focus on traces of routine (or method) calls. Such traces tend to be at a level of abstraction that seems to be useful for program comprehension tasks. They have, for example, been shown to be useful in recovering behavioural design models [HBAL05], uncovering faulty behaviours [Sys99], helping with feature enhancement tasks [EKS01, JR97, JSB97], or detecting system inefficiencies [DJMSVY02].

It is very common that traces, once generated, are saved in text files. A trace file usually contains a sequence of lines in which each line represents an event. For example, a trace of routine calls will require at least two events to be generated from each routine: an event at the entry of the routine and another one that records the exit. There exist various techniques for generating and saving traces. The first and most common technique is based on instrumenting the source code, which consists of inserting probes (e.g. print statements) at appropriate locations in the source code. For example, to generate a trace of routine call, one needs to insert two probes at the entry and exit of each routine. Instrumentation is usually done automatically and they are plenty of tools that can be used (e.g. LTTng, TPTP, etc.). The second technique consists of instrumenting the execution environment in which the system runs such as the operating system or the Virtual Machine. LTTng, for example, is an instrumentation tool that can be used to instrument the Linux kernel with minimal overhead [FDD09]. The advantage of this technique is that it does not require modification of the source code. This is especially important when the source code is not available in the first place. Finally, it is also possible to run the system under the control of a debugger by setting breakpoints are locations of interest. This technique, however, can considerably slow down the execution of the system and it therefore not recommended.
2.2. Related Work

Perhaps one of the most comprehensive and cited surveys of trace analysis tools and techniques is the one proposed by Hamou-Lhadj et al. in [HL04]. The authors studied the features embedded in eight trace analysis tools with a focus on program comprehension tasks. Although the study covered three main aspects, namely, trace modeling, trace abstraction, and the granularity of the analysis supported by the various tools, the results have set the ground for many advances in this area. An important result reported in the paper was the proposed classification of trace abstraction techniques and which consisted of two categories of techniques: trace exploration and trace compression. Trace compression techniques operate on execution traces independently from any visualization schemes, whereas trace exploration techniques rely on tool features which involve some sort of visualization methods. Although the authors agreed that these techniques could be very useful in understanding the content of a trace, they did not explore in depth this family of approaches in their paper. The objective of our paper is to fill this void by studying the various visualization techniques and features supported in many tools including the ones presented in Hamou-Lhadj et al.’s study.

Bennett et al. presented a tool review of the visualization features that support exploration of sequence diagrams reverse engineered from execution traces [BMSGOSC08]. Their study uncovered many features that are used to explore sequence diagrams and that they grouped into two categories: presentation and interaction features. Based on their analysis of the various tool features as well as a field study that involved software engineers working on understanding the content of sequence diagrams, they have developed a tool, called OASIS (which is one of the tools that is studied in this paper), which supports most of the features found useful for trace analysis. The difference between their work and the content of this paper is that we do not limit ourselves to tools that view traces in the form of sequence diagrams; we cover also trace visualization tools including the ones that use graphs, trees, treemaps, tables, bar charts, lines, pixels, as their main display. This represents a spectrum of tool features that is considerably larger than the ones presented in Bennett et al.’s study.

Pacione et al. [PRW03] conducted a study in which they evaluated five dynamic analysis tools including debuggers based on their effectiveness in enabling software engineers to perform a number of reverse engineering tasks such as architecture recovery, identifying design patterns, etc. Although the paper provides interesting insight in the role of dynamic analysis, particularly trace analysis, in helping software engineers understand a poorly documented software system, the scope of the study is broader than the one of this paper which focuses exclusively on visualization techniques and
features supported by trace analysis tools. In addition, our study involves 26 analysis tools, which makes it more comprehensive than the work presented by Pacione et al.

Cornelissen et al. presented a systematic literature on program comprehension research through dynamic analysis in which they surveyed several research papers published in proceedings of major conferences in the field [CZDMK09]. The objective of the survey was to understand the trends in the area by classifying the surveyed studies according to three main criteria: The specific program comprehension activity targeted by the study (e.g., feature location, design and architecture recovery), the approach used by the authors (e.g., concept analysis, pattern detection, etc.), the nature of the system under study (e.g., object-oriented, distributed, etc.), and the evaluation techniques. The authors, however, did not discuss specific techniques and features used for visualizing and exploring large traces, which is the focus of our study.

3. Trace Visualization Techniques

We organized the methods supported by the surveyed trace visualization and analysis tools into three categories: Presentation methods, interaction techniques, and functional features. The presentation methods consist of the techniques used to render the content of a trace. These include tree structures, graphs, treemaps, and so on. The interaction techniques describe fine-grained features that allow the users to interact with the tool to navigate and explore the content of a trace. Examples of such techniques include highlighting the content of a trace and hiding some of its components. The functional features, which represent the third category, consist of higher-level features that are supported by the tools to perform more advanced handling of a trace content including the application of trace abstraction algorithms and animation. Interaction techniques can be considered as a specific subtype of functional features. The next subsections cover each of these categories in more detail by discussing the various approaches that have been used along with their advantages and limitations. These approaches are also discussed in the broader context of data visualization to have a better understanding of their benefits and potential shortcomings.

3.1. Presentation Methods

3.1.1. Tables

One of the traditional and frequently used means of visualizing data is tables [Was27]. The key benefit of tables as a representation technique is compactness. Furthermore, tables make it easy to locate individual pieces of information from the data. Trace analysis tools often use tables to
represent statistical information about a trace. For example, as shown in Figure 1, the Execution Statistics view in TPTP (Hyades) [Hya10] is a table that can be used to represent different quantitative attributes of a trace. In this table, the top ten methods with the highest base time and their execution related information are represented.

![Table View](image)

**Figure 1. TPTP (Hyades): Execution statistics view (from [Hya10])**

The Table View in Jinsight (Figure 2-left) is another example of using tables in trace analysis tools [DJMSVY02]. This view shows measures of the execution activity and memory used by threads, packages, classes, methods, or objects. The cumulative processing time spent by each method is represented in the table along with their total number.

![Table View](image)

**Figure 2. Left: Jinsight: Table view (from [Jin01]) - Right: Evolve: Bar chart representation (from [WWBDDHV03])**

When the number of attributes of entities is small, tables can easily handle the visualized information. However, as the number of attributes grows the chance of confusion between the columns and the difficulties to sort the significant features in a table increase. Although, hybrid table representation methods have not been put into use in trace visualization, techniques such as the Table Lens [RC94] and Parallel Coordinates [Ins97] use glyphs and bars instead of text in each cell to address the scalability problem of tables.

3.1.2. Charts

Charts use visual objects such as lines, points, bars, and shapes in an area bounded by axes to
visualize data. A chart can represent qualitative and quantitative data. One advantage of using charts is that they allow users to perceive important distinctions at a glance rather than going through the data as in tables. Another advantage of charts is that they make it easier to compare the attributes of entities.

With regards to visualizing execution traces, charts can be used depending on the relations between the data that is visualized. A simple relation such as comparison between classes or packages (or any other categorical subdivisions) can be visualized by bar charts. If the categorical subdivisions are of no particular order, the comparison is called nominal comparison. Otherwise, the comparison is a ranking one (e.g., classes are arranged by their size or fan-in count). Similarly, if the comparison needs to be based on a reference measure (i.e., a deviation) bar charts can be helpful. EVolve [WWBDDHV03] uses bar charts to illustrate the relation between different properties of a trace. This tool enables the user to assign the desired properties to the axes of the bar chart. The bar chart representation in EVolve supports two types of predefined ranking comparison, namely temporal and lexical rankings. In temporal ranking, the entities are sorted in the order in which they appear in the execution trace while in lexical ranking the name of an entity is used to specify its rank. For example, Figure 2 (right) shows a bar chart representation where the y-axis shows method invocation locations (lexical ranking) and the x-axis shows the total number of invocations occurring at each location. To avoid cluttering, the name of an invoked method is displayed when the cursor hovers over the invoking location.

ParaGraph [HE91] is a tracer and a visualization tool for performance and behavioral analysis of parallel programs that uses bar charts in its processors activity and processor task view. Once an execution trace is loaded to the tool, in the processors activity view, each bar that corresponds to a processor shows the busy, overhead, or idle status of that processor in different colors. The processor task view shows the task activity of individual processors where bars are color coded to indicate the current task that is being executed by the corresponding processor on the vertical axis at the time on the horizontal axis.

The linear view in ALMOST visualizes call trees using bar charts (Figure 12 (left)). The called function and its level are correspondingly represented by the vertical position of a bar and its color. The temporal order of calls is retrieved by scrolling the horizontal scroll bar of the view. Bar charts can also be used to visualize frequency distribution where quantitative counts of some attributes (e.g., frequency distribution of the invoked routines) of one or several execution traces are used as a metric
for further analysis.

Another type of data in execution traces is Time-Series data. VisTrace [PW08] uses points, lines, and bars in its time-series chart to express a system's behavior over an interval of time. Rivet [BSTGRH00] employs several types of charts to visualize execution data of large software systems.

Several other studies have used pie charts to visualize CPU activities of multiple nodes running distributed programs PARvis [NA94], PABLO [RANRSS93], VAMPIR [NAWH96], and Pajé [COB00]. Part-to-whole measures are also visualized using pie charts. Although different measures of trace content might be distributed differently in time, there is a possibility for correlation to exist between them (e.g., a correlation between nesting level of method calls, CPU usage, and memory usage). Points can be used for paired sets of measures to determine the impact of one set of measurement on another. Table 1 shows the visual encoding objects that are best (✓✓) or preferred (✓) for displaying quantitative relations between sets of data based on [Tex95] and [Few04].

In addition, a study conducted by Benbasat and Dexter [BDT86] suggests that charts lead them to substantially faster decision making, whereas tables often lead users to marginally better decision making (because one can lookup precise values).

<table>
<thead>
<tr>
<th></th>
<th>Points</th>
<th>Lines</th>
<th>Points and Lines</th>
<th>Bars</th>
<th>Pies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Time-series</td>
<td>✓</td>
<td>✓✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Ranking</td>
<td>✓</td>
<td>✓✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Part-to-whole</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓✓</td>
</tr>
<tr>
<td>Deviation</td>
<td>✓</td>
<td>✓✓</td>
<td>✓</td>
<td>✓</td>
<td>✓✓</td>
</tr>
<tr>
<td>Distribution</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓✓</td>
</tr>
<tr>
<td>Correlation</td>
<td>✓</td>
<td>✓✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

### 3.1.3. Graph-based Methods

#### 3.1.3.1. Graphs

Graphs have also been used to visualize information about an execution trace (e.g. SEAT [HLF05] and Program Explorer [KN97]). A graph is a visual representation consisting of a collection of nodes, and arcs linking nodes together. Collberg et al. [CKNPW03] argued that, theoretically, any problem can be encoded as a graph problem. The graphs can be readily be used to represent trace data. For example, the nodes can represent the entities such as procedures, objects, classes, or subsystems, while the arcs represent relationships between these entities, such as inheritance or routine calls.
Nodes of a graph are usually represented by a dot and arcs by lines or curves linking them.

Program Explorer [LN97] uses graphs in its “object graph” and “call graph” views. Object interaction is modeled using a directed graph called an Interaction Graph (Figure 3 (left)). The nodes of the graph represent objects and the arcs represent method invocations. Arcs are labeled with the name of the method, the time at which the invocation of the method takes place and the time at which the execution returns to the caller. An object graph shows the objects in the program where there is an arc from every object to the objects it creates. Similarly, the call graph view shows method call relationships where each caller object is linked to its callee objects.

The Reference Pattern view in Jinsight [DJMSVY02, DHKV93] is a graph-based view which is useful for studying data structures and finding memory leaks. Given a trace of object reference information, this view shows patterns of references in varying detail to or from a set of objects (Figure 3 (middle)). Bertuli et al. implemented a tool to visualize object-oriented program executions that is based on a graph-based representation approach called the Polymetric View [BDL03]. In their approach, the graph nodes are shown as rectangles for which the color and dimensions represent different statistical properties. These rectangles are then used in different views such as the communication interaction view, the instance usage overview view, and the creation interaction views. The VTune Performance Analyzer implemented by Reinders from Intel [Rei05] uses a call graph representation in which program functions are represented as nodes. The nodes are expandable, and when expanded the callees of the node are shown. In the Cels and the Summary view diagrams of AVID [WMFSI98], clusters are represented as boxes and dynamic relationships extracted from the trace are represented as directed arcs. An arc between two entities A and B is labeled with the number of calls the methods of classes in A make to the methods of classes in B. Instantiation and destruction of objects are shown as histograms associated with each box. Rectangles represent high-level system components. Directed arcs represent the interaction between these components. Histograms illustrate object population (i.e. the number of object created and deleted in each component).

3.1.3.2. UML Diagrams

UML defines several types of diagrams to be used in forward engineering for conveying design information. The UML diagrams are, in fact, a sort of labeled graphs. Therefore, they can be used for visualization purposes in reverse engineering tasks as well. This way, the visualization can take advantage of user’s familiarity with the predefined semantics of UML diagrams.

One of the most frequently used UML diagrams in execution trace visualization is the sequence
A sequence diagram can be used to show how trace elements (e.g., packages, classes, or objects) operate with one another and in what order. TPTP (Hyades) [The08] uses UML sequence diagrams to visualize trace interactions in different levels (host interactions, process interactions, class interactions, object interactions, thread interactions, and agent interactions). In these diagrams, lifelines (shown as vertical dashed lines) indicate specific roles played by instances of the trace elements. The communications between the instances are displayed as arrows. Tall thin rectangles in these diagrams, known as activations, show the period during which an instance is executing a specific task (Figure 3 (right)).

When dealing with large traces, in TPTP’s interaction view, the user can hardly see the whole sequence diagram in one view and may easily get lost in the diagram. This typically holds for any tool that uses sequence diagrams for visualizing execution traces. To alleviate this problem, many tools use a second view (usually synchronized with the sequence view with less detail) that helps users having an overview of the trace or additional information about it. For example, OSE [BMSGOSC08] uses a linked view called the outline view (Figure 4 (2)) to display a miniature version of the complete sequence diagram. As shown in Figure 4 (the view on the left) when an area is selected in the outline view, the detailed version of the selected area is displayed in the Sequence Diagram view.
Figure 4. OSE: the sequence diagram view is on the left (1) and on the right (2) is the outline view (modified from [BMSGOSC08])

The temporal message-flow diagram in ISVis [JR97, JSB97], which is used to display the detailed content of the trace, is a variant of UML sequence diagrams (Figure 5 (left)). ISVis also uses an auxiliary view, called the mural view, to help users better navigate through the message-flow diagram. The mural view is a miniature version of the trace where the trace events appear as colored lines. Also, in the Collaboration Browser [RD02], an instance of a collaboration pattern (which represents sequences of calls repeated non-contiguously in a trace) is displayed as a UML sequence diagram. Similarly, in VET [MWM06], execution traces are displayed in two synchronized views using UML-based visualization: a sequence diagram view and a class diagram view.

Shimba [Sys00] is based on a tool called SCED [KMST96] for analyzing traces. SCED is a forward engineering tool that permits the creation and manipulation of scenario diagrams, which are similar in principle to UML sequence diagrams. SCED also has the ability to synthesize state machines given several scenario diagrams.

Since UML diagrams are labelled graphs, they should be drawn in a way that keeps the textual label of each node readable. This requires that there should be no overlapping nodes, which, in turn, causes less effective use of screen real-state.

3.1.3.3. Trees

De Pauw et al. [DLVW98] suggested using tree structures to represent routine call traces instead of UML sequence diagrams. According to the authors, tree structures tend to be less cumbersome than sequence diagrams. The argued that the absence of cycles in trees and the hierarchical nature of trees make them generally easy to comprehend and interpret in comparison with graphs [CMS98].

Shimba [SKM01] uses Rigi [MK88] to visualize hierarchical graphs of system artefacts and their dependencies as trees. Using Rigi, a software engineer can run a few scripts to exclude the nodes that
are not of interest and keep only those he or she wants to investigate. Ovation [DLVW98] visualizes traces using a view based on tree structures called the execution pattern view (Figure 6 (left)). The execution pattern view lets the user browse the program execution at various levels of detail. For example the user can collapse and expand subtrees, show only messages sent to a particular object, remove contiguous repetitions of sequences of calls, zoom in and out the trace panel and many other features. SEAT [HLF05] takes traces of routine calls as input and displays them using PictureTree; a tree-like control window similar to the tree browser (Figure 6 (right)). Traditionally a non-leaf tree node has two states: expanded or collapsed. In SEAT, because various filtering algorithms can be applied to the trace, some nodes in the trace can be hidden if they meet certain criteria. As a result, in a certain exploration point, a non-leaf node can have the following expanded states: fully expanded, that is, all the nodes are shown; and partially expanded, meaning some or all the children are hidden by some filtering algorithms, so only part of children are currently displayed.

When a subtree is first displayed, it is in collapsed state, indicated by the icon '+'. When a user wants to expand the subtree to visit its child nodes, the icon that represents the new state will change to either '-' icon or '~' icon depending on whether there are children nodes being hidden by any filtering algorithms. If no children nodes are marked filtered, '-' icon will be displayed for the subtree. Otherwise, '~' icon will be used to indicate that some children cannot be seen. By using Ctrl+Click, the user can switch between these two icons, and refresh children nodes respectively.

Jinsig [DJMSVY02, DHKV93] uses an “execution patterns” view for showing a sequence of method calls as a tree. In this view, the user can click on collapsed methods to see more of the tree. “Object references” is another view of Jinsight which shows patterns of memory references as a tree.
that can be used for finding memory leaks in Java programs.

Although graph-based representation methods are helpful in visualizing different aspects of a system, they tend to become incomprehensible as system’s complexity increases. They lose their effectiveness when dealing with a large number of nodes [Kni00]. The reason is the lack of visualization space as the amount of information to display grows. In general, the graphs can be categorized according to their size. For this, [Mar01] uses number of elements (nodes and arcs) as a metric: small graphs (less than 100 elements), medium graphs (less than 1000 elements), large graphs (less than 10000), and huge graphs (more than 10000 elements). Shneiderman et al. suggest that successful graph representations are small ones where users can count the number of nodes and links and follow each link from the source to the destination [SA06]. Therefore, the challenge of graph-based techniques is to provide visualization which display many nodes but remain readable, navigable, and intractable. For this, many graph layouts (that we briefly review later in this section) have been proposed to arrange the nodes and arcs to achieve more readability, understandability, usability, and etc.

When representing a directed graph, a hierarchical layout can be used to reveal the dependencies between objects by assigning nodes to distinct levels. Research effort in this area has been mostly related to improvements of Sugiyama’s algorithm [ES90] usually used to draw directed graphs. A hierarchical layout is perhaps the most common layout when visualizing software structures. Shimba uses Rigi to represent directed acyclic graphs in a hierarchical layout based on Sugiyama’s algorithm.

![Figure 6. Left: The Execution Pattern View of Ovation (from DLVW98) - Right: SEAT: A Snapshot of PictureTree (from HLF05)](image)

Eades [Ead84] proposed a class of layouts called force-directed layouts (also known as spring-
embedding) for drawing general graphs. These layouts use a physics metaphor in which arcs of the graph are springs and nodes are repulsive magnets. The physical forces of pulling and pushing the nodes apart continues until a layout is reached in which these forces are in equilibrium. This class of layouts has the advantage of grouping nodes into identifiable groups (e.g., in symmetrical layouts where nodes are grouped as symmetrical patterns) based on increased connectivity (e.g., could be used to check loose coupling). North et al. [NK00] suggest that call graphs are best visualized using force-based layouts. Rigi is used by Shimba for displaying general graphs in a spring embedder layout (Rigi itself uses an integrated layout tool called GraphEd [Uni99] to implement different layouts). TraceVis [DH07] also uses a force-based layout in the representation of a call graph because this dynamic layout can be updated easily on every change of the graph during an execution (Figure 7(a)). Thus, another advantage of force-based layouts is their good adaptability (easy adjustments of the forces in order to achieve additional layout goals). On the other hand, the main shortcoming of force-based layouts lies in their slowness and potentially unpredictable results.

As mentioned earlier, UML diagrams can be used in visualizing execution traces for software maintenance, reverse engineering, and re-engineering purposes. Koschke [Kos03] suggests that since in these domains the graphs are often quite large and are usually generated through an automatic process, automatic layouts can be helpful. As proposed by Tamassia [Tam90], one way to achieve very good results in UML-based representations is to use orthogonal layouts. In orthogonal layouts, arcs are drawn either horizontally or vertically with orthogonal bends to reduce the number of arc crossovers. Similar groups of layouts can be used for drawing trees. However, the fact that, unlike graphs, trees have no cycles makes them easier to layout. Therefore, when we have a noncircular graph in which the hierarchical relations are unimportant a tree layout could be used to automatically draw a comprehensible tree structure.

(a) An overview of an execution trace drawn in a force-directed layout in TraceVis (modified from [DH07])
Figure 7. Sample usage of graph layouts in trace visualization tools

The increasing power of graphic displays offers a new opportunity to incorporate new methods for 3D graph layouts. Some studies suggest that 3D layouts can make graphs easier to understand [WF94]. In comparison with 2D layouts, in 3D layouts an extra attribute of the traced elements can be mapped to the third dimension. Nested layout is a 2D orthogonal layout which results in diagrams similar to UML package diagrams [NK00]. Parker [Par98] implemented a tool called NestedVision3D (NV3D) for visualizing software execution in 3D (Figure 7 (c)). In NV3D, large relational information structures (e.g., method uses or part-of relationships respectively between methods and objects) are visualized in a 3D nested layout. Reiss et al [Rei98] also implemented a 3D layout to visualize call graphs.

As for trees, Collapsible Cylindrical [DE01] and 3D Cone Trees [RMC03] are among the 3D layouts where a 2D drawing of the graph is wrapped around a 3D object such as a cube, a sphere, a cylinder, or a cone [MKP04]. Helix Cone Tree Visualizations can be used to visualize runtime behaviour of Java programs. In their work [JCE04] Joshi et al proposed the visualization of an execution trace of a Java program using 3D call graphs (Figure 7 (b)) that is an extension of the original Cone Tree visualization technique. Their tool also enables the user to create multiple views of a call graph from one execution trace to help users with various level of expertise.

3.1.4. Space-filling Methods

One issue with graph-based representation methods is their inefficient use of screen real-state because of their node-link nature. Moreover, when dealing with large number of nodes, graph-based representation methods become incomprehensible. Space-filling techniques aim to present the
maximum amount of structured information in the minimum amount of space. Examples are the Treemap method, SunBurst, and the Information Slices technique.

3.1.4.1. Treemaps

Treemaps are designed to visualize data modeled as trees, such as directory trees, where showing the attributes distributions is more important than showing the relations between nodes [JS91]. The attributes can be ranked by their importance to be used in generating the Treemap from the tree model a software system (a sample Treemap is shown in Figure 9 (left)). In this technique, each node of the tree is displayed as a rectangle whose area is relative to the most important attribute for the corresponding software entity such as size (number of lines of code), number of sub-entities, etc. All offspring of a node are drawn as rectangles inside the node’s rectangle. The same process is applied on each drawn rectangle recursively until leaf nodes are encountered. A second attribute (e.g., number of fan-in) can be used in calculating the order in which the rectangles are drawn. If the treemap is color-coded, the range of a third attribute can be mapped to the range of colors used in the treemap. Treemaps are used to visualize containment relationship while allowing absolute comparison of the entities in the software structure.

Therefore treemaps are recursive partitioning and filling of a space where the total space is represented by the whole rectangle. In this way, the most important entities of the system can be displayed as outliers on treemaps. The main advantage of treemaps lies in their efficient use of display space. Whereas treemaps scale well with both increasing size and complexity of the modeled system, an issue with the original treemaps is the difficulty in perceiving the hierarchy as the size and complexity of the system grows. To overcome this problem different layouts of treemaps have been proposed (we review these layouts later in this section).

Treemaps first were introduced and used by [JS91] as a method of visualizing the file structure on a hard disk to find large files that could be deleted for disk cleanup. Many tools use treemaps to represent large volumes of source code while showing the code structure as a hierarchy. For instance, SourceMiner [CMM09] uses the Prefuse Java Toolkit [Pre08] to create treemaps to represent large codes in software modules. Tracing tools can benefit from the ability of treemaps in showing structures along with a number of important attributes of a trace. JProfiler [Cho06] is a commercial tracer that helps Java programmers find performance bottlenecks of a Java program. This tool provides the programmer with a view in which a trace of method calls in the form of a call tree is represented as a treemap. In this view, the sizes of the rectangles are proportional to execution time of
methods and depth limiting can be used to set the maximum nesting depth. The detailed information about each method can be shown as a tooltip when the programmer hover the mouse over rectangles. A number of tools (e.g., ORCA [SST08]) represent the elements of an execution trace by highlighting those elements in the source code to make an easy mapping between the trace elements and the source code elements. In this case, the problem of visualizing a trace can be reduced to representing the executed elements. ORCA displays the entire source code within one screen using treemaps. For this the classes that contain the executed elements are show as rectangles of the treemap and the arrangement of the rectangles shows the call edges between the elements. Inside the rectangles the executed elements are highlighted. Using this approach, the programmer can focus on interesting call edges, read the source code related to the edge. Gammatella by Orso et al. [JOH04] is a toolset for collecting, storing, and visualizing program-execution data gathered from deployed instances of a software product. One of the views in Gammatella provides programmers with a treemap representation of execution-related information. This view can be used for visualization of profiling information. For this, first the programmer generates a trace by exercising a set of specific features. Then, the code elements are colored according to their frequency in the trace (red for the elements that are executed very often, a green for elements that are executed rarely). The treemap shows the system files as rectangles according to the hierarchical structure of the system. The color of each rectangle which maps the color distribution of the statements in the source file of that rectangle can help programmers find places in the code that are executed most often for next potential optimization steps. Given a set of successful and unsuccessful traces Gammatella can assist developers in fault location by coloring the treemap in a different fashion. The percentages of successful and unsuccessful executions traces that execute a code element determine a hue value for that element (a greener hue to represent more confidence in its correctness and a redder hue for suspiciousness). The color of the rectangles in the Treemap then can be used to spot the more suspicious files. Bloom introduced by Reiss et al [RR03] is another tool which uses a variant of treemaps where the size of each rectangle (representing a method) corresponds to the amount of time spent in that method, the color is used to hotspots, and a third dimension Z reflects the fan-in value of the method.

3.1.4.2. Box Tree

A box tree, introduced by Reiss et al [Rei01], is a tree representation in which the children of parents nodes are drawn next to each other without having to display the arcs. This visualization method is used in Bloom [RR03]. Bloom can be used for understanding software systems through visualization.
Figure 9 (middle) shows the visualization of a trace of routine calls as a box tree. In this visualization the routines are represented as boxes. The size of each box in X, Y, and Z axis respectively represent the nesting level, time fraction taken from parent, and number of call of the routine represented by the box. Boxes are also color-coded to draw attention to the routines that have longer execution time. The authors suggest that the representation makes it easy to determine where and why the program is spending its time.

3.1.4.3. SunBurst

SunBurst [SCGM00] is a space-filling technique which displays items in a hierarchy and their containment relationship on a disc. In this technique, the root of the tree model is at the center of the disc while its children appear as arcs on the circumference strip of the disc. If a child is selected it expands to a new level on the same sector. In this way, SunBurst displays depth by mapping it to radius in such a way that deeper nodes are farther away from the center. Moreover, size, angle, and color of the sectors (slices) can correspond to the three attributes of the represented item. Baobab [Dis05], a disk usage analyzer in gnome, scans folders and presents the disk usage statistics reports in a SunBurst display (Figure 9 (right)).

A comparative study of space-filling techniques conducted by [BCS04] shows that SunBurst can improve task performance, both in correctness and in time for novices by clearly displaying the structure of the hierarchy. It also showed that SunBurst outperformed Treemaps in tasks related to structural interpretation (e.g., locating the deepest directory). However, SunBurst may be less scalable because of using one dimensional circumferential space rather than the two dimensional area available to Treemaps.
Similar to graph-based techniques, space-filling techniques can have different layouts. As mentioned earlier, one issue with the original Treemaps is the difficulty in perceiving the hierarchy as the size and complexity of the system grows. To overcome this problem Wijk et al [WW99] proposed a layout technique called Cushioning. This layout makes use of shading to improve the perception of structure by simulating a cushion-like curved surface on Treemap rectangles (Figure 10(a)).

Another Treemaps layout for large hierarchical structure is the squarified layout proposed by Brils et al [BHW00]. This layout tries to create much squarer rectangles (Figure 10 (b)). They suggest that squares are easier to detect and point at and therefore this layout can improve the accuracy of the representation. StepTree, proposed by Bladh [Bla02], is a variant of a squarified Treemap layout for file browsing. In this 3D layout, rectangles are presented as boxes for which the heights show the depth in the tree (Figure 10 (c)). Churcher et al [CKI99] also used Virtual Reality Modeling Language (VRML) to implement a 3D version of Treemaps which shows the relation between the Treemap and the underlying tree model.

### 3.1.5. Murals of Lines and Pixels

Ball et al. [BE96] defined Line representation, a graphical depiction of the entire source code of a program. This global overview of the source code is given by reducing each line of code to a single row of pixels with row length and indentation relative to the ones of the original line in the code. The lines in this technique can be color-coded to show a range of values of a metric (e.g., showing code age from old to new by coloring their corresponding lines from green to red). Figure 11 (left) shows the use of a line representation for a global overview for a pair of files. Rivet [BSTGRH00] execution trace visualization tool and SeeSoft system, developed by Eick et al. [ESS92], use line representation
to show source code.

Jerding et al. [JS95] defined information mural, which is a representation method similar to line representation. Information mural aims at displaying large information space within one view. It creates a miniature representation of the information space (e.g., source code, execution traces). Roughly, this miniature representation is drawn by scaling the items in the trace into the available pixels (or line) in the mural space, overwriting items that happen to overlap. The overwriting varies the intensity of location in which the overlap has happened to make it visually more apparent. This is known as anti-aliasing. An information mural can easily show repeated sequences and patterns in the overall distribution of information. If used to represent execution traces, the mural is called an Execution Mural. An execution mural can give a quick insight into different phases of the program execution by creating a miniature representation of the entire trace. This is done by emphasizing repeated sequences of events in trace using intensity shading and changing the size or color of pixels in addition to what has been mentioned.

In ISVis, execution traces are visualized using two kinds of diagrams: the execution mural and the temporal message-flow diagram (a variant of UML sequence diagrams). The two diagrams are connected and are presented in one view called the scenario view as shown in Figure 11 (right). The execution mural is used to create a miniature representation of the entire trace that can easily show repeated sequences of events. The software engineer can spot a pattern on the execution mural, select it and investigate its content using the temporal message-flow diagram.

EXTRAVIS offers an overview of the execution trace through its massive sequence (mural) view (Figure 5 (right)). This view is similar to the information mural view in ISVis. The system’s structural elements are shown at the top of this view while the main part of the view visualizes the call relations. This way, the mural view helps the user to visually detect the different phases of the system’s execution. Being aware of the execution scenario, the user can also hypothetically relate the repetitive (or ordered) patterns in the mural view to the repetitive (or ordered) features in the execution scenario. Then, zooming on a pattern, the user can have a closer inspection of the pattern to verify his hypothesis.
In their prototype tool, Jerding et al. used a horizontal execution mural to examine message traces from object-oriented programs [JR97]. They also defined a Pattern Mural which visually represents a summary of a trace in terms of recurring execution patterns. Representing the trace in a horizontal or vertical mural can however be an issue when the number of trace elements is larger than the number of lines in a screen. To address this issue, ALMOST, a trace analysis tool proposed by Renieris et al. [RR99], uses a mural representation to represent traces, in which the visualization is mapped onto a spiral view (Figure 12 (left)).

Line representation and Information Mural both suffer from the fact that not all items of a trace can be visualized in one mural view in a non-ambiguous fashion in cases where the number of items is more than the available horizontal pixel lines. The pixel representation, proposed by Ball et al. [BE96] shows each item using a small number of color-coded pixels. In this way, it is possible to achieve a higher information density than the two previous approaches (i.e., line representation and information mural). As shown in Figure 12 (right) the pixels can be ordered left to right in rows according to the original order of the items (e.g., corresponding line numbers) or according to the color-coding criteria (e.g., from the old code lines to new ones: from red to purple). Using this representation, it is possible to show over a million lines of code using a standard high-resolution monitor. The pixel representation can work as a scrollbar to help to ease the problem in Line representation and Information Mural stated earlier as done in the browser window of Ovation [DLVW98]. Ovation by De Pauw et al. [DLVW98] incorporates a zoomed-out execution mural-like view to show the highest level of the execution of a program. This view gives a general idea of the different phases in the program while reflecting the stack depth at each particular phase as the width...
of the pattern. Color coding is used in this view to indicate the classes that are widespread in each phase.

![Figure 12. Left: Trace visualization tool with different views of ALMOST (from [RR99]) Right: Pixel representation (from [BE96])](image)

### 3.1.6. Virtual Realities

Several researchers have exploited real-world visual metaphors such as buildings, cities, and planets to help the user grasp insights about software. The idea behind using Virtual Reality Environments (VRE) to visualize software systems is to take advantage of the user’s natural perception and his existing knowledge about scenes, their composing elements and properties. In these techniques, first, different elements of the software system are mapped to the elements that compose the selected metaphor. Then, the visual attributes of the metaphor elements are defined in accordance with the metrics of the software elements. One popular VRE is the city metaphor proposed and used by Santos et al [DGALTP00] in the CyberNet project to build three-dimensional virtual worlds to visualize network data and analyze network services. Knight and Munro [KM99] used the same idea to visualize Java code in a 3-D virtual world called Software World where the code is represented as buildings and districts in a city. The same idea was exploited by Wettle et al [WL07] in CodeCity in which classes are visualized as buildings in city districts, which, in turn, represent packages. The VREs can also be used to shown dynamic behaviours of a software system by visualizing execution traces.

In order to build a suitable representation of execution traces, Dugerdil and Alam [DA08, AD07] used the city metaphor (originally proposed by Knight [Kni00]) in which the classes and files are shown as buildings in a 3D city landscape. The sizes of the buildings in Software City can be set using predefined metrics (e.g., length of the code of classes, number of methods, number of attributes,
and etc). The buildings are distributed in different city districts (the city itself is the biggest district), where, each district shows a containment relation (e.g., a package or a directory hierarchy). Zooming in on a building makes it possible to see the methods and variables inside it or to check the corresponding part of source code. The user can also see relationships between the selected buildings (or methods inside buildings) as directed pipes between them (see Figure 13 (right)). Trace visualization in a software city is done through two types of dynamic trace display: macroscopic view and microscopic view. The idea behind having two views is to cope with the large traces and size explosion problem through segmentation. That is, a trace is represented as a sequence of contiguous segments in a macroscopic view and, then, the sequence of events in any selected segment can be inspected in a microscopic view. The city metaphor is also used by Panas et al [PBG03] to depict cost related information of a system in software production. In their work, they tried to make software maintenance an entertaining task, when navigating through a software city in 3D. For instance, as shown in Figure 13 (left) when visualizing a software system the components that are frequently executed are shown in fire and the components that cross-cut many features (the calls to these components are highly distributed) are shown in blue and green.

Another attractive metaphor used by Santos et al [DGALTP00] in CyberNet project, is the Solar System to which network services and workstation information are mapped. Graham et al [GYB04] also used the solar system metaphor for 3D visualization of object oriented software metrics (Figure 15 (left)). In general, the solar system metaphor has the ability to effectively show size and inheritance respectively as planets size and orbits (rings).

3D height map (mountains) is another interesting metaphor used by Telea et al [TV04] to give an overview of a software system. In this visualization, classes are distributed on the map the call dependency relation between classes determines their relative distance on the map (Figure 14 (a)).
This way, height plots encode the frequency of calls. The height map can also be used to show the density of elements per unit area. Therefore, a height map is effective for detecting tightly coupled subsystems. ScenarioML implemented by Alspaugh et al [ATB06], uses a video game like metaphor to visualize the interactions between entities in a scenario. For this, animated agents visually perform the scenario and vocalize its textual components (Figure 14 (b)).

![Height Map and ScenarioML](image)

**Figure 14. Hight Map and ScenarioML**

VARE [AMNB06] is web-based visualizations of remotely executing object-oriented Java programs that uses an information landscape to layout of all the events from an execution trace. Figure 15 (right) shows the information landscape in VARE, where the red box represents the main class, blue spheres object creation, green boxes method calls, white cones method returns and end of the object. In [ZZHW09] uses linked 2D planes to visualize cross-references across multiple execution traces to represent meaningful and viewable comparison between traces from different perspectives. The authors claim that linked 2D planes can help feature location and program behavior verification.

Although VREs can be easy to use as they take advantage of user’s pre-existing knowledge of the environments, they can quickly become impractical as the amount of information to be visualized grows. Moreover, natural limitation of navigations in these types of representation methods is an issue in VREs.

### 3.1.7. Hybrid Representation Methods

Many tools use innovative combinations of original types of representations as new representation methods that can better do the visualization tasks in comparison with the original ones. These techniques can be considered as Hybrid representation methods. The Circular view in EXTRAVIS [CHZMWD07, CZHMDW08] is an example of a hybrid representation method that combines ideas from SunBurst and hierarchical graphs (call trees) with innovative ways of bundling hierarchical
edges.

As shown in Figure 5 (right), this view shows the system’s structural entities as a strip around its circumference. Each parent element and its children form an arc on this round strip. The interrelationships of these structural entities are shown on the inner circle. Since viewing these relations as straight lines may cause visual clutter, EXTRAVIS bundles these relations together. The implicit relations between the parents and the explicit relations between their children can be better represented through the bundled splines. These splines are color-coded to represent their chronological or directional properties. The circular view also offers different levels of abstraction by collapsing. When the users collapses (folds) an element, all of its child elements go out of sight and the relations pertaining to these child elements would be lifted to the parent element. A Massive sequence view is synchronized with the circular bundle view helps in a more precise inspection of the interactions pertaining to the involved elements.

3.2. Interaction Techniques

Interaction techniques are used to establish and facilitate the dialog between the user and the representation. This dialog is a sequence of interaction features (where one feature linked to other features) used to enrich user exploration and discovery.

3.2.1. Highlighting

Highlighting is a feature by which users can mark a representation element of interest to keep track of it. A representation element can be a subset of components, relations, and patterns that form the visualization. Depending on the representation method, highlighting can be performed by coloring, bolding, tagging, annotating, labelling, or use of other techniques which result in visual
discrimination of an element. For instance, in graph based representations annotating a node or a relation can help users remember certain aspects of that node or relation without rebrowsing the whole graph and also help add the gained knowledge to the presentation [BG98]. As the number of representation elements grows it becomes harder to track or relocate an element of interest. Therefore, even after a change in the arrangement of the elements or switching between different views, users can easily track and relocate the element, which in turn, helps the user keep his/her mental map.

3.2.2. Rearranging

Rearranging is an interaction feature which provides user with the ability to change the parameters of the representation. The goal of rearranging is to derive a new perspective (that might reveal new information) or a more suitable and a more understandable and readable representation. Depending on the representation technique, rearranging can be implemented and performed differently. For example, in graph-based representation, a graph needs to be represented in a way such that it has fast and high readability. It is also important for a graph representation to have high density so that the user can see more in one sight and use less memory. These and many other similar concerns are addressed by graph layouts using the rearranging feature. One graph can have many layouts between each of which the only difference is the arrangement of the nodes and arcs which impacts readability, understandability, usability, and etc. The sorting and rearranging rows and columns in tables (e.g., in TPTP), modifying the scales (of the axes) and change of the origin in charts are other examples of the rearranging feature.

3.2.3. Switching

Switching between different representations of the same data helps find appropriate representations for certain types of data. As discussed earlier, some type of visual encoding objects in charts are more suitable for displaying certain quantitative relations. Therefore, switching the representation might be helpful in better understanding of the relations or even uncovering new aspects of these relations. Moreover, depending on the type and the size of the data, switching between different representation techniques can be used to understand and discover different aspects and dimensions of the visualized data in different levels of granularity. A combination of highlighting and switching can be used as a mechanism to increase the navigation capabilities.
3.2.4. Showing More/Less Details

This interaction technique allows user to adjust the level of abstraction of the displayed information. The user can change the representation from an overview to a detailed view using this “showing more/less details” interaction feature. Depending on the representation technique, extending and collapsing can be implemented differently. For example, this can be implemented in graph-based representation techniques by the having the sub-trees collapsed to their parent and extending a node to see its children as we have in SEAT and many other trace visualization tools that use tree structures. In space filling techniques, SunBurst implements this interaction feature using animation as details-on-demand. Zooming-in and zooming-out are also used in many representation techniques to vary the amount of details displayed in a view. For instance, the mural view in EXTRAVIS, charts in EVOLVE, sequence diagrams in TPTP, the linier view in Almost, the trace panel in Ovation, and the sceneries in EvoSpaces all support zooming. Interactive distortion techniques [Kei02] are another example of this type of interaction where an overview is preserved during drill-down operations. The basic idea is to show portions of the data with a high level of detail while others are shown with a lower level of detail. Popular distortion techniques are hyperbolic and spherical distortions which are often used on hierarchies or graphs but may be also applied to any other representation method.

3.2.5. Hiding

<table>
<thead>
<tr>
<th>Interaction Techniques</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highlighting</td>
<td>Mark a representation element of interest to keep track of it</td>
</tr>
<tr>
<td>Rearranging</td>
<td>Change the parameters of the representation intended to derive an improved perspective of the same representation</td>
</tr>
<tr>
<td>Switching</td>
<td>Switching between different representations of the same data intended to find appropriate representations for certain types of data.</td>
</tr>
<tr>
<td>Showing More/Less Details</td>
<td>Allows user to adjust the level of abstraction of the displayed information</td>
</tr>
<tr>
<td>Hiding</td>
<td>Hides undesired elements to reduce the complexity of focusing on desired elements</td>
</tr>
</tbody>
</table>

When the number of representation elements is high, focusing on a desired subset of elements can be a difficult and complex task. The “hiding” interaction feature can be used to hide undesired elements to reduce the complexity of focusing on desired elements. Although the hidden elements are not displayed, the actual data from which the representation is built usually remain unchanged. This makes it possible for the user to show and recover the hidden elements. Hiding can be implemented
differently depending on the representation technique. For example, in TPTP the lifelines (representing classes) and communications (representing methods) can be hidden. In ISVis a user can decide to hide the classes that belong to the same subsystem and only show the interactions between this subsystem and the other components of the trace. The user can hide method return events in VET that results hide them from its two views according to their defined rules. SEAT also supports manual hiding. The Scene tool [KM96] uses several hiding techniques including call compression, Partitioning, and projection and removal of trace content. A summary of interactions features is presented in Table 2.

3.3. Navigation Strategies

Navigation strategies can be regarded to as the use of a virtual camera across the representation scene to enhance visualization. This camera can potentially move, change lens, and show overviews to help the user uncover where he/she is within the representation (context) and makes it possible for the user to go to other locations of interest and show them in more details (focus). Navigation strategies can be implemented differently according to the representation technique used in the representation scene. Camera movement-based navigation can be seen when users perform panning, zooming, and scrolling. In Panning, the user can grab the scene and move it in different directions using a mouse. Zooming gives the feeling to the user as if the camera is moving towards or away from the scene. Scrolling is similar to the panning except the camera movement is controlled via scrollbars. The Histogram, Execution, Execution Pattern, and Reference Pattern views in Jinsight and the liner view in ALMOST are examples of views in trace analysis tools that support zooming and scrolling.

A major drawback of camera movement is that it is hard to understand the context (the area surrounding the currently viewed scene) in which the camera is moving, which, in turn, can result in navigation and comprehension difficulties. An advantage of scrolling over panning and zooming is that in scrolling the user can approximate where he is based on the status of the scrollbars. That is, the size and the position of the scrollbars can correspondingly be indications of the scale and the position of the visible area within the entire scene. Since in zooming and panning it is hard for the user to find out the context, showing an overview in a different (though usually embedded) view at the same time might be helpful. In fact, in comparison with scrolling, an overview equipped with zooming and panning could be more practical as there would be no need for guess works in finding the position of the visible area within the entire scene. The user can also roughly navigate to a location of interest within a view and then refine to the exact place of interest within the zoom or pan view (detail view). This makes overview equipped strategies helpful in quicker navigation. For example, in Rigi, one
window displays a portion of the scene in detail, while another window shows a miniature version of
the entire scene in which the displayed detail area is highlighted. The developers of Scene [KM96]
also point out that horizontal scrolling makes the diagrams cumbersome and that there is a need for
techniques that center the information conveyed by scenario diagrams on the screen. They name this
problem the focusing problem.

Table 3. Navigation strategies and their pros and cons

<table>
<thead>
<tr>
<th>Navigation Strategies</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camera Movement-based</td>
<td>Basic navigation, user familiarity</td>
<td>Context of the current view is not immediately apparent</td>
</tr>
<tr>
<td>Show Overview</td>
<td>Easier finding of the position of the visible area within the entire scene</td>
<td>Decreases the efficiency in usage of screen real-state due to the extra view</td>
</tr>
<tr>
<td>Change Lens</td>
<td>More efficient use of screen real-state: the details (focus) and the overview (context) both fit in a single view</td>
<td>Virtually impossible to determine the actual distance between representation elements</td>
</tr>
</tbody>
</table>

Another way of preserving an overview while showing sufficient details is to demonstrate one part of
the same view through a different camera lens. Then, the area shown through the lens is called focus
and the area surrounding the lens is called context. The main advantage is that the details (focus) and
the overview (context) both fit in a single view. A famous example of a focus and context view is the
fish-eye lens [SB92] where the focused elements are displayed in great detail while the rest of the
elements have less details. SHriMP [SBM01], a software exploration tool, shows inheritance
relationships in a software system as nested graphs using the fish-eye technique. Table 3 shows
navigation strategies along with their pros and cons.

3.4. Functional Features

3.4.1. Search

The search feature in trace analysis tools can help users find the trace elements of interest. Depending
on the size and the complexity of the system under analysis, the lack of a search feature in trace
analysis tool can easily slow down or even make the whole investigation impractical. For example,
although using EXTRAVIS brings about much convenience to maintainers, it suffers from the lack of
search capabilities.

3.4.2. History

Shneiderman suggests History related actions are among the tasks that a visualization user performs.
Users may change the visualization (e.g., change the parameters of the representation or hide elements). In this case, users need support for undo. The same stands for redo (replay). Therefore, it is necessary for trace visualization tools to keep a history of the performed actions. This history can be as simple as recording one step back and one step ahead or as complete as recording all the interaction performed by the user during the visualization.

3.4.3. View-Related Functionalities

Trace visualization tools usually use different views for visualization. Multiple views can be used to open more than one trace and help the user compare their content. They can also be used to represent different aspects of the traced system at various level of granularity. It is important for the views to be linked together to make the interaction and navigation easier and also to keep the consistency and integrity of the visualization. That is, when the trace elements are modified in one view the user expects to see the modification reflected in other views. For this, synchronization between views is needed. Furthermore, synchronized views can help in quicker investigation of the system since the views contain complementary information needed for effective comprehension of the system (e.g., one can find his context of interest in one view and focus on his element of interest in a more detailed view). Linked views can also help in providing easy and fast access to underlying source code of the system which is often used during system comprehension. For example, in SEAT, at any time, the analyst can map the trace components to the source code (if available) to retrieve more information about the trace components. In ALMOST, double clicking on a method call in the linear view displays the corresponding implementation of the method in the source code view. Similarly, in VET, the location within the execution trace and the information about the selected element are synchronized in the two available views. Zooming in on a building in Evospaces allows displaying the methods and variables inside it or to check the source code of each element.

3.4.4. Data Preparation

Other than introducing their compatible trace format, trace visualization tool may have built-in trace data generation or trace data conversion functionalities. We refer to these as Data Preparation functional feature. In some tools such as Jinsight [DJMSVY02, DHKV93], AVID [WMFSI98], Ovation [DLVW98], Shimba [SKM01], ISVis [JR97, JSB97], and ParaGraph [HE91] data preparation functionalities are included as front-ends to generate trace data. The trace generation is implemented in three different ways: instrumenting the program under study, instrumenting the environment where the program is getting executed, or generating a trace using a debugger which
controls the execution of the program. Tools like VET [MWM06], SEAT [HLF05], and EVOLVE [WWBDDHV03] are among tools that include a trace data converter as their data preparation functionality.

3.4.5. State Recording

Bennet points out the importance of saving views by stating “either to share or to revisit, [saving views] is also important when documenting a user’s understanding of the diagram” [BMSGOSC08]. An insightful representation achieved by using one copy of a tool should be export/importable to another copy of the same tool. Furthermore, a maintainer may use different tools to analyze traces of object interactions. For these reasons, saving the state, views, and diagrams (which we refer to as State Recording) is one of the most important functionalities in trace visualization tools. Shneiderman [Shn96] refers to this as the necessity for the tool to enable the user to extract subsets of the data for further analysis and apply the results of an analysis to other data sets. The necessity of the state recording functionalities brings about the question of interoperability between these tools as they use different exchange formats. An example of an exchange format for traces of routine calls is CTF (The Compact Trace Format) proposed by Hamou-Lhadj et al. [HL04].

3.4.6. Automatic Parameters Tuning

Most interaction features require setting some sort of parameters. For example, the rearranging feature, discussed in Section 3.2.2, necessitates from the users to set the parameters of the representation so that new insights can be achieved. Although manual tuning could be helpful for advanced users who have enough experience with the tool or the system under study, it is not practical for novices. Therefore, many trace visualization tools provide users with predefined parameter settings. For example, tool may have predefined graph layouts for their graph-based representations or support their table-based representations with different types of sort. This can provide the users (both beginners and advanced) with a set of predefined states from where they can continue their investigation.

3.4.7. Animation

Animation can be defined as rapid and smooth display of a sequence of transitions between different inanimate states of the representation of a system. It can also be regarded as showing a specific behavior of the system as it changes over time. Animation can help the user better understand
sequential behaviours of the system. This way, a large of number of events pass in a matter of seconds and yet give an understanding to the user, something that is hardly possible to achieve by exploring the events one at a single time. As an example of the use of animation in trace visualization tools, in AVID [WMFSI98], the analyst can control the sequence of events he or she wants to visualize. This is done by breaking the execution trace into a sequence of views called cels. Animation techniques allow the analyst to show the whole execution cel by cel (which is also called play mode), stop the animation, and go forward and backward. These techniques aim to reduce the information overhead when dealing with large execution traces. Furthermore, AVID contains a summary view in which all the interactions are shown.

In both VET [MWM06] and ExtraVis [CHZMWD07, CZHMDW08], it is possible to “play” the execution trace in forward or reverse one event at a time. This animation feature gives the feeling of being at different points in a continuous execution of the program under study. ISVis [JR97, JSB97] uses animation to show the instantiation of objects and the message passing among objects in a single view. In VARE [AMNB06], the objects in information landscape grow starting at the red box and animated from right to left (See Figure 15 (right)).

Although animation techniques can help reduce the information overhead, traces are very large and there is a need to investigate more techniques to reduce their size. In [CHMY03], AVID was improved to consider compression techniques based on sampling.

3.4.8. Abstraction and Filtering

A key functional feature needed in a successful dynamic analysis tool is an automatic or semiautomatic way of efficiently reducing the size or a trace so it can be more easily manipulated and understood. Automatic filtering and abstraction features are usually tightly coupled with the visualization tools that implement them. It seems that there is an agreement about the importance of such features in reducing the information overhead and most of the tools support these features. SEAT, for example, enables the user automatic filtering while ISVis enables the analysts to use wildcards to formulate sophisticated queries. ALMOST allows users to express queries that contain criteria by which elements of the trace are removed. Trace filtering techniques operate on the execution traces independently of any visualization scheme.

4. Summary

In this paper, we have surveyed and analyzed a wide range of trace visualization tools to understand
their common features and their differences. The study can also be used by software engineers to understand the capability of this class of tools.

Our study shows that these representation methods have a number of characteristics, which can be classified according to the following criteria:

[C1] Quality of decision making: It is determined in term of accuracy and accessibility of values for items in the trace. It also refers to the possibility to look up the precise value of specific attributes of item in the trace using a presentation method.

[C2] Speed of decision making: It is defined in terms of how fast the user can find general feeling about the data using presentation methods.

[C3] Ability to contain the data in one screen: It is concerned with visualizing data while utilizing screen real-estate efficiently.

[C4] Comprehension of high-level trends: The possibility to spot high-level trends and anomalies easily. It also determines if the representation methods in a tool can show how the trace elements and their attributes are distributed.

[C5] Comprehension of correlations: It is used to asses if a presentation method can show potential correlation between different measures of trace content that are not related to one another directly (e.g., they are distributed differently in time).

[C6] Comprehension of relations: It is used to assess if a presentation method can show relations between different trace items.

[C7] Interactivity: It is used to determine if the visualization enables the user to change graphic rendering parameters and to interactively control the presentation to clarify specific aspects of the visualization.

[C8] Ability to show dynamic behaviour: It is used to identify the presentation methods that support replaying the traced scenario by browsing the execution trace elements (e.g. using animation techniques).
### Table 4. Classification of tools based on the visualization criteria

<table>
<thead>
<tr>
<th>Tool</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
<th>C7</th>
<th>C8</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALMOST</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>❌</td>
<td>✔</td>
<td>✔</td>
<td>❌</td>
</tr>
<tr>
<td>AVID</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Bloom</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Collaboration Browser</td>
<td>❌</td>
<td>❌</td>
<td>✔</td>
<td>✔</td>
<td>❌</td>
<td>✔</td>
<td>✔</td>
<td>❌</td>
</tr>
<tr>
<td>EVOLVE</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>EvoSpaces</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>EXTRAVIS</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Gammatella</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Helix Cone Tree</td>
<td>❌</td>
<td>❌</td>
<td>✔</td>
<td>✔</td>
<td>❌</td>
<td>✔</td>
<td>✔</td>
<td>❌</td>
</tr>
<tr>
<td>Hyades</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>ISVis</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>JInsight</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>JProfiler</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>NestedVision3D</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>ORCA</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>OSE</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Ovation</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Paragraph</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Program Explorer</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Scene</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>SEAT</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Shimba</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>TraceVis</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>VET</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>VisTrace</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Vtune</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
As shown in Table 4 almost all of the trace visualization tools support visualizing high-level trends which makes it the most common characteristic. This is not surprising as providing higher level of abstraction of traces is one of the most important goals for trace visualization tools. Similar to visualization tools in other domains, Interactivity is the second characteristic satisfied by most of the trace visualization tools. The reason is that the users need to be able to control the presentation to clarify specific aspects of the visualization. We also found that the representation methods in about 90% of the studied tool have the ability of visualizing gradual changes that happen through the execution by automatic or manual stepping in the visualized execution trace. This enables the user to effectively animate the execution. A similar majority of the tools can show the relations (e.g., containment, caller/callee) between trace elements (or properties) which can play an important role in basic understanding of the software system. Moving on to correlations this percentage drops to about 45% of the tools.

The ability to contain the trace data in one screen is a both important and challenging in trace visualization tools. Only 12 of the 26 studied tools are mentioned to have this characteristic. This can be explained by the screen real-estate problem defined as fitting large amounts of information into a small space. Considering the time spent on maintenance and understanding the system under maintenance, speed of the decision making becomes an important factor. About 70% of the tools aim at improving the speed of decision making. However, only 6 of the 26 tools use representation methods that can support precision and quality of decision making. This may have several reasons among which the fact that precision tends to decrease as more abstraction is used. This abstraction could happen as part of data preparation through the representation due to limited space.

Graph-based methods turn out to be the most common representation method used by trace visualization tools (about 85% of the tools use this method). This may be explained by the accessibility of standard packages (for graphs, sequence diagrams, etc.). Furthermore, we observe that half of the surveyed tools employ tables and charts for visualization. We also note that almost all of the tools that use tables and charts for visualization also use one or more other representation methods. This shows that traditional representation methods should suffice for trace visualization when used in conjunction with other representation methods. In general, more than 70% of the tools use more than one representation method for visualizing traces.
## 5. References


[Tex95] Texas State Auditor's Office, Data Analysis: Displaying Data - Graphs, Methodology Manual, rev. 5/95


