Exploration and Visualization of Large Execution Traces

Lianjiang Fu

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements for the degree of

Master of Computer Science

Under the auspices of the Ottawa-Carleton Institute for Computer Science

University of Ottawa
Ottawa, Ontario, Canada
June 2005

Abstract

Trace analysis is one of major methods in dynamic analysis of reverse engineering. Runtime information contained in traces can reveal extensive relationships of different communicating entities that would be less obvious if only static analysis of source code was performed. Traces can be of better assistance to understand a system when they display only high-level information– showing the essence of the behaviour of a system, as opposed to large masses of detail. However, most existing trace exploration systems cannot gracefully handle the size explosion problem that normally occurs, especially when trace data is captured and gathered over a long period.

One step towards dealing with large traces is representing it using a schema developed in our laboratory called "Compact Trace Format" [1], where the whole trace is converted from a tree structure to a graph representation and only distinct nodes exist in it. Various filtering algorithms can then be applied to this model, and as a result, some nodes that are of less interest to a user can be filtered out and only those that help in understanding will be displayed.

In this thesis we have done three things: Firstly we implemented a tool based on the above approach. Secondly, as part of the tool we implemented a dynamic trace-loading scheme allowing a user to navigate through a large trace more rapidly. Instead of loading a whole tree for exploration, the loading algorithm only retrieves the part of the tree needed by the current display window. With this method, not only can the size explosion be conquered, but performance and responsiveness requirements can also be achieved. The third item achieved is to implement a special tree widget as part of the tool. This extends the UI of standard tree widgets, and is aimed to help users quickly navigate the trace and further aid them in understanding part of the system in which they are interested.

Acknowledgment

To Yuchuan and Rainey.

My special thanks to my supervisor, Dr. Timothy C. Lethbridge, for his insight, patience, and encouragement.

Thanks to the reviewers for their valuable feedback and other members of Knowledge Based Reverse Engineering Group (KBRE) for their sparkling ideas.

I also want to extend my sincere appreciation to people who participated in the user test. Without their invaluable input, this thesis would never have been completed.

QNX Software Systems and the National Capital Institute of Telecommunications provide funding for this research. I want to thank them for their continuous support.

Table of Contents

List of Figures

List of Tables

List of Acronyms

2D	Two Dimensions
3D	Three Dimensions
ARE	A Reverse Engineering Tool
ASCII	American Standard Code for Information Interchange
CPU	Central Processing Unit
CTF	Compact Trace Format
DAG	Direct Acyclic Graph
DOM	Document Object Model
EMF	Eclipse Modeling Framework
GUI	Graphical User Interface
GXL	Graph eXchange Language
HTTP	Hyper Text Transfer Protocol
IDE	Integrated Development Environment
I/O	Input/Output
JVMTI	Java Virtual Machine Tool Interface
JVMPI	Java Virtual Machine Profiler Interface
MSC	Message Sequence Chart
RAC	Remote Agent Controller
RCD	Reusable Component Descriptions language
SEAT	Software Exploration and Analysis Tool
SDDF	Self-Defining Data Format
SUT	System Under Test
SVG	Scalable Vector Graphics
SWT	Standard Widget Toolkit
TCP	Transmission Control Protocol
TDL	Trace Description Language
UI	User Interface
UML	Unified Modeling Language
VARE	Visualization Architecture for REuse
XML	eXtensible Markup Language
XMI	XML Metadata Interchange
XTE	eXtensible Trace Execution language

Chapter 1 Introduction

The domain of this research is the field of software reverse engineering; in this field, relevant information is extracted from the system and presented to a software engineer who needs to carry out some maintenance tasks to an existing system. To facilitate understanding of a system, two methodologies, static analysis and dynamic analysis are often used. We focus on trace analysis, a form of dynamic analysis.

Trace analysis is a form of dynamic analysis used to understand the behaviour of complex systems. However, the size of traces often becomes an obstacle because of the time and the memory required to process them, as well as the time required by software engineers to manipulate them.

Our research will investigate how traces can be explored by a software engineer to help him or her understand the subsystem under investigation.

1.1 Motivation

During our interactions with software engineers in QNX[1], problems dealing with large traces arise. Traces are generally captured over a relatively long period of time and have millions lines of data representing message sends or procedure calls. Handling these traces can not only take a long time and require very large amount of physical memory; but also tend to overwhelm software engineers visually due to the sheer volume of data. Also large traces make navigation actions, such as zooming and panning, very slow [2].

A similar situation also exists in other contexts where there is a requirement for handling large data sets. For example, in the biomedical area, large biological trees need to be effectively displayed, compared and identified [3]. This thesis will explore several issues,

[1] QNX Software Systems sponsored this research in conjunction with the National Capital Institute of Telecommunications.

primarily related to the user interfaces of tree exploration widgets, which could perhaps also be useful in these other contexts.

Based on the aforementioned observations, the main research objectives of this thesis can be summarized as follows:

- Finding effective techniques to deal with exploring large traces.
- Designing specific user interface controls to visualize and explore large traces.
- Implementing an appropriate tool to help software engineers understand traces.
- Validating the techniques and the tool through a series of studies.

1.2 Related Work

1.2.1 Program Comprehension

Program comprehension, also known as program understanding, is a central activity in software maintenance [4]. Before a software engineer can make changes to an existing system, at least part of the system needs to be understood. In fact, correct understanding is necessary for various tasks including: fixing the system's defects, code optimization, reuse, and legacy system migration. However, human understanding is a complicated learning process and is less studied because of its multi-disciplinary characteristics [5]. To do a good job of developing tools for program comprehension, it is necessary to investigate how the mental processes of software engineers work when carrying out maintenance tasks, and then to build tools that facilitate these mental activities. We must also evaluate any resulting tools with real users.

1.2.2 Processing Large Data Sets

Many visualization systems work well with a moderate amount of data, but they do not scale well to extremely large data sets [6]. Not only does data loading become slow, but data processing and the computation of visualizations also take longer to complete [7]. Zooming a detailed view out to a global level view can be used to identify useful application structures. However, this can become ineffective if response time becomes too slow.

Many areas, such as medicine, and data mining also have to deal with large data sets [3, 8], and have developed sophisticated special-purpose tools with adequate response time. Many tool developers, however, want to build applications using the standard widgets available in GUI toolkits available with programming languages (for example, Swing in Java, or SWT in Eclipse). The standard list or tree-oriented widgets were not, however, designed to deal with truly massive amounts of data. A major objective of this thesis will be to overcome this limitation.

1.2.3 Visualization Techniques

Visualization has been proved to be effective in helping human understanding [9, 10]. As a result, a plethora of visualization techniques exist in various contexts. We investigated different visualization techniques and how they can be used in trace visualization. However, we will selectively introduce those techniques that are widely used in the area of reverse engineering, particularly focusing on tree-structured data as found in traces.

1.3 Thesis Contributions

Our research activities have been focusing on effective visualization and navigation of very large software execution traces. To conquer the size explosion of large traces, a dynamic data-loading algorithm has been developed which only retrieves nodes needed for the current window as a user navigates through the tree. Instead of generating the whole tree for visualization, the dynamic data loading algorithm will only retrieve and render those tree nodes that are required in the current window. A new tree widget that displays this dynamic tree has also been constructed. This gives a similar look and feel as the traditional tree widget but additionally supports quick filtering of nodes so the user can more effectively explore the traces.

In this thesis, we have achieved the following:

- We have demonstrated the effectiveness of a new tree browsing capability in Java that allows for only those parts of the tree that are currently needed to be managed by the user interface.

- We have shown the effectiveness of a tree browser UI that, in addition to the standard + and – icons to expand and collapse tree nodes, has a ~ icon which can show sub items that are partly hidden by some filtering algorithms.

- We have identified the requirements for building an effective reverse engineering tool and developed a tool called SEAT (Software Exploration and Analysis Tool) that addresses the requirements. We also conducted several studies to evaluate the usefulness of the tool.

Our research can be of interests to several groups of people:

- Software engineers can use the trace analysis tool SEAT to help understand the dynamics of the analyzed system.

- Software designers who need to circumvent traditional user interface widgets when facing large data sets can use the dynamic loading technique and the tree widget in user interfaces they are designing.

- Individuals in disciplines other than computer science who need to dynamically process large data sets can also use the facilities we have developed.

Our work may also contribute to those who conduct usability testing of novel user designed widgets and evaluate the benefit of new widgets vs. existing ones.

1.4 Thesis Outline

The thesis will begin with a background literature review in Chapter 2. Four areas related to our research are surveyed, including reverse engineering, program comprehension, visualization techniques of large data sets, and trace exploration tools.

In Chapter 3, we will discuss the challenges with dealing with large traces, and focus on requirements for dynamic data loading and trace.

In Chapter 4, we will identify requirements for building reverse engineering tools and the design of SEAT, a tool encompassing the new widget that addresses the challenges described in Chapter 3.

In Chapter 5, we will explain in detail how the dynamic loading algorithm works.

In Chapter 6, we will describe the design of a special user widget with additional features that support trace exploration using the dynamic loading methodology.

In Chapter 7, we will present the evaluation of our tool through user testing.

We will conclude in Chapter 8 and present suggested future work.

Chapter 2 Background

2.1 Introduction

In this chapter, we will discuss the following areas that are related to this thesis:

- Reverse engineering
- Program comprehension
- Visualization techniques for large data set
- Trace exploration tools and formats

2.2 Reverse Engineering

Chikofsky provided the following definition for reverse engineering [11]:

″Reverse engineering is the process of analyzing a subject system to identify the system's components and their inter-relationships and to create representations of the system in another form or at a higher level of abstraction″.

Reverse engineering investigates techniques and tools to help software maintenance. It helps software engineers who explore the software systems determine where modifications should be done. In order to successfully carry out maintenance tasks, the target system or at least part of it must first be understood [12]. As a result, reverse engineering is summarized as focusing on "understanding the code." [13]. However, the code does not contain all the needed information for understanding the system [12]. Other knowledge, such as architecture and design tradeoffs often only exists in the minds of people since they are typically not well documented. Therefore, reverse engineering techniques are needed to build a descriptive view of the system at various levels of abstraction.

In reverse engineering, there are often two complementary ways for understanding an existing system: static analysis and dynamic analysis.

Static analysis deals with source code, and it is the process that evaluates a system or component using code, structure, architecture and documentation without executing the program [14]. Static analysis includes techniques such as manual inspection, automated program analysis and data flow analysis [15]. The advantage of static analysis is that it can provide information about software by giving objective measurements [15]. Control flow, complexity metrics and class relations can all be extracted. The static model extracted can also be used to verify that architectural design guidelines are followed [16].

While static analysis does not necessitate actual execution of the software, dynamic analysis involves running the system formally under controlled circumstances and results are often known and expected before running the system [17]. Dynamic analysis generally captures various kinds of run-time information, with the goal of understanding the dynamic characteristics of a design (execution sequences and relative time ordering of events) [18]. Profiling is one of the techniques often performed in dynamic analysis: this involves periodic sampling of what is being executed. Tracing on the other hand involves recording all the events that occur in the system; the events recorded could be statement execution, routine calls or inter-process message sends.

Dynamic analysis tends to be less favoured than static analysis because of the difficulties in information gathering, the size of the information gathered, and consequent difficulties in the interpretation of this information. Collecting dynamic information often needs special settings in source code or the environment; these can be cumbersome in many situations. Also the data collected can be extremely large which makes interpreting it and extracting useful information a daunting task. As a result, compared with pervasive static analysis tools, such as those available in IBM's Rational products, dynamic tools tend not to be widely used even though they could be a great aid to understanding. The causes of bugs and system irregularities may be most easily found through dynamic approaches, especially in legacy systems that are process centric rather than data centric [19]. Therefore we need ways to allow software engineers to somehow find and absorb dynamic information despite the aforementioned difficulties.

2.3 Program Comprehension

Program comprehension is the process of understanding a program with the purpose of performing further tasks such as fixing bugs, refactoring code, and porting code to different platforms. Program comprehension is important for several reasons: most development projects do not involve writing a program from scratch but instead start from existing components and frameworks. High turnover of large projects often results in introducing developers that are new to the systems and these developers need to familiarize themselves with the structure of the new systems before they can make changes or add new functionality. Lack of documentation or poor documentation, such as outdated ones that cannot reflect the latest design prevent developers from effectively comprehending the existing systems and force them to dig into source code to understand how a system is really designed. Other activities such as debugging are also comprehension-related.

2.3.1 Understanding Complex Systems

Understanding existing software systems is known to be a key issue in the area of software reverse engineering. Compared with understanding simple “programs”, software engineers need to comprehend large complex software systems, which include not only data and algorithms, but also components and architectures. Knowledge of architectural concepts is key to understand legacy software and in designing new software. Such concepts include subsystem structures, layered structures, inheritance hierarchies, etc. However, several difficulties arise in understanding: these include lack of abstraction, lack of documentation, mixed programming languages, the scaling problem (difficulties that increase with size in a greater-than-linear manner) and unclear tool semantics (inability of tools to address understanding issues, and conflicts of concepts of different tools in addressing similar problem) [20].

The good news to maintainers is that we do not need to understand the whole system before changes can be made. Through the observation of software engineers’ work practices, Lakhotia concludes that changes are often within a relatively small amount of functionality at a time and therefore it is not always necessary to understand the design of the whole system in order to make correct changes [21]. From the perspective of traces, this means that only events that correspond to a programmer's interest need to be further studied.

2.3.2 Human Cognitive Model

Exploring how human mental processes work can help us understand program comprehension better. The process of how humans understand code has been extensively investigated [22]. Often software engineers use programming knowledge, domain knowledge and various strategies to understand a new piece of program [23]. For example one might rely solely on source code to extract syntactic abstractions. Researchers have proposed several models for program comprehension and have concluded that software engineers will try to construct mappings through mental models during a program understanding process [5]. A mental model is defined as "an internal, working representation of the software under consideration" [5].

Cognitive models are often used to represent the mental processes software engineers use and the interactions that happen between humans and machines. They correspond to the mental models used by software engineers as they form a mental representation of the program under study. In research about how human beings acquire knowledge, several cognitive models have been proposed:

1. In the top-down model the software engineer formulates hypotheses and decomposes a system into subsystems to form a deeper and deeper understanding of the system and its functionalities. Thus the mental model is composed of a hierarchy of goals and plans and it represents knowledge about an application domain. It is often used when code is familiar [24].

2. Brooks extended the top-down model and proposed a theory that describes programming as a process of constructing mappings from a problem domain to the programming

domain, with some possible intermediate domain [25]. The theory states that, “Comprehending a program involves reconstructing part or all of these mappings”. The reconstruction process is driven by creation, confirmation, and refinement of hypotheses.

3. The bottom-up approach begins with reading source code and mentally constructing a series of higher-level abstractions called chunks by synthesizing details. A program model is constructed through clustering basic program text to build control flow level abstractions [26].

4. Mayrhauser proposed an integrated model based on the previous models and defined it with four components: the top-down, situation and program models and the knowledge base [27]. In this model, the knowledge base which contains entities like text structures, plans, hypothesis, and rules of discourse, is necessary to construct the other three models. The top-down model is invoked when the type of code is familiar, and the model represents high-level functionality of a subsystem. Whereas a program model is built from bottom up when the code is new and the model mainly consists of disjointed flow-level functionality synthesized from code. The situation model is an intermediate model representing functional knowledge and is also constructed from bottom up but is higher than the program model. Typically, a chuck of code will correspond to a functional description in the situation model. The integrated model often exploits all the three sub models and they may be active at any time during the comprehension process. For example, when understanding a piece of the system when building a program model, a software engineer may find some code that indicates a hypothesis, and will therefore change to the top–down model. To support the hypothesis, several sub-goals may be generated leading to switches to the other two models. Two dynamic elements are often used in this process: chunking to create new, higher-level structures from lower level chunks, and cross-referencing to relate two different levels of abstraction together.

The consensus in the program comprehension community is that the integrated model is closer to how human mental processes work.

2.4 Visualization Techniques for Large Data Sets

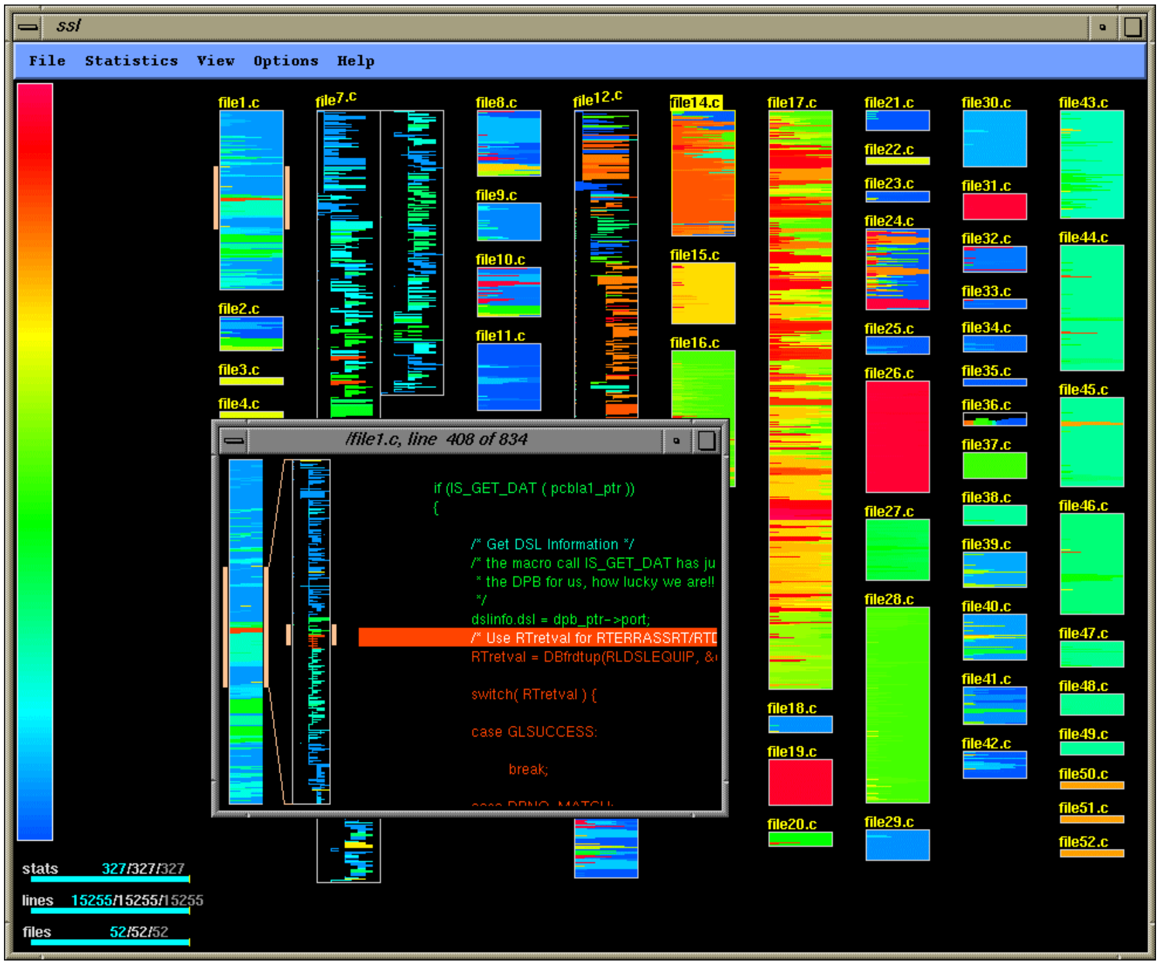
Price defined software visualization as "the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software" [28].

Visualization is a heavily employed technique in software engineering. Generally, it uses colours, shapes, space and text to provide the user with a model of the software. For example, system architecture can be depicted using class diagrams and interactions can be represented as sequence diagrams. Hendrix concluded that effective software visualizations could provide measurable benefits in program comprehension [29].

To satisfy the need to visualize large data set, different visualization techniques have been proposed.

2.4.1 SeeSoft

SeeSoft [30] is primarily used to visualize text-based files such as source code (See Figure 1). It maps each row of text into a line with the colour denoting a statistic of interest. The statistics can be any attributes derived for the source, such as revision history or execution frequency. The main advantage of SeeSoft is that it can dramatically reduce the size of the representation so interesting visual patterns may be found and these patterns often relate to attributes that are repeated in data.



```
if (IS_GET_DAT ( pcbla1_ptr ))
{

        /* Get DSL Information */
        /* the macro call IS_GET_DAT has ju
         * the DPB for us, how lucky we are!!
         */
        dslinfo.dsl = dpb_ptr->port;
        /* Use RTretval for RTERRASSRT/RTD
        RTretval = DBfrdtup(RLDSLEQUIP, &

        switch( RTretval ) {

        case GLSUCCESS:

                break;

        case DBNO_MATCH:
```

Figure 1 SeeSoft showing editing history of source files

Because 2D views of SeeSoft can only express limited attributes and relationships, 3D extensions have been proposed to support visualization of multiple attributes at the same time [31]. Elements of visualizations include polycylinders, height, depth, colour, and position. User interactions support navigation, scale and rotation in each direction.

2.4.2 Focus + Context Visualization

Focus + Context [32] visualization is another approach that is aimed to maximize the use of the display resource (See Figure 2). It tries to present all the data in one screen to give the user an overview, but at the same time, distorting the representation so that data in focus will be displayed in more detail while data at the edge of focus is blurred. One example of this technique is the hyperbolic tree browser [33].

Focus + Context supports automatic calculation of the degree of user interest. If a node is clicked or expanded, that node becomes the focus node; the degree of interest of sibling nodes is calculated according to their distances from the focused node. The technique uses different node sizes to show the importance of nodes.

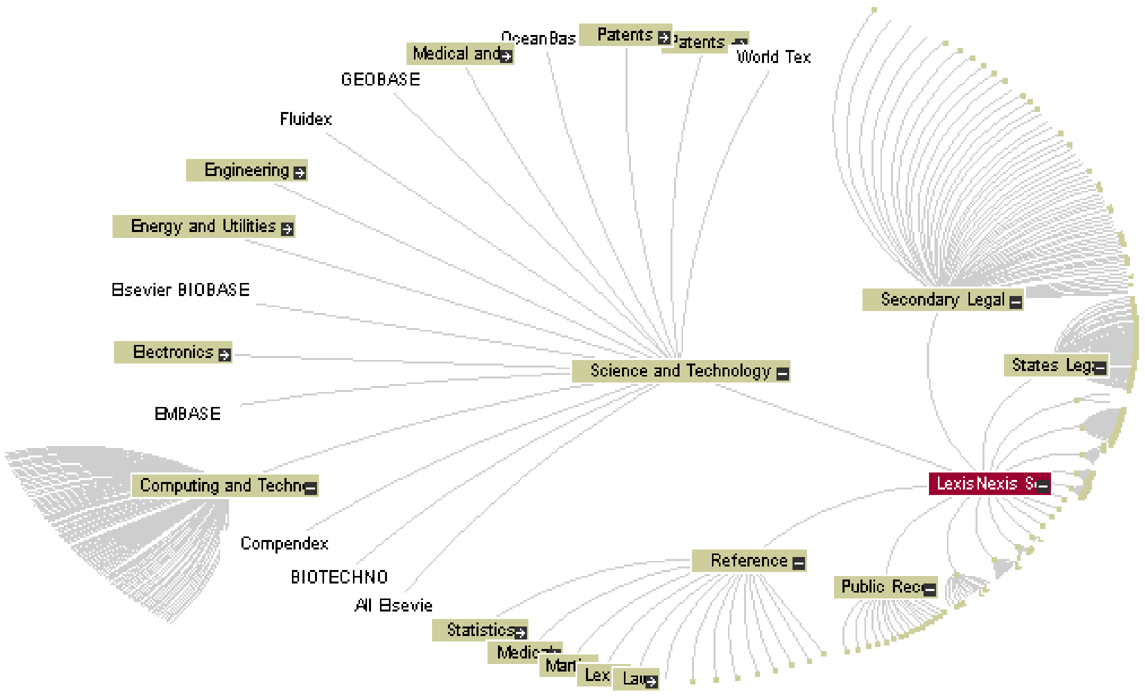


Figure 2 Focus + Context visualization

When displaying a large volume of information, the drawback is that information distant from the focus will become so small that it cannot be perceived. A second concern is speed: a change of focus results in a need to recalculate and redraw the whole layout [34].

2.4.3 Multiple Views

The Multiple Views technique uses two or more views to visualize various aspects of the same entity [35]. These views are often correlated and synchronized to provide the user with more visual aid. A global view is often used to display system-level information and different auxiliary views can display various facets of interest through synchronization with the global view. By displaying the data in multiple ways, a user may understand the information through different perspectives [36, 37]. Also the information contained in individual views can be integrated into a coherent image of the data as a whole by the tool user. The direct benefit is improved user performance.

2.4.4 TreeMap

TreeMap [38] visualizes hierarchical structures by a space filling and dividing technique (See Figure 3). It starts with a rectangle area to denote the root node, and then in each level, children will divide the space allocated to their parent. Information in intermediate levels is not clear in this visualization. If only attributes at leaf level are of interest, this technique can be a choice.

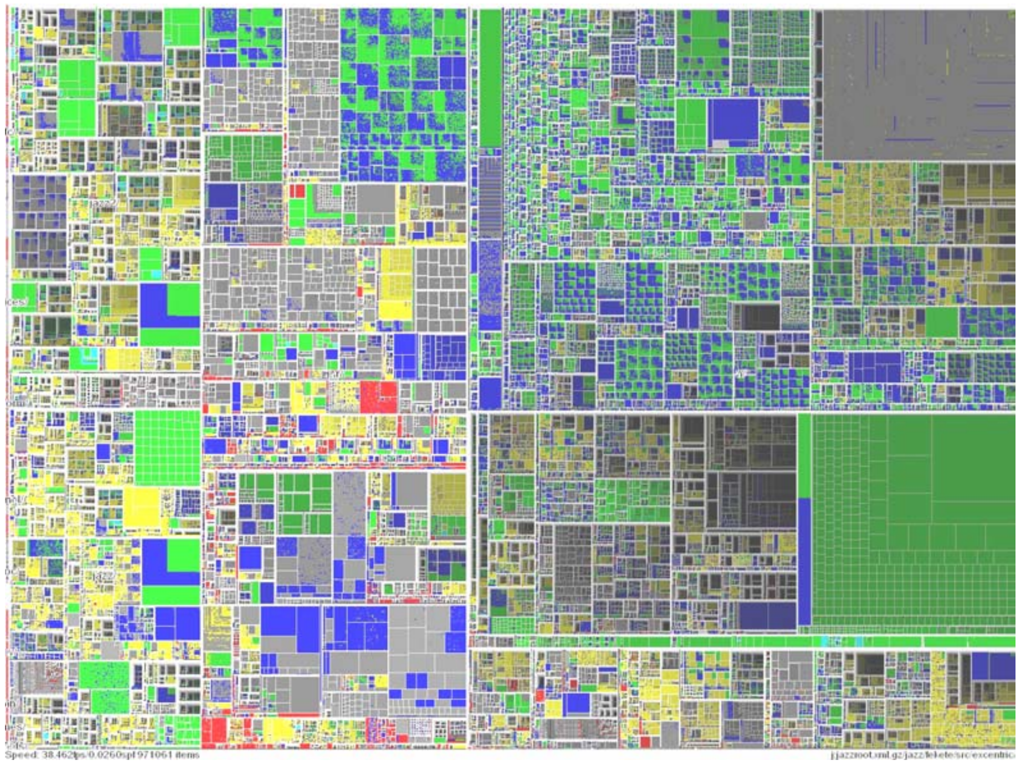


Figure 3 TreeMap visualization of 1 million items

2.4.5 ConeTree

ConeTree [39] is a three-dimensional tree representation (See Figure 4). It presents hierarchical information either horizontally or vertically. It draws one node at the apex of the cone and all the children nodes in the circular base of the cone. The children nodes are drawn in layers and overlaid in depth to minimize the visual clustering of nodes. The primary goal of the core tree approach is to present the whole hierarchy with minimum scrolling or no scrolling required. But experiments [40] showed that when the total number of tree nodes exceeds 1000, the display becomes too cluttered to be effective.

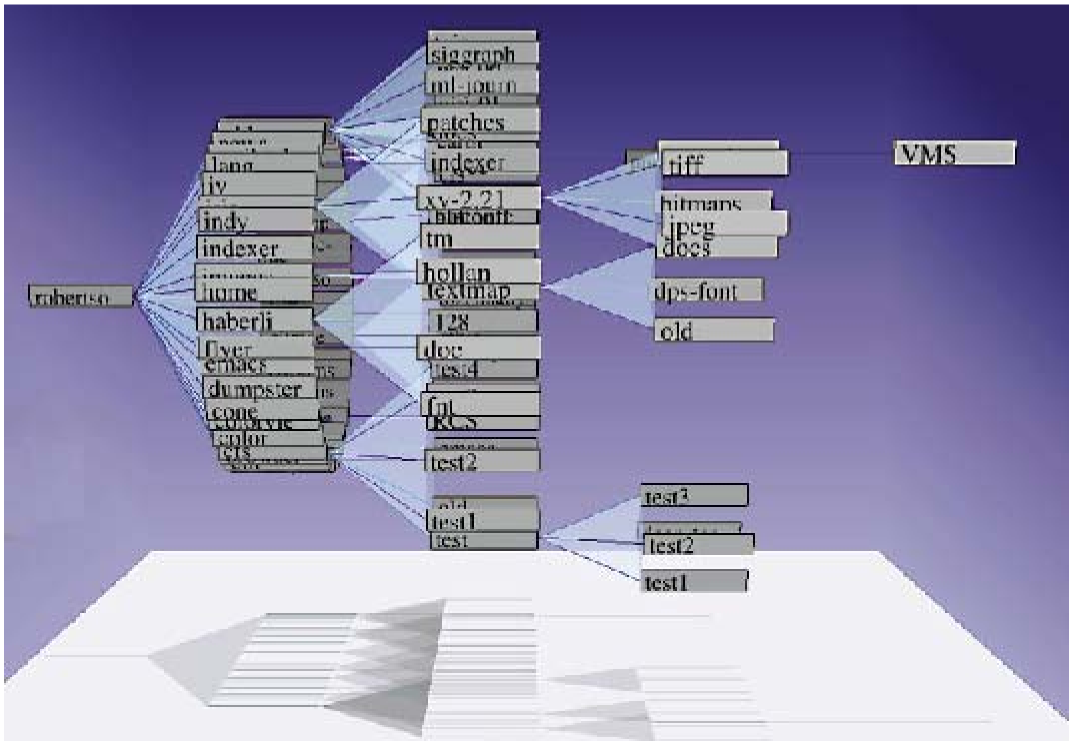


Figure 4 ConeTree visualization of a directory structure

To overcome the limitation of ConeTree, fsviz is proposed to augment ConeTree with different graphical and interaction techniques to scale to larger hierarchies while maintaining user control [41]. It makes better use of colour, shape and text to designate different types of objects. A layout algorithm based on scaling each cone, and a focus + context view is used to address the degree of interest as discussed in section 2.4.2. The process that brings some nodes to the focus and others to the edge is also animated.

The challenge for visualization using three dimensions is how to best exploit its capabilities. A consensus has not yet been reached about whether 3D is more effective and understandable than 2D [41].

2.4.6 T2.5D

T2.5D [42] is a technique to conquer the expensive animation for 3D and the space limitations of 2D (See Figure 5). It displays nodes in both highlighted and dimmed modes. The highlighted nodes are those the user is currently interested in and they are displayed in the foreground to be viewed and navigated with ease. The dim nodes are displayed in z-order to provide a 3D effect and they allow the user to get an idea about the overall structure of the hierarchy. The labels of background nodes are overlapped, but a tool-tip like popup can help ease the problem. A node can be switched between background and foreground by user selection.

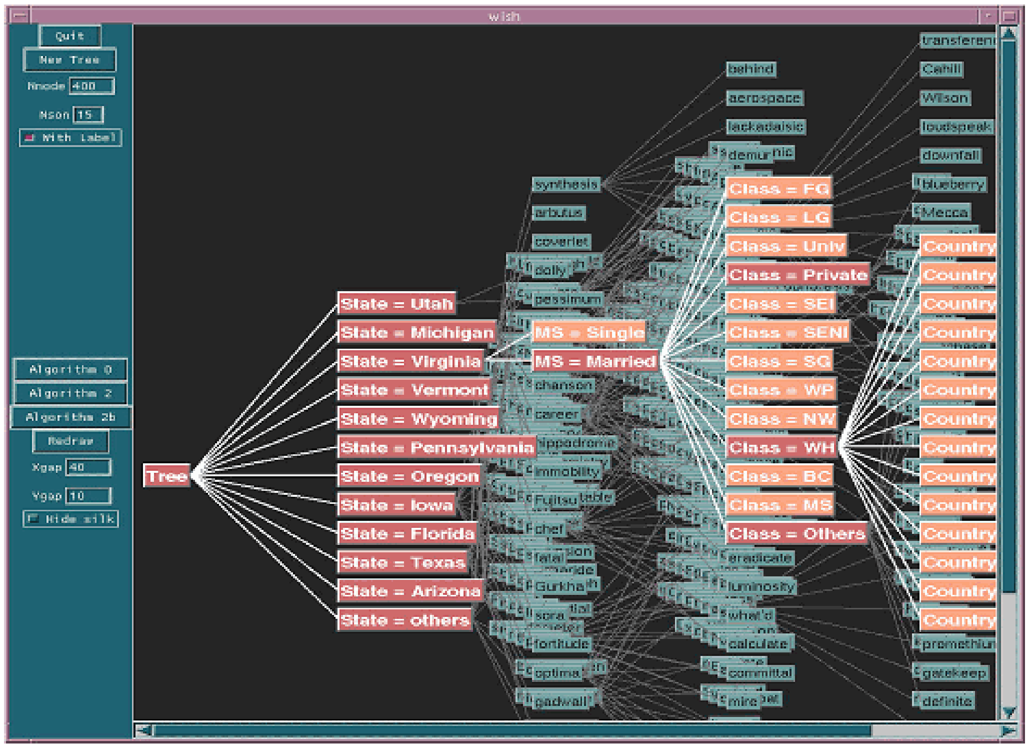


Figure 5 T2.5D visualization of large decision trees

2.4.7 Nested Graph View

Nested graph view is another way to present complex hierarchical information. SHriMP proposes nested interchangeable views to represent the organization of the software at different levels of abstraction. The nesting feature of nodes depicts the hierarchical structure of the software (See Figure 6)[43]. The primary nested view employs fish-eye techniques to provide context cues, such as in the top-down strategy. It is built on the idea that a system can be divided into subsidiary systems recursively. The division finally leads to a hierarchy with a nesting relationship of containment. The zooming interface in SHriMP incorporates the hypertext browsing metaphor so different views can be browsed through links; animation and panning are used to keep continuous user orientation when the focused view is changed.

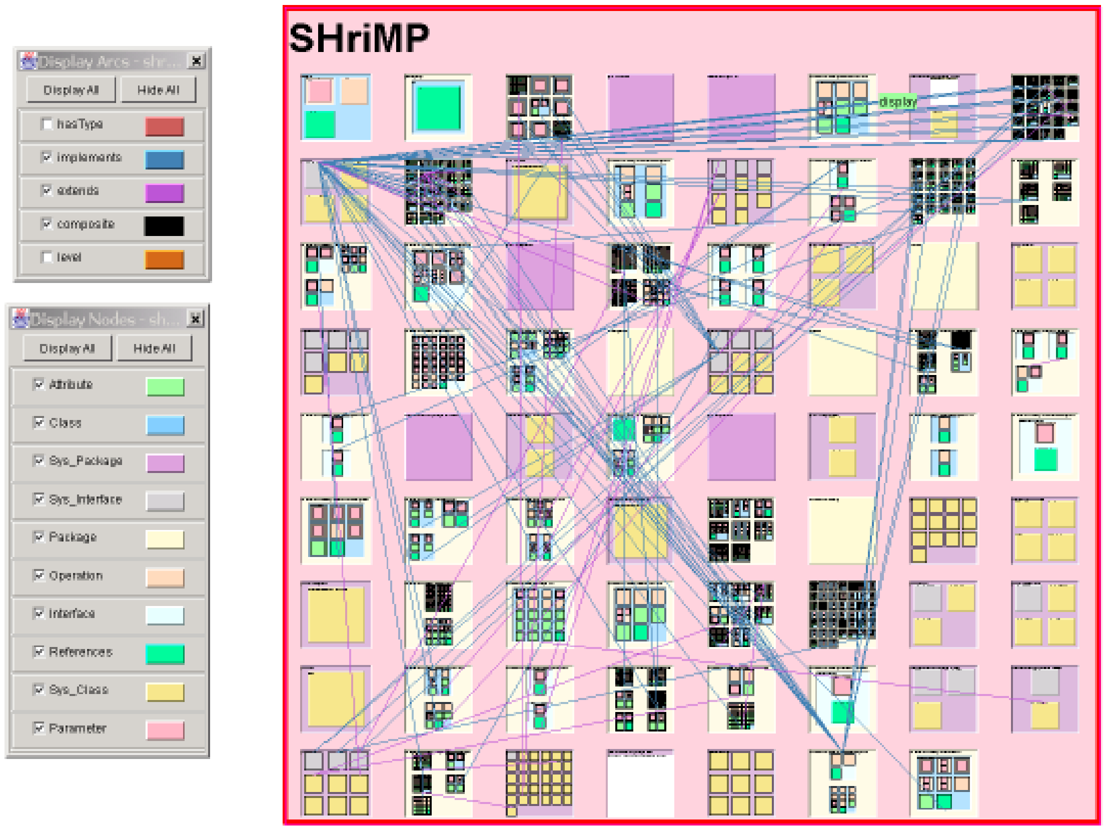


Figure 6 A SHriMP view showing the architecture of a software package

2.4.8 Animation

Animation is often used in visualization to promote human understanding. It is often used to clearly show to the user the differences between two consecutive states of the display. Animation is helpful in preserving the user's mental map between successive displays. However, proper use is important to ensure its usefulness, such as reducing the amount of information handled by the user and maximizing the pertinent information for the user's task [44]. Ideally, the picture should depict all and only the information the user's tasks require. To alleviate the cognitive effort needed to understand the animation, and to increase effectiveness of tools, animation should be complemented by automatic presentation algorithms and graphic legends to convey the visualization syntax. An example system is TKSee visualizer described in Wang's thesis [45].

2.5 Trace Definition and Context

In this section, we will describe the traces used in this thesis in more detail as well as trace analysis techniques and how they are used in reverse engineering.

2.5.1 Trace

A trace, also called a program trace, or execution trace, following definition as found in [14], is:

*"*A record of the sequence of instructions executed during the execution of a computer program. It often takes the form of a list of code labels encountered as the program executes."

Besides execution sequences, a trace can also contain relative time ordering and location of various types of events that occur during program execution [18]. The temporal information is imperative for a multi-threaded system where threads need synchronization to accomplish certain tasks. In software visualization, a trace can typically be laid on a time axis and be scrolled back and forth by manipulating a scroll bar to show different parts of data. [46]. This approach is necessary because the trace is often so large that not all data can be visualized in the same screen. The location information in a trace, such as package names, class names and method names, can be used to identify corresponding source code that relates to current trace.

Traces belong to the domain of dynamic analysis. Different aspects of complex software systems, especially runtime information, can be understood through trace analysis. Historically, the primary motivation for tracing program execution is to capture performance behaviour. The objective was to identify the parts of the program that consume the most processing time and become bottlenecks [47]. The benefit of execution traces is that once created, they can be used for multiple visualization needs from different perspectives. A subset of the information can be extracted for visualization based on given time interval instead of using a whole trace. Therefore the user of a saved trace can choose to visualize any points of interest in the trace.

2.5.2 Trace Data Collection

There are different ways to collect an execution trace without affecting the functionality of a system:

- Code instrumentation: probes are inserted into the source code, outputting information when needed. Instrumentation can be done automatically through tool support or through a built-in language facility. For example, in AspectJ, pointcuts can define where data will be gathered [48].
- Modified environment: a modified version of run time environment is used, such as a modified Java Virtual Machine, to gather data automatically at given execution points.
- Programming interface: data can be either queried or notified using an API or the underlying operating system. An example of a tool that does this is the Java Debugging Interface.

2.5.3 Difficulties in Trace Analysis

The main difficulty in dealing with traces is their size. With today's fast computers, even tracing just a few seconds of execution can result in a trace that contains millions of steps. Systems left to accumulate traces for several minutes can generate traces that are billions of lines long. It is difficult to represent these large traces in an efficient way and allow for the manipulation of them with an appropriate response time. An expected response time for today's software tools is almost instantaneous response for actions such as hiding or showing parts of the trace.

Presenting a visualization of an entire trace is also complicated. Trying to understand all that happened during the execution results in a heavy cognitive load.

2.6 Trace Exploration Tools

The approach that most trace exploration and visualization tools adopt can be decomposed into three steps [49, 50, 51, 52]:

- Gather raw trace data.
- Represent data using an internal model; this includes format conversion, event ordering, and generation of summary statistics.
- Analyze the data in various ways and present the result on the screen for the user.

These tools depend heavily on filtering to reduce the data size. For example, filtering by time, by name space, by depth of events in the call tree hierarchy, etc. They also depend heavily on visualization techniques to help understanding.

Some of the tools that are representative in dealing with large traces are discussed here.

2.6.1 ISVis

Jerding proposed the idea to save the trace as a directed acyclic graph (DAG), and his tool, ISVis (See Figure 7), belongs to the category of multiple view tools [53]. Identical sequences of calls, called patterns, are detected and represented uniquely in the DAG. In ISVis, a large data set is mapped into a small display window, called the information mural. The information mural is primarily an extension of SeeSoft and uses visual attributes such as colour and intensity to represent information density. It can be used as a global view and gives the user the context to support further browsing and searching tasks. Also patterns within a trace can be visually identified from the information mural representation. Interactions are displayed in a detailed view as a message-flow diagram.

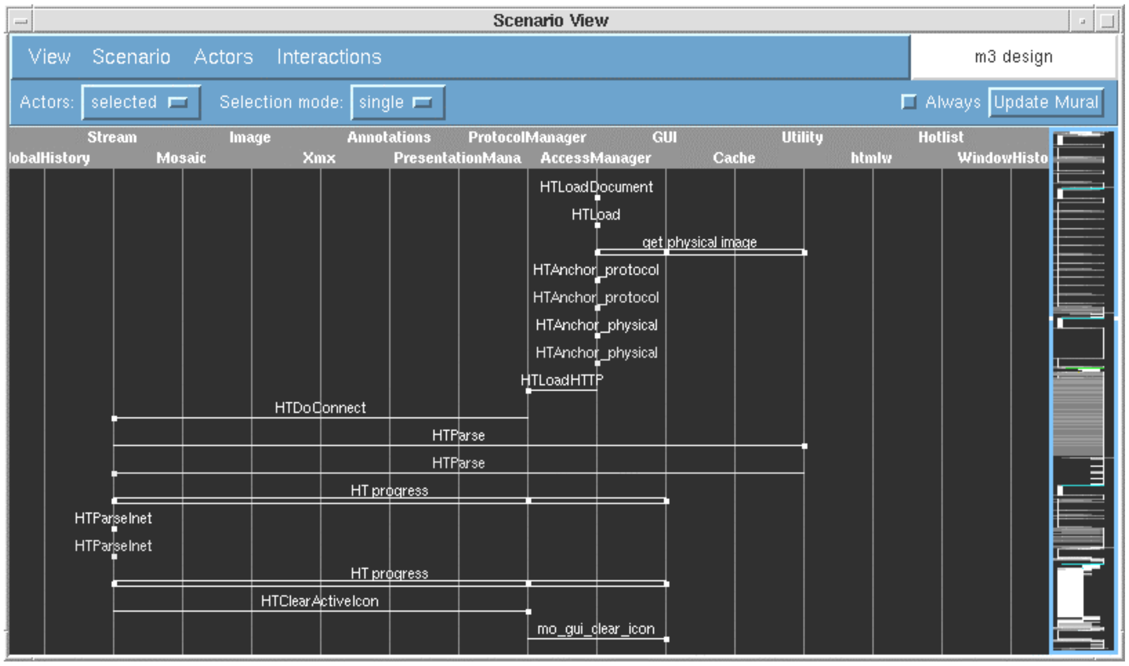


Figure 7 ISVis with a Information Mural view and Scenario view

2.6.2 Almost

The design goal of Almost [54] is to help programmers quickly get enough knowledge about the structure of a system so that they can make small to medium changes. Almost has a linear view that shows the step-like method calls of a trace. The horizontal axis of the linear view represents temporal information, while the vertical axis shows a method call and all its ancestor calls with different colours. This view supports panning, zooming and filtering. To address the space utilization problem, a spiral view is developed (See Figure 8), however the usefulness and intuitiveness of this view needs further validation.

To synchronize multiple views, time points are used. At a certain time, the focused view works as a controller and sends out synchronization information to other views connected to it. Another kind of view available is the code view. This is a SeeSoft-like view to display code structure, but it does not connect to other views.

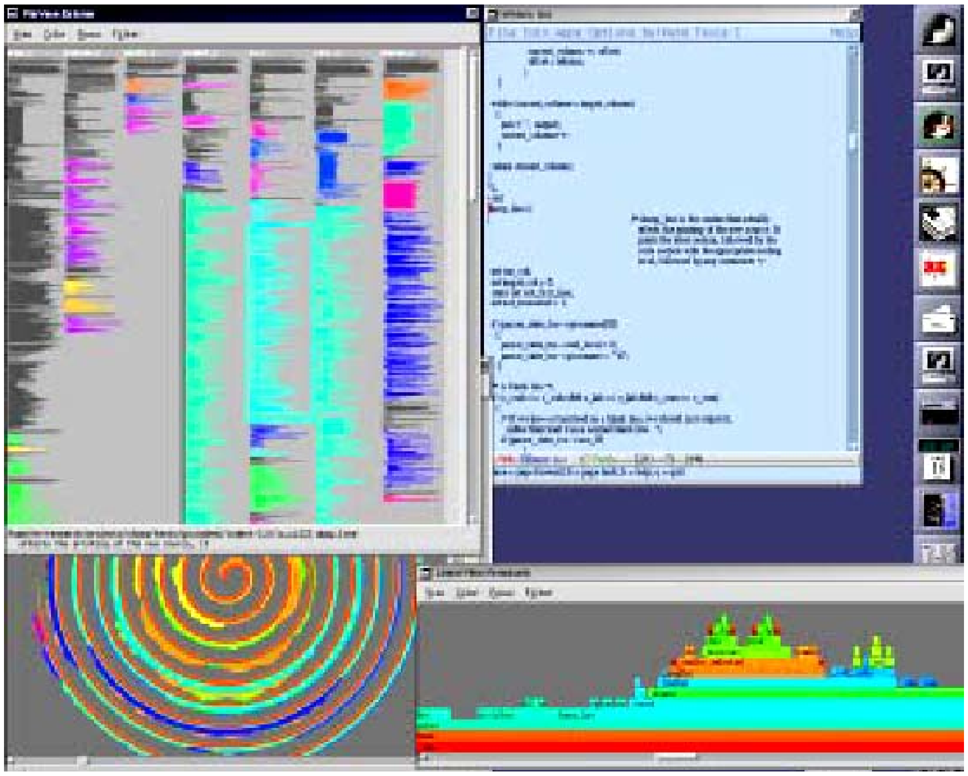
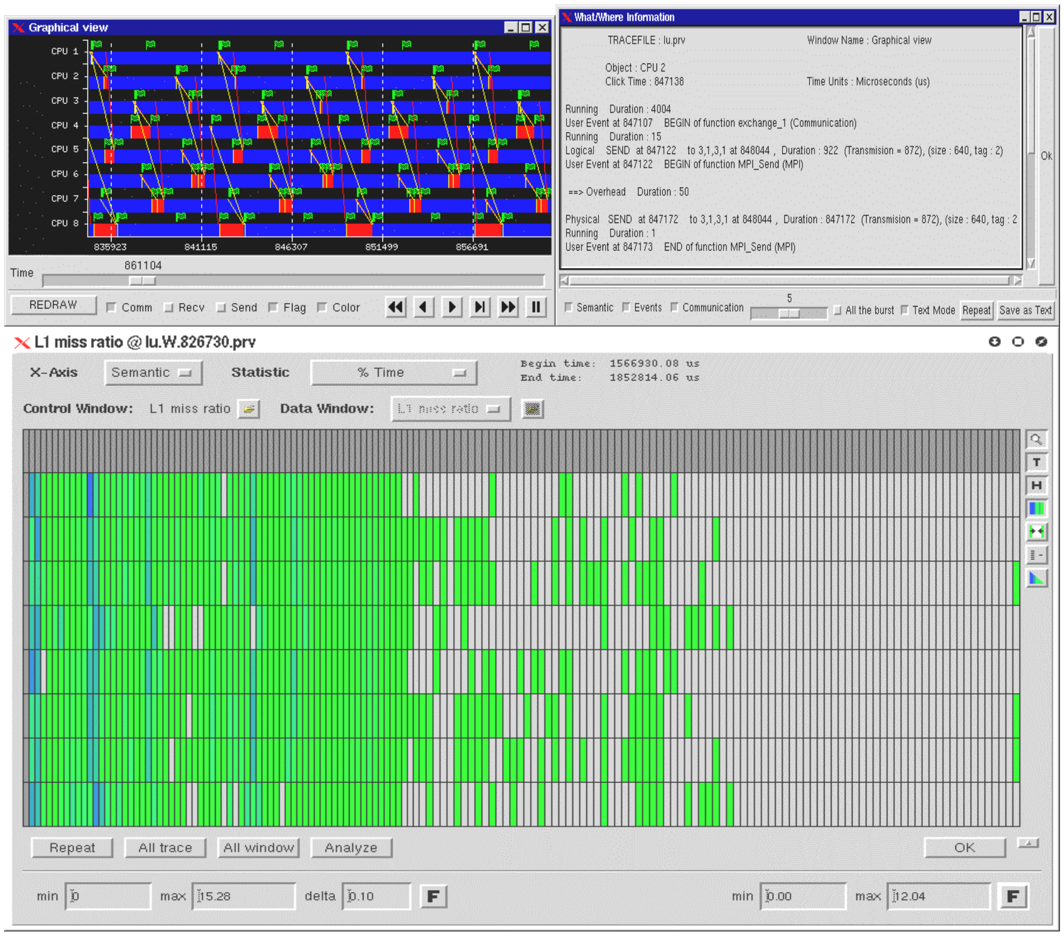


Figure 8 The Almost trace visualization tool with different views

2.6.3 Paraver

Paraver (See Figure 9) is a visualization tool used for program understanding and performance optimization for parallel environment. It is designed to address the need to have a global perception of a system with multiple CPUs and then switch to a quantitative analysis of problem details [46].

Paraver provides a minimal set of views based on the idea that a different view is only needed if it can provide a different type of information. The views include a graphic view for overall behaviour of an application through time, a textual view for extreme detail and an analysis view for quantitative data.



```
TRACEFILE : lu.prv                    Window Name : Graphical view

Object : CPU 2
Click Time : 847138                   Time Units : Microseconds (us)

Running    Duration : 4004
User Event at 847107    BEGIN of function exchange_1 (Communication)
Running    Duration : 15
Logical    SEND  at 847122    to 3,1,3,1 at 848044 ,  Duration : 922  (Transmision = 872), (size : 640, tag : 2)
User Event at 847122    BEGIN of function MPI_Send (MPI)

 ==> Overhead    Duration : 50

Physical   SEND  at 847172    to 3,1,3,1 at 848044 ,  Duration : 847172  (Transmision = 872), (size : 640, tag : 2
Running    Duration : 1
User Event at 847173    END of function MPI_Send (MPI)
```

```
X-Axis   Semantic     Statistic   % Time        Begin time:   1566930.08 us
                                                End time:     1852814.06 us
Control Window:  L1 miss ratio     Data Window:  L1 miss ratio

min 0     max 15.28     delta 0.10                          min 0.00     max 12.04
```

Figure 9 Pavarer with a process view, source code view and thread view

Paraver is designed with the capability of efficiently handling large traces and simultaneous visualization of several traces so comparisons can be made among the traces: e.g. traces of two versions of a system, or behaviour on two machines can be done. Paraver specifies a trace format and some mechanisms for how the trace records and the values will be processed in the visualization. An ASCII trace file contains records that describe the absolute time of an event in a thread of parallel code. Every record specifies the object to which it refers (indicating application task and thread) and the absolute time at which it happens. For each type of record, some additional fields can be encoded as desired by the user. These records are:

- State records include an integer value that is usually referred to as the thread state or resource information.

- Event records include a user event type and a user event value that marks the entry or exit of a code block.
- Communication records include a communication tag and a communication size that represents a point-to-point communication between a sender and a receiver.

The structure of trace file in Paraver typified data organization of trace tools. A trace file also associates with some other files which configure environments, such as colours, number of states, state labels and row labels. These files are used by the trace file to facilitate visualization.

2.6.4 Hyades

Hyades is the Eclipse Test & Performance Project with the aim to "build a generic, extensible, standards-based tool platform" for testing, tracing/profiling, tuning, logging, monitoring, and analysis [55]. Because most current trace tools are incompatible with each other, functionality reuse and tool interoperability are nonexistent. The Hyades project is organized to address this integration issue by defining a common data model based on the Eclipse Modeling Framework (EMF), data collection infrastructure, environment support and a common user interface (See Figure 10). Based on the framework support of the overall Hyades platform, Tracing and Profiling Tools sub project provides specific data collection for trace model, and different viewers for visualizing data from the model.

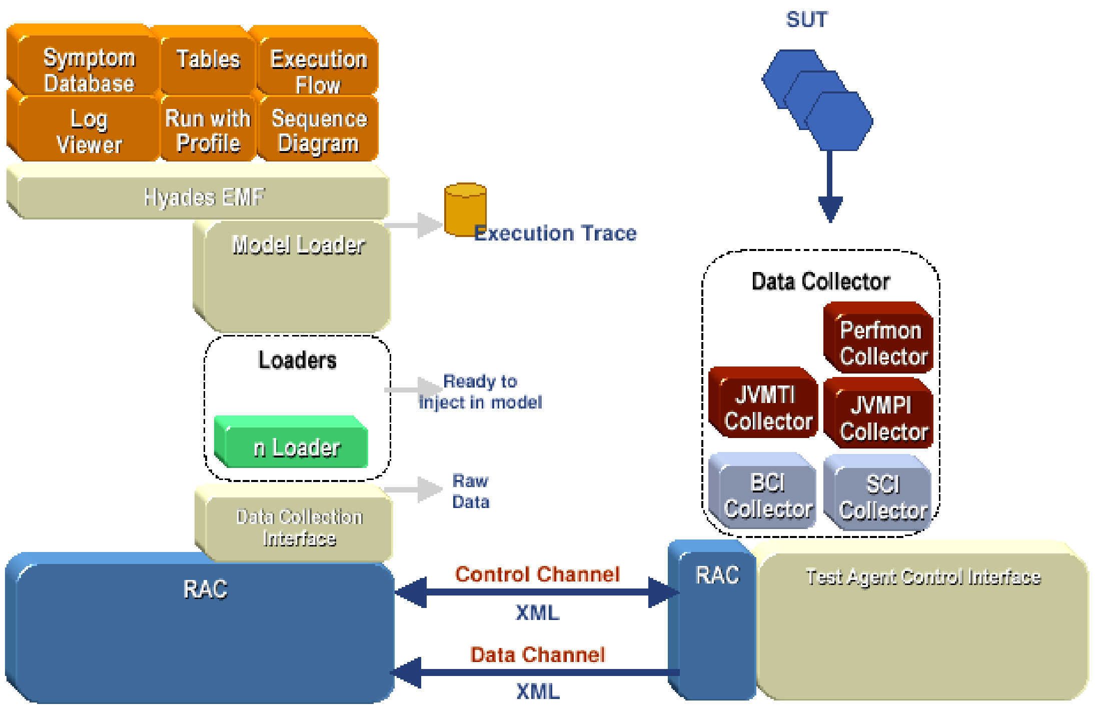


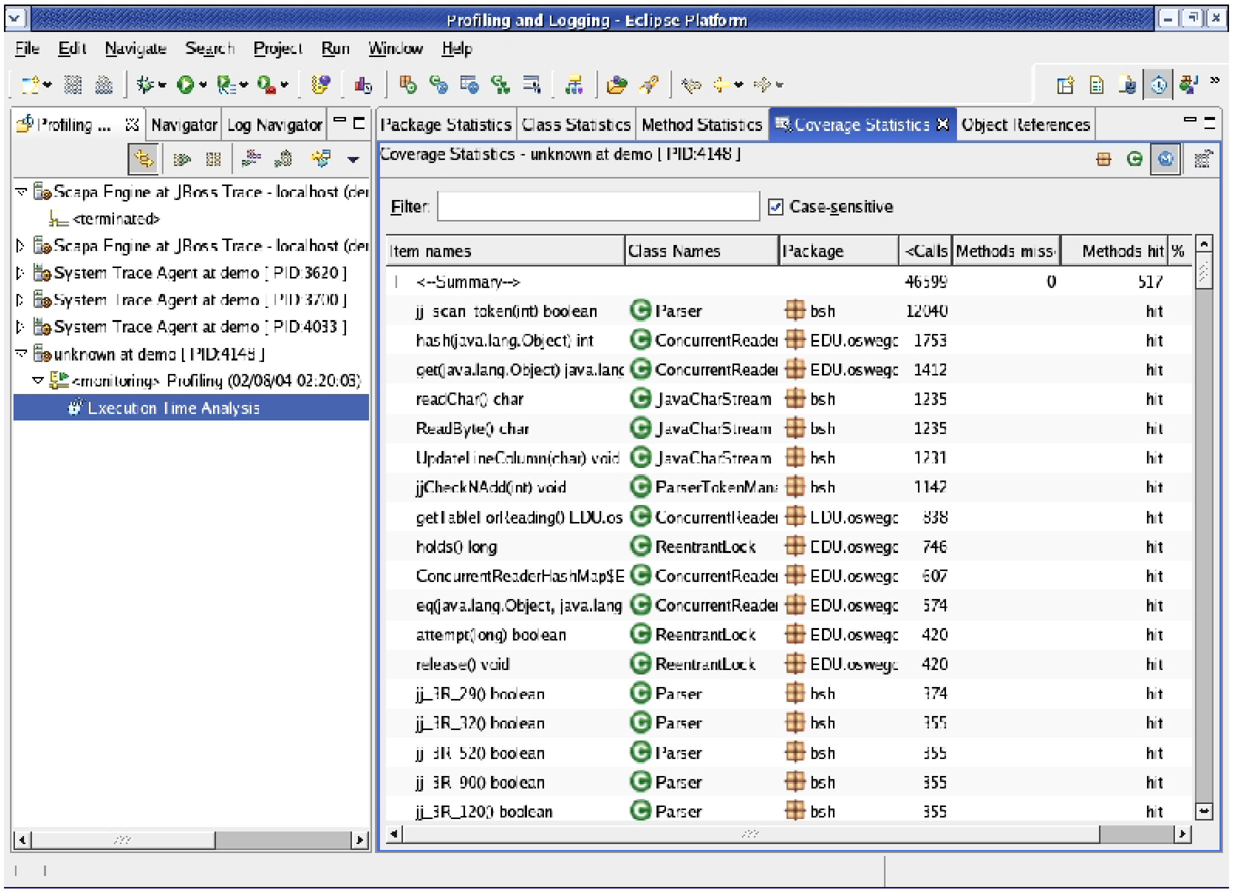
Figure 10 System architecture of Hyades

Hyades uses agents to collect data from a remote System Under Test (SUT) and serializes model data and resources to XMI by default. Remote Agent Controllers (RAC), residing on both the client and host side, allow a client to launch new process on the host side. Corresponding to the control channel and data channel in Figure 10, there are two interfaces: the test control interface and the data collection interface. Because raw trace data gathered from agents can be encoded in any format, a parser extension point is defined so users can develop and contribute their parser implementation to the platform. Therefore, regardless of trace format, relevant information can be extracted by the parser and sent to the internal trace model.

Internally, Hyades uses a set of EMF based models to store execution traces, statistical data structures, as well as test case definitions and execution histories. The trace metamodel allows capturing data within a thread, across threads, processes and even machines. It allows correlations of events and method calls across different boundaries and statistical data are also contained to eliminate subsequent calculation expenses. The goal of Hyades is to defined a common trace model and unify the proprietary trace models in one theme, and thus to improve tool interoperability.

When initially launching the tool, the user can specify filter criteria so only data from within the given scope is captured; for example, the scope may be limited to certain packages. On top of the trace model, there are various views that extract data and display them as different diagrams. These views can be used to analyze data and assist in understanding of program behaviour. The views include Statistical Views of object references, package/class/method coverage, and memory usage (See Figure 11). There are also views that display different diagrams, such as execution flow, sequence diagrams (See Figure 12). The views can also be dynamically refreshed as Hyades receives additional data from the underlying system.

Besides visualizing traces, Hyades can also generate a test case from the trace data and re-run the test if necessary. Another main area is for performance monitoring and analysis. In the same manner as the trace model, these models are also EMF based and Hyades provides a set of widgets to produce SVG based graphics for various charting styles.


Profiling and Logging - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Profiling ... Navigator Log Navigator
Package Statistics Class Statistics Method Statistics Coverage Statistics Object References
Scapa Engine at JBoss Trace - localhost (de
<terminated>
Scapa Engine at JBoss Trace - localhost (de
System Trace Agent at demo [PID:3620]
System Trace Agent at demo [PID:3700]
System Trace Agent at demo [PID:4033]
unknown at demo [PID:4148]
<monitoring> Profiling (02/08/04 02:20:03)
Execution Time Analysis

Coverage Statistics - unknown at demo [PID:4148]

Filter: ☑ Case-sensitive

Item names	Class Names	Package	<Calls	Methods miss	Methods hit	%
<--Summary-->			46599	0	517	
jj_scan_token(int) boolean	Parser	bsh	12040		hit	
hash(java.lang.Object) int	ConcurrentReade	EDU.oswego	1753		hit	
get(java.lang.Object) java.lang	ConcurrentReade	EDU.oswego	1412		hit	
readChar() char	JavaCharStream	bsh	1235		hit	
ReadByte() char	JavaCharStream	bsh	1235		hit	
UpdateLineColumn(char) void	JavaCharStream	bsh	1231		hit	
jjCheckNAdd(int) void	ParserTokenMan	bsh	1142		hit	
getTableForReading() EDU.os	ConcurrentReade	EDU.oswego	838		hit	
holds() long	ReentrantLock	EDU.oswego	746		hit	
ConcurrentReaderHashMap$E	ConcurrentReade	EDU.oswego	607		hit	
eq(java.lang.Object, java.lang	ConcurrentReade	EDU.oswego	574		hit	
attempt(long) boolean	ReentrantLock	EDU.oswego	420		hit	
release() void	ReentrantLock	EDU.oswego	420		hit	
jj_3R_29() boolean	Parser	bsh	374		hit	
jj_3R_32() boolean	Parser	bsh	355		hit	
jj_3R_52() boolean	Parser	bsh	355		hit	
jj_3R_90() boolean	Parser	bsh	355		hit	
jj_3R_120() boolean	Parser	bsh	355		hit	

Figure 11 Hyades code coverage statistics from a trace

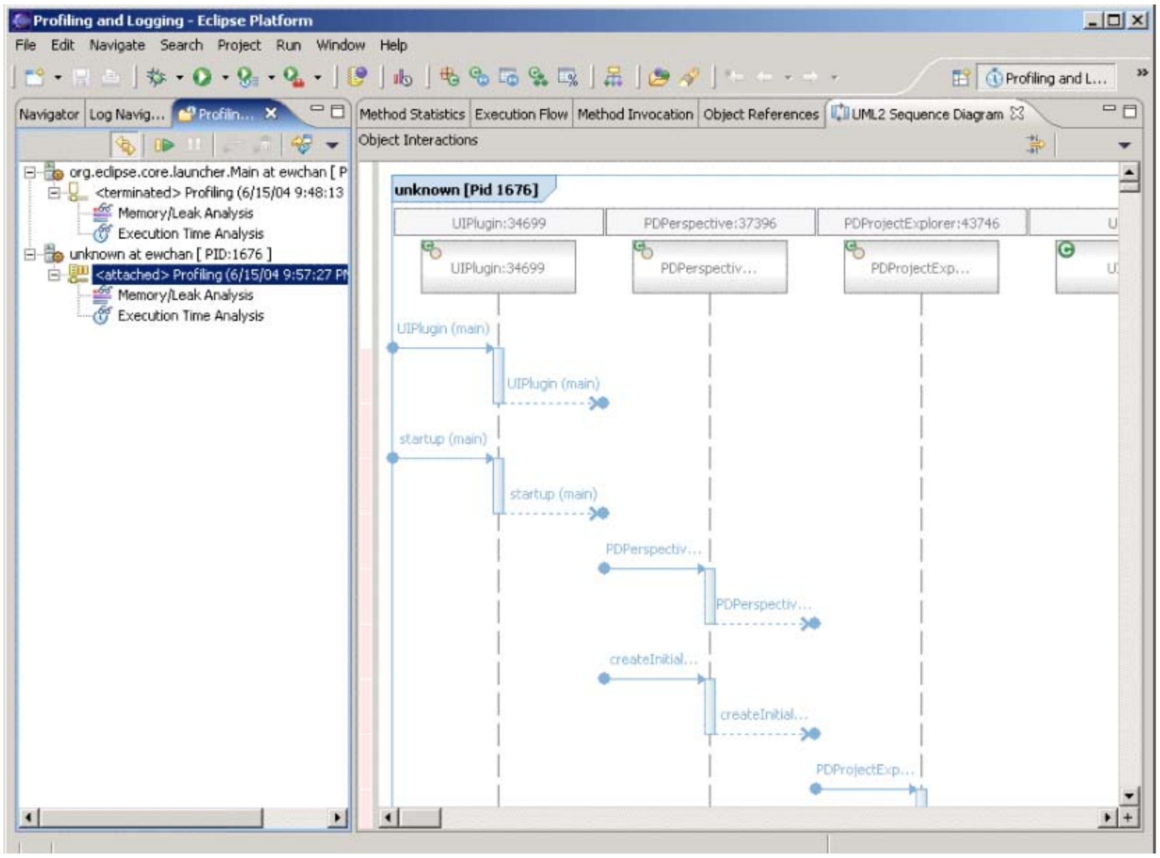


Figure 12 Hyades sequence diagram generated from a trace

2.6.5 VARE

VARE (Visualization Architecture for REuse) [56] is an architecture (a set of tools) that allows web-based visualization of remotely executing software. The remote data are exchanged through SOAP messages using HTTP connections. The trace data are represented in XML format and saved in a native XML database on the VARE server. To support different visualization goals, execution traces are encoded using XML based language XTE (eXtensible Trace Execution) and static information is represented in another XML language RCD (Reusable Component Description). XTE stores the runtime information of RCD components.

On the client side, the user can manage activities that are associated with creating and viewing visualizations. A specific visualization can also be stored into a visualization repository so they can be retrieved at a later time. The advantage of VARE is that only relevant information needs to be extracted from the native XML database to create visualizations.

VARE uses standard UML diagrams, such as class and sequence diagrams to present the structure and interactions found in the execution trace. These diagrams are rendered through SVG in a web browser.

2.6.6 ARE

ARE [57] is a special-purpose dynamic analysis tool to convert Java reflective (dynamic) calls to real object and method names that exist in a trace. When object and method names are invoked using the Java Reflection API, such as "*class.newInstance()*" or "*method.invoke()*", as is commonly done in tools, displaying the reflective method names will result in an unreadable trace. To facilitate a better understanding of the application, ARE replaces these reflective calls with actual class or method names. For example, "*class.newInstance()*" may be changed to "*new TraceLoader()*", and "*method.invoke()*" may be changed to "*TraceLoader.open()*". Besides the execution traces, it also takes into account the actual parameters of each invocation. Therefore, analysis can be concentrated to focus on those method invocations that involve passing a certain object. This also enables the understanding of how data are exchanged between threads.

ARE uses a layered architecture and employs AspectJ to instrument Java applications. A tracing aspect identifies each object instantiation and method invocation and forwards this to the recording layer. The recording layer incorporates filtering capabilities and records data of interest to a database. This trace aspect can either be weaved at the program level for the whole application or be tuned to weave only a subset of classes.

ARE also supports tracking how a single object is being used by the system, including objects passed in aggregate or child objects or as a wrapper. This can provide a variety of insights into the execution because it only reports events to the object of interest.

2.7 Related Tool Evaluations

Pacoine et al. evaluated five dynamic visualization tools focusing on their presentation capabilities: AVID, Jinsight, jRMTool, Together ControlCenter diagrams and Together ControlCenter debugger. The tools were evaluated on a number of program comprehension and reverse engineering tasks using the JHotDraw framework. The tasks included identification of software structure, design pattern extraction, function localization, etc. The diagramming techniques were categorized into graphs, UML diagrams and message sequence charts (MSC). The abstraction levels that the tools supported were classified into low, medium and high. The results revealed that level of abstraction affected the success of a tool to certain tasks. It showed that no dynamic visualization tools can respond to all the questions in the tasks and implied that tools were not adequate in isolation to support software comprehension [58].

Hamou-Lhadj et al. conducted a survey of trace exploration tools and techniques. The focus was on how big traces are represented and reduced in these tools and how abstractions are achieved. The tools included Shimba, ISVis, Ovation, Jinsight, Program Explorer, AVID, Scene, and Collaboration Browser. The results indicated that there was a lack of proper representation of large traces. Most tools supports pattern detection and matching in reducing the size of traces but there were no further experiments on their effectiveness. Some tools also implemented the same techniques using different terminologies; this indicated the lack of a much-needed framework for trace analysis [59].

2.8 Trace Formats

2.8.1 Classification

Different trace encoding formats have been introduced in accompanying trace tools. Generally, formats can be divided into three categories:

2.8.1.1 Proprietary formats

A proprietary format is only used by a specific tool; the format is only understood and interpreted by the tool. Converters are needed if trace data is to be shared among different tools.

2.8.1.2 Self-descriptive formats

To alleviate the incompatibility problems of different trace formats and tools, self-descriptive formats have been invented. These include instructions on how to interpret the format. The two representatives we will consider are TDL and SDDF.

TDL [60] is a Trace Description Language that enables users to access and decode data in an execution trace file. TDL assumes that event data in a trace file is a generic abstract data structure. It defines that a trace file will have the following hierarchy: event trace / trace segment / trace record / record field. Therefore, the main differences between different trace formats are record fields. Besides predefined field types, it also includes user defined identifiers and the interpretations for the values of that fields. The user will describe the meaning of all the data elements in a description file using TDL. At runtime, a system will first load the description file and then use it to further decode a trace.

SDDF format (Self-Defining Data Format) [61] does not have separate description file as in TDL; instead it contains a header section in each trace file that describes various type of records in the trace. Parsing the trace records is dependent on the header information. A trace file in SDDF format typically includes a group of record definitions in the header section, and a subsequent sequence of tagged data records. The tag indicates the type of the record, and a parser can interpret record using a particular record definition.

2.8.1.3 General format

To satisfy broader needs and promote interoperability of reverse engineering tools, different general formats have been proposed to support data exchange.

GXL is a standard exchange format for graphs [62]. Its development was particularly motivated by the need to enable interoperability among reverse engineering tools, such as code parsers, data flow analysis tools, and software visualization tools. GXL was developed based on the observation that much of the information about software systems can be best represented as a mathematical graph – and if this is done, then the mathematical power of graph algorithms can be employed. The drawback of GXL is that it is verbose- that is, a large number of tags is required to describe even a small graph. Furthermore the use of XML tags will make it even more verbose and this will lead to performance implications.

The Hyades project, introduced in section 2.6.4, also addresses the interoperability issue and provides a model in an XML based format. The drawback of these general formats is the performance issue in representing large traces.

The current tendency is towards the general standard format for data exchange and tool interoperability as addressed by the Hyades project. But for any encoding scheme of a trace, the key is determining the patterns that are used to encode the information of interest [63].

2.8.2 Rationale

The rationale we use for adopting a specific format lies mainly in our application context. For a format to be useful, it must contain the following attributes:

2.8.2.1 Dealing with size

As introduced in previous chapters, our research is motivated by the size explosion problems of large traces. Three main techniques are often used to deal with the size problem: filtering [64], sampling [65, 66, 67] and compression [68, 69]. Trace filtering will discard all redundant information from original trace when references map to those defined in a filter. Trace sampling only stores data for relatively short intervals. The drawback of these techniques is that they may not be able to represent the exact program behaviour because of loss of information. Only trace compression can both reduce the size of a trace and retain all the information from the trace. Because raw trace data can accumulate quickly and can take up gigabytes on average for ASCII based formats [68], a format should not only ease requirements for data storage, but also assist in accessing the information contained in large traces.

To effectively handle large traces, we need a model with an encoding scheme that can represent the whole trace – even those with millions of lines. Another benefit of a model is that summary information can be saved and queried easily to assist performance analysis. A trace format should ease all these tasks.

2.8.2.2 Facilitating visualization

One goal of trace analysis is to visually present the result to end users to help them determine attributes of the system under consideration. Besides helping model representation of large traces, the designers of a format need also to consider how easy the format can be mapped to a visual representation.

2.9 CTF (Compact Trace Format)

CTF (Compact Trace Format) was introduced by Hamou-Lhadj and Lethbridge based on the idea that common subtree in traces can be shared and represented only once [1]. CTF is a lossless format because it does not discard any events or method calls recorded in trace capture phase. The metamodel of CTF is shown in Figure 13.

The CTF format represents a trace using a Directed Acyclic Graph (DAG). The DAG can be built by traversing the tree in post-order to identify identical sub trees and represent each exactly once in the DAG. Multiple threads can be considered similarly as a forest of call trees; they share the sub tree pattern when generating the DAG form. The resultant graph is composed of root nodes of all the threads.

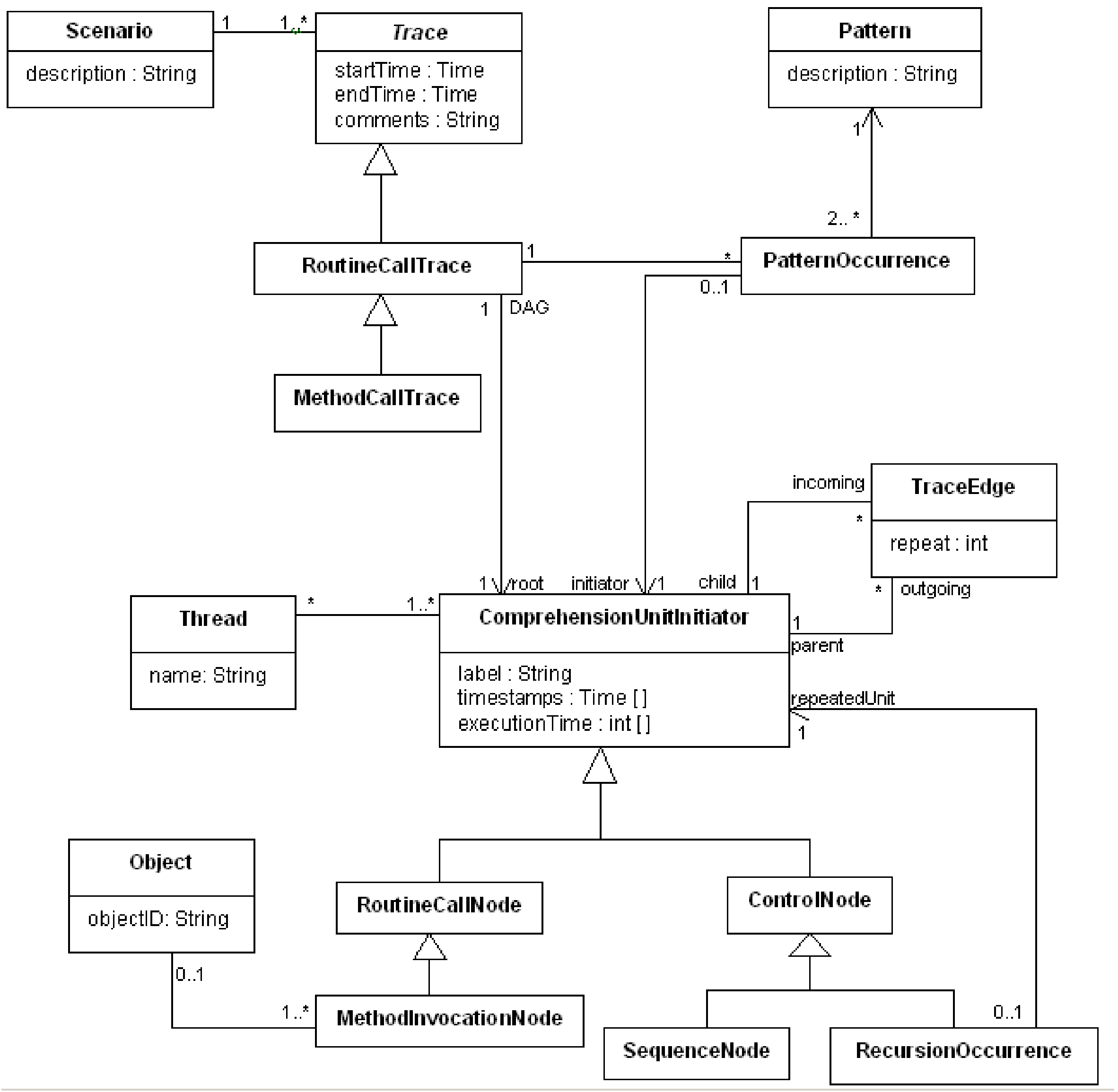


Figure 13 CTF metamodel

A further step, building on the DAG approach is to compress a sequence of calls using the Sequitur algorithm into the form of a context-free grammar [70, 71]. This grammar reflects its input's hierarchical structure and can give back original trace data when fully expanded.

The DAG representation has the following properties:

- The descendents of any node form a tree when traversed.

- The ancestors of any node form an inverted tree when traversed. (Because this DAG is ordered, traversal must be done in this order.)

The DAG structure emphasizes reuse of hierarchy structures. It forbids two nodes from having more than one distinct path between them. DAGs are constructed based on the observation that the information in different places in the hierarchy has a high degree of correlation (repetition) [53].

2.9.1 Filtering Algorithms and CTF

One goal of CTF is to represent large traces for data exchange. Filtering algorithms can process CTF formatted traces to hide implementation details. A filtering algorithm can be applied to a whole trace or part of it; it can filter some parts of trace and identify the main high-level properties [72]. There are three types of filtering techniques that can be used on a trace.

- Pattern matching
- Detection of utility routines
- Automatic detection of abstract operations

2.9.1.1 Pattern matching

Pattern matching is used to group similar sequences of events as execution patterns. Because identical subtrees are already identified and organized in CTF, finding such patterns can be done very quickly. Besides exact matching, other matching criteria have been used such as ignoring the order of calls, ignoring the number of repetitions, and limiting comparison to certain depth. Other comparison of tree similarity can also be found [73].

2.9.1.2 Detecting utilities

Utilities are those routines and classes that help the implementation of the system's functionality but whose details can normally be ignored when understanding how the system works. Removing utilities would therefore not affect the comprehension process [69]. Normally, statistical information can be used to help determine whether something is a utility. For example, if a method has a high fan-in (the number of methods that call this method), low fan-out (the number of methods called by this method) and is called from places scattered throughout the system, then it can be a good candidate to be a utility. However, the difficult part about creating methods to detect utilities is the need to conduct empirical evaluations to ensure the utilities detected are valid from the perspective of software developers.

2.9.1.3 Detecting abstract operations

An abstract operation is an operation that can be implemented in different ways depending on the context where it is defined. Polymorphism is a typical way for implementing abstract operations. In procedural languages, similar naming convention for routine names can also indicate a common abstract operation [74].

2.9.2 Comparison of CTF Model and Hyades Model

CTF model is specifically designed to enable program understanding tools exchange traces of method calls, which form a natural hierarchy. On the other hand, Hyades trace model is based on sequential logs of events; it focuses more on trace-to-test conversion and automatic testing instead of program understanding. For example, a process can have several threads, and a thread invokes a series of methods (method attributes and invocation attributes are separated). Clearly, DAG based CTF model is not directly supported by the trace model from Hyades. However, a trace using CTF model can be built upon information extracted from a trace using Hyades model. Extra information found in Hyades model, such as temporal information, correlation between threads, etc. can also supplement CTF model. The direct benefit is that we will no longer be concerned about trace capture and format conversion and raw trace data persistence, which are provided by the Hyades platform.

On the other hand, traces encoded in CTF format can also be manipulated to generate the original trace. Because a CTF model is a directed acyclic graph, traversing the graph in pre-order will reproduce the trace as a tree of method calls.

As a result, CTF format traces can be constructed from Hyades traces and CTF is complementary to Hyades.

2.10 Summary

From the previous review, we can draw the following conclusions:

Trace analysis is a crucial method in reversing engineering to understand interactions between participating components. It allows the software engineer to study the run-time domain of an actual program execution, and gain precise information which would otherwise be unavailable if only static analysis approaches were used.

Human understanding of programs uses a mixed combination of top-down and bottom-up approach, with a tendency towards the intermediate level of abstraction between the application domain and implementation details. Tools should therefore facilitate human cognitive models by providing different views, different abstraction levels, and the ability to switch among them.

Visualization techniques can help users understand the system. However one question relating to visualization is what kind of pictures should be drawn. The preferred philosophy is to use standard diagrams (such as UML) before inventing new ones [75].

Trace exploration tools can help users understand the abstractions of a system and obtain summaries of its dynamic behaviour. Some tools can generate sequence diagrams or the like, while others can generate class diagrams [76, 77]. This functionality provides good support for abstraction. Many trace visualization systems [49,78] focus on the time taken to perform some operations and how the systems respond to real-time changes. This can help identify performance problems. However, when understanding large complex systems, software engineers must focus more on system-level behaviour and structure instead of details such as CPU time. Therefore, trace exploration tools need to support extraction of high-level information from traces.

The Hyades project is the most promising one for laying out a framework for tool integration and interoperability. Furthermore, it allows reverse engineering tools to interact with other tools, such as IDEs. As supported by different tool evaluations and experiments [58, 59, 79], such integration is crucial in precisely evaluating and comparing the effectiveness of reverse engineering tools. Only through a high level of integration with development environments are reverse engineering tools likely to be adopted.

Chapter 3 Challenges

There are many challenges in building an effective and usable tool for reverse engineering; these include dealing with the large size of traces, and providing effective navigation capabilities. This chapter will investigate these challenges and discuss how to address them in building efficient tools.

3.1 General Challenges

3.1.1 Difficulties in Empirical Studies of Software Engineering Tools

Redmiles pointed out that there are several potential obstacles in doing controlled empirical studies on tools [80]. These studies require an integration of various disciplines, including psychology, sociology and software engineering. Second, the effectiveness of a tool is often difficult to be measured precisely. Third, market considerations are often the driving force of such development, rather than the scientific question of what works best for software engineers.

3.1.2 Tool Adoption

Most reverse engineering tools attempt to create an environment in which the tool itself has dominant control [13]. This approach hinders the easy integration of reverse engineering tool with other commonly used development tools. Therefore, in order to increase the adoption, the tools first need to be interoperable with other software engineering tools or need to be integrated into commonly used development platforms. Also there are many other factors that affect tool adoption, such as costs, benefits and risks [81]. The adoption problem mainly originates from different perception of tool developers and end users, where the developers are convinced that the tool has value, but the end users are not.

3.1.3 Understanding Advantages and Disadvantages of New Emerging Media

Effectiveness of emerging visualization techniques, such as animation and 3D visualization, and their impact on humans should be further studied and validated. To minimize the side effects of new visualization techniques, human-centered visualizations should always be adopted [82]. Therefore, we need to better understand human behaviours, including how people interact with information, how they perceive the information both visually and non-visually, and how human minds work when searching for information.

3.2 Challenges in Our Research Context

From the background review presented earlier, we can see that many trace-related tools have been built using a variety of techniques. Also, experiments with these tools have produced many guidelines that can be used to direct succeeding tool development. How to make a good synthesis and create a general set of guidelines remains a challenge.

3.2.1 Our Research

Our current research focuses on effective handling of large traces – those with millions of lines. We will develop ways to quickly load traces and present traces so that a software engineer can easily understand and manipulate them, with the ultimate goal being to improve the productivity of software engineers as they solve maintenance problems, particularly in real time systems. To help understanding, we will abstract away non-essential aspects of traces, such as calls to utilities, or accesses to architectural components of the system in which a software engineer is not interested.

The main practical result we expect to achieve is a working tool in Eclipse that incorporates a variety of algorithms that are used to filter trace events to improve understandability. The tool would have a user interface for browsing traces and a control panel that would allow the software engineer to fine-tune various aspects of the filtering. From a scientific perspective, the main result will be a better understanding of how traces can be filtered, explored and visualized so that software engineers can be helped in understanding a system.

3.2.2 Challenges in Large Trace Exploration

During the analysis of requirements and further brainstorming with software engineers, we have uncovered a number of requirements that raise very interesting research challenges that need to be addressed in order to build an effective tool for dynamic analysis of large software systems. Some of these challenges can be found in [83].

More specifically, the following challenges relating to performance and user interface are discussed.

3.2.2.1 Effective trace loading and processing techniques

Traces need to be efficiently loaded and processed in order for a tool to be adopted. According to a well-researched usability design guideline, the system response time for a loading operation in order to keep a user's attention focused on the current functionality should be at most 10 seconds [84]. Beyond that time, the user may lose patience and do some other task, or think the system is no longer responding. Part of the solution to this performance problem is to adopt CTF as our model for representing all the needed data. As discussed in [1], CTF has performance advantages as compared to competing formats because it is designed to be quick to parse and does not require re-loading repeated patterns.

3.2.2.2 Modularity of the analysis framework

As discussed in the previous chapter, different filtering algorithms can be applied on traces to reduce the apparent complexity. Each algorithm can have its specific requirements for input parameters and interface elements. In order to allow general initialization and loading of the algorithms, a framework must be defined so existing and future algorithms can be easily integrated.

3.2.2.3 Design of a user interface widget that loads only what is needed

Many user interface elements for displaying large amounts of information build a complete representation of the display in memory, and then make only sections of it visible through user interactions. However, in the filtering context, when the user changes the parameters of any of the filtering algorithms, these changes will cause the reconstruction of the entire internal display, despite the fact that only a tiny fraction will be visible.

Therefore, a new type of browsing widget that will generate the visual part of the display is needed. When the user scrolls, uses page up/down keys, expands the window or modifies the content of the display, any necessary new model data will be loaded dynamically and the new part of the display will be quickly generated. The widget also needs to support long scroll jumps, for example quickly scrolling to the bottom of a large tree and displaying the relevant section of tree nodes.

3.2.2.4 Supporting additional expanded states for a node

Traditionally a non-leaf tree node has two states: expanded or collapsed. In our trace exploration context, 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 certain exploration point, a non-leaf node can have the following three states: collapsed, that is, all the child nodes are not displayed; fully expanded, that is, all the child nodes are shown; and partially expanded, meaning some or all the children are hidden by some filtering algorithm, so only part of children are currently displayed.

Supporting this additional state will be very useful, we propose, in helping the user to conduct effective trace exploration. We consider the following application scenario: A user applies some filtering algorithms to a trace, some child nodes of a sub tree *A* are filtered and become hidden, but the user still wants to further look at the details of sub tree *A*. Instead of revoking all the algorithms that filter the trace to make all children of sub tree *A* visible, a mechanism is needed to quickly switch the filtered nodes between visible and invisible at the node level without affecting filtering algorithms that are in effect.

3.2.2.5 Integration with development environments such as Eclipse and Hyades

One factor that can significantly increase usability of a trace tool is integrating the trace browsing facilities with a standard integrated development environment (IDE), such as Eclipse. Researchers developing SHriMP [79] have shown the necessity for such integration. The result of their experiments indicates that without such integration, precise evaluation of the effectiveness of a reverse engineering tool is infeasible.

Eclipse is a universal platform that can be extended by plugins and is now widely used by tool providers. Integration with Eclipse has the following advantages:

The end user can reuse the extensive supply of current IDE features, such as facilities to program in C++ and Java. Fully featured IDE features are available free. The user will often be actively programming while exploring traces.

Enhanced tool usability. There has been much effort to improve the usability and accessibility of Eclipse's interface. If designed according to the guidelines, a tool can greatly benefit from current user interface design expertise.

The tool developer can reuse modelling tools and other tools in the same environment. There are many development tools available on the Eclipse platform, for example, various UML tools. Therefore, a developer does not need to start from scratch when such a feature is desired. Required features provided by other plugins can be easily reused.

Despite these benefits, we discovered that, making a tool seamlessly integrated with the Eclipse platform and other plugins so the user can have a common look and feel is a demanding task. We had to call upon an Eclipse expert to criticize our earlier UI designs; this allowed us to adjust some of the decisions we made so our tool would feel more Eclipse-like.

3.3 Summary

Some of the challenges can also be formulated into tool requirements. We will address these challenges in the following chapters, and propose our solutions to them.

Chapter 4 Architecture of SEAT

This chapter will introduce SEAT (Software Exploration and Analysis Tool), a tool encompassing the new widget and dynamic loading techniques. We will describe how it is constructed to tackle the challenges discussed in previous chapters.

4.1 Desired Features for A Trace Analysis Tool

Based on analysis of research in related fields and evaluations of various tools, we present a group of requirements for a Reverse Engineering tool.

4.1.1 Facilitate Program Comprehension

Our high-level research objective is to help software engineers understand existing systems through manipulation of captured traces. Through studying of mental models of human understanding, we know we need to support both exploring the architecture or high-level abstractions embodied in a trace, as well as the low-level details of the source code. Intermediate level abstractions need to be visible as well [27].

Requirement 1*: The tool needs to support high level abstraction, intermediate level abstraction and source code navigation.*

Soloway et al. [24] characterize software-understanding activities as a series of inquiries, during which the maintainer reads some code, asks a question about this code, conjectures a hypothesis, and then searches the code and other documentation to confirm the conjecture. Often the information needed to confirm a hypothesis is disseminated across the system; so an effective tool should provide mechanisms to locate code that programmers are interested in precisely and quickly through search support. The tool should also be flexible to facilitate the situation when programmers need to change their hypothesis or backtrack to an earlier hypothesis. Searching activities can reflect how a programmer traces basic software artifacts and constructs his mental models. According to Lethbridge, a code exploration tool should meet two functional requirements: (1) a user should be able to search for software artifacts by arbitrary strings or regular expressions and (2) display attributes of retrieved artefacts, such as the call hierarchy [85].

Requirement 2*: The tool needs to support broad search capabilities, including basic search, customizable search, wildcard search, regular expression search and with search history being traced.*

4.1.2 Visualization

Visualization is about rendering large data sets on the screen. An ideal visualization system should allow understanding of detailed information while always providing a global context [34]. However, such a global view that displays all the trace data in one screen is difficult to generate. On the other hand, because of restrictions of human cognition and perception, humans can only understand and absorb a small volume of information instead of the whole visually compelling representation. Therefore, current strategies focus on fragmenting the view of the information and providing a set of correlated views so a user can explore detailed information about a trace if desired.

Requirement 3*: The tool needs to support the global view of the trace being explored and various views of data slices from different perspectives.*

As discussed in Chapter 3, to reveal high-level behaviour of a system, different filtering techniques are introduced to filter uninteresting events, so implementation details are hidden. Interesting entities, such as patterns and utilities should be easily highlighted and distinguishable. It should be noted that such operations should be accomplished within appropriate time, such as 10 seconds according to general usability design guidelines.

Requirement 4*: The tool needs to support filtering of events, highlighting events that have specific attributes using colours or shapes.*

4.1.3 Exploration

Exploration should facilitate users in finding specific information. When a user is exploring a large information space, it is easy to lose orientation [33]. Facilities should be there to keep user oriented so the user can always be aware of his position with in the data set. One strategy is to provide an indication of the current location in the global view if such a view is available as discussed in Requirement 3. An alternative is to provide landmarks so the user can keep track of previous steps and go back when necessary. Providing landmarks is often proposed as an aid to navigation so interesting information is always discriminable [86].

Requirement 5*: The tool needs to support user orientation, such as the position in the global view; this concept is also known as landmarks.*

When developers want to navigate to artifacts that are related to the current one, they are forced to switch to different views or even to switch between tools. Switching is disorienting and it breaks the streamline of the exploration path [87]. As a result, a developer loses his or her current position with respect to the exploration task.

Requirement 6: *The tool needs to minimize view switching when a user is carrying out a task.*

4.1.4 Other Features

Data may come from various sources and be saved to various destinations, such as a file, network connection [55], database [56].

Requirement *7: The tool needs to be extensible to support different sources and destinations.*

A session allows a user to save current work and return to it without losing anything previously done. This includes: views and their positions, layout and algorithms applied, current visualization context, etc. It also needs to record existing knowledge gained so far for future maintenance and reference [22].

***Requirement 8**: The tool needs to support workspace and session persistence so a user can continue previous work at a later time.*

Handling large data sets with real-time response is the main motivation of our research. Most existing tools cannot effectively handle very large traces; therefore, their usefulness is severely discounted into solving real-world problems.

***Requirement 9**: The tool needs to be able to handle traces with millions of lines of data easily and effectively, including loading and saving.*

The last item discussed here is tool adoption problem. For real tool adoption, both utility and usability need to be experimented with end users. *Utility* means the system can help users to accomplish the general tasks encountered in maintenance. If a tool is not useful in this aspect, it will be less likely to be used. The tool also needs to be designed with *usability* carefully considered; this includes aesthetics, feedback, responsiveness, learnability, error handling, etc. Another aspect that can promote the acceptance of a tool is it closely integrated with a development IDE, especially a development environment with which the user is familiar [79].

***Requirement 10**: The tool needs to adopt user centered design and can be integrated easily with an IDE and other tools to increase its adoption.*

4.2 Overview of SEAT

Based on tool requirements described in the previous section, we built a prototype tool, SEAT, on Eclipse platform for exploring large execution traces. Figure 14 shows the overall flow of information using SEAT. The tool manipulates traces in CTF (Compact Trace Format) and displays them using the new tree widget we have developed. To help the user extract useful information for a trace, SEAT incorporates several trace filtering techniques. An analyst can filter a trace to the level where he or she can understand important aspects of its structure so complexity of the trace is greatly reduced.

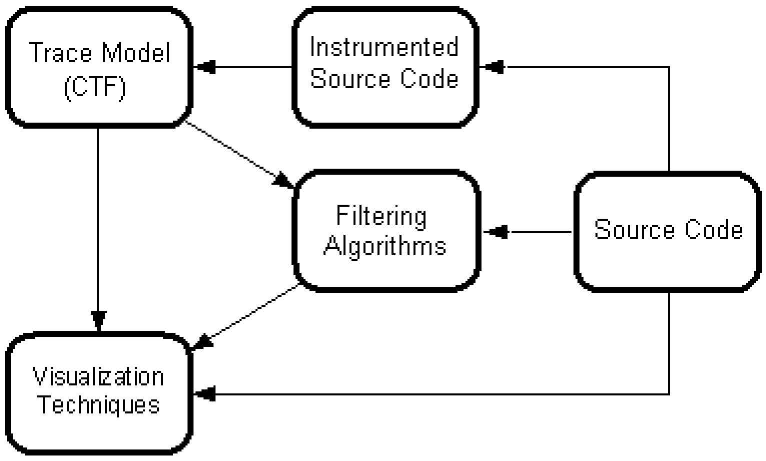


Figure 14 Data flow in SEAT

Figure 14 also shows one possible way for generating traces of routine calls, which is based on source code instrumentation. It should be noted that some filtering algorithms could refer to data based on static analysis of source code, for example, method fan-in and fan-out, used in determining utility methods.

4.3 Eclipse

Eclipse is an open source software tool platform. It has gained great popularity with its successful Java development environment. This section will describe the core functionality of Eclipse platform and how its plugin infrastructure facilitates tool integration.

The Eclipse platform is developed based on the concept of *plugins*. A plugin is a basic unit of functionality that can be treated separately on the platform. On the Eclipse platform, except for a small kernel that starts the system, all the functionality is provided through plugins. A plugin has a *manifest file* that describes the plugin in XML. The manifest file defines the runtime library required for the plugin, as well as extensions provided by the plugin that contribute to extension points defined by the platform and other plugins.

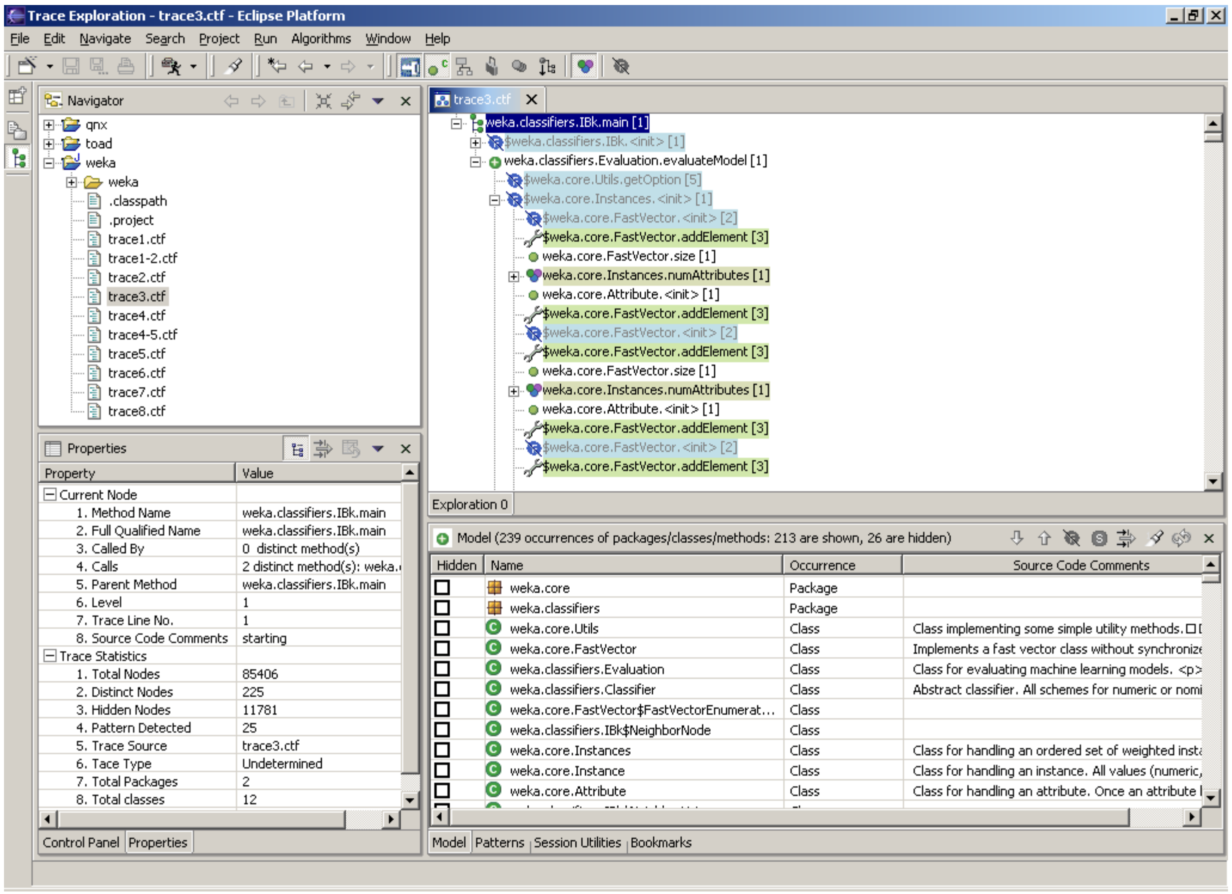
A plugin can also declare any number of extension points so other plugins can provide customized functionality.

Besides the runtime environment, the core infrastructure of Eclipse platform includes resource management, a GUI workbench, debugging and team support. To facilitate user interface development, Eclipse platform provides two sets of widgets: Standard Widget Toolkit (SWT) and JFace. SWT is a graphics library that use native window system widgets but in an OS-independent manner. JFace is a high-level interface toolkit based on SWT that simplifies common UI programming tasks. It provides APIs for editors, viewers, wizards, preferences, and so on. Most of the time when we are dealing with complex widgets, such as tables, or trees, the JFace API ought to be used. The SWT API is used when basic widgets are used, or when a special user widget needs to be developed.

When developing a plugin on the Eclipse platform, we need to consider two aspects: which extensions the plugin will provide to the platform and which extension points are defined for this plugin so other plugins can contribute. Typically, a plugin will provide extensions to platform extension points, such as "*org.eclipse.ui.editors*", "*org.eclipse.ui.views*", so data can be displayed in customized editors and views. Java implementations that extend the platform API are needed so the platform runtime can discover the extensions when the Eclipse platform launches. When a plugin defines its own extension points, the plugin must be responsible to find and handle extensions itself. Therefore, the plugin will typically define some common interfaces to be used as its API, and extensions to it can be discovered by querying the Eclipse platform plugin registry.

4.4 SEAT on Eclipse Platform

SEAT provides several extensions to the Eclipse platform. It includes a multiple page editor for visualizing traces, several views to help trace exploration, and preference settings to control data visualization. SEAT also defines an extension point, called "seat.algorithm" so other plugins can contribute filtering algorithms besides those provided by default. Figure 15 is a screen shot of SEAT. We can see that the workbench is divided into four parts. The upper left is the navigator where all the traces and their parent projects are displayed. The upper right area is the default editor area where a trace can be explored in several independent explorations concurrently using a multiple-page editor. The lower left is the control panel used to manipulate the global parameters that can affect display of all traces. A properties view that displays attributes of the trace and the current node is also stacked in this area. The lower right area is a set of auxiliary views to help view repetition patterns, utility methods, search result and bookmarks.



Trace Exploration - trace3.ctf - Eclipse Platform
File Edit Navigate Search Project Run Algorithms Window Help
Navigator
Properties
trace3.ctf
Exploration 0
Control Panel
Properties
Model
Patterns
Session Utilities
Bookmarks

```
qnx
toad
weka
  weka
  .classpath
  .project
  trace1.ctf
  trace1-2.ctf
  trace2.ctf
  trace3.ctf
  trace4.ctf
  trace4-5.ctf
  trace5.ctf
  trace6.ctf
  trace7.ctf
  trace8.ctf
```

```
weka.classifiers.IBk.main [1]
  $weka.classifiers.IBk.<init> [1]
  weka.classifiers.Evaluation.evaluateModel [1]
    $weka.core.Utils.getOption [5]
    $weka.core.Instances.<init> [1]
      $weka.core.FastVector.<init> [2]
      $weka.core.FastVector.addElement [3]
      weka.core.FastVector.size [1]
      weka.core.Instances.numAttributes [1]
      weka.core.Attribute.<init> [1]
      $weka.core.FastVector.addElement [3]
      $weka.core.FastVector.<init> [2]
      $weka.core.FastVector.addElement [3]
      weka.core.FastVector.size [1]
      weka.core.Instances.numAttributes [1]
      weka.core.Attribute.<init> [1]
      $weka.core.FastVector.addElement [3]
      $weka.core.FastVector.<init> [2]
      $weka.core.FastVector.addElement [3]
```

Property	Value
Current Node	
1. Method Name	weka.classifiers.IBk.main
2. Full Qualified Name	weka.classifiers.IBk.main
3. Called By	0 distinct method(s)
4. Calls	2 distinct method(s): weka.
5. Parent Method	weka.classifiers.IBk.main
6. Level	1
7. Trace Line No.	1
8. Source Code Comments	starting
Trace Statistics	
1. Total Nodes	85406
2. Distinct Nodes	225
3. Hidden Nodes	11781
4. Pattern Detected	25
5. Trace Source	trace3.ctf
6. Tace Type	Undetermined
7. Total Packages	2
8. Total classes	12

Model (239 occurrences of packages/classes/methods: 213 are shown, 26 are hidden)

Hidden	Name	Occurrence	Source Code Comments
☐	weka.core	Package	
☐	weka.classifiers	Package	
☐	weka.core.Utils	Class	Class implementing some simple utility methods.
☐	weka.core.FastVector	Class	Implements a fast vector class without synchronize
☐	weka.classifiers.Evaluation	Class	Class for evaluating machine learning models. <p>
☐	weka.classifiers.Classifier	Class	Abstract classifier. All schemes for numeric or nomi
☐	weka.core.FastVector$FastVectorEnumerat…	Class	
☐	weka.classifiers.IBk$NeighborNode	Class	
☐	weka.core.Instances	Class	Class for handling an ordered set of weighted insta
☐	weka.core.Instance	Class	Class for handling an instance. All values (numeric,
☐	weka.core.Attribute	Class	Class for handling an attribute. Once an attribute i

Figure 15 SEAT screenshot

SEAT integrates several filtering algorithms to reduce the complexity of traces. Filtering algorithms are independent of the user interface, and they are exposed to SEAT by extension point mechanism provided by Eclipse platform. Each algorithm is described in an XML configuration file and will be loaded by SEAT in runtime. On Eclipse the workbench, an algorithm is represented as both an icon in the tool bar and a menu item of a main drop-down menu. Besides automatic filtering by algorithms, SEAT also allows user oriented filtering so a user can select nodes and subtrees and hide them from the trace manually. Users can also specify a filtering criteria and the trace will be filtered based on the user input. All tree nodes marked by users, either manually or by algorithms, are highlighted by different colours and can be quickly hidden and shown in the editor. Filtering is done at model level and new data is retrieved and rendered after filtering. Compared with the traditional approach that filters data at the user interface level, our filtering and rendering technique does not require a heavy amount of refresh inside the user interface widget (i.e. a complete replacement of the content of the UI widget), so it has a major advantage from performance perspective. According to our experiment on an IBM P4 2.8G PC with 760M RAM, loading time for a trace with 2 million lines is about 8 seconds, and filtering time is about 5 seconds for a filtering algorithm.

4.5 Architecture

The architecture of SEAT can be divided into several components: I/O, algorithm, dynamic tree loading, editor, and views. We will discuss each component in detail.

4.5.1 I/O Component

In the following, we will describe some of the classes involved in I/O. Figure 16 presents a class diagram containing the details of these classes.

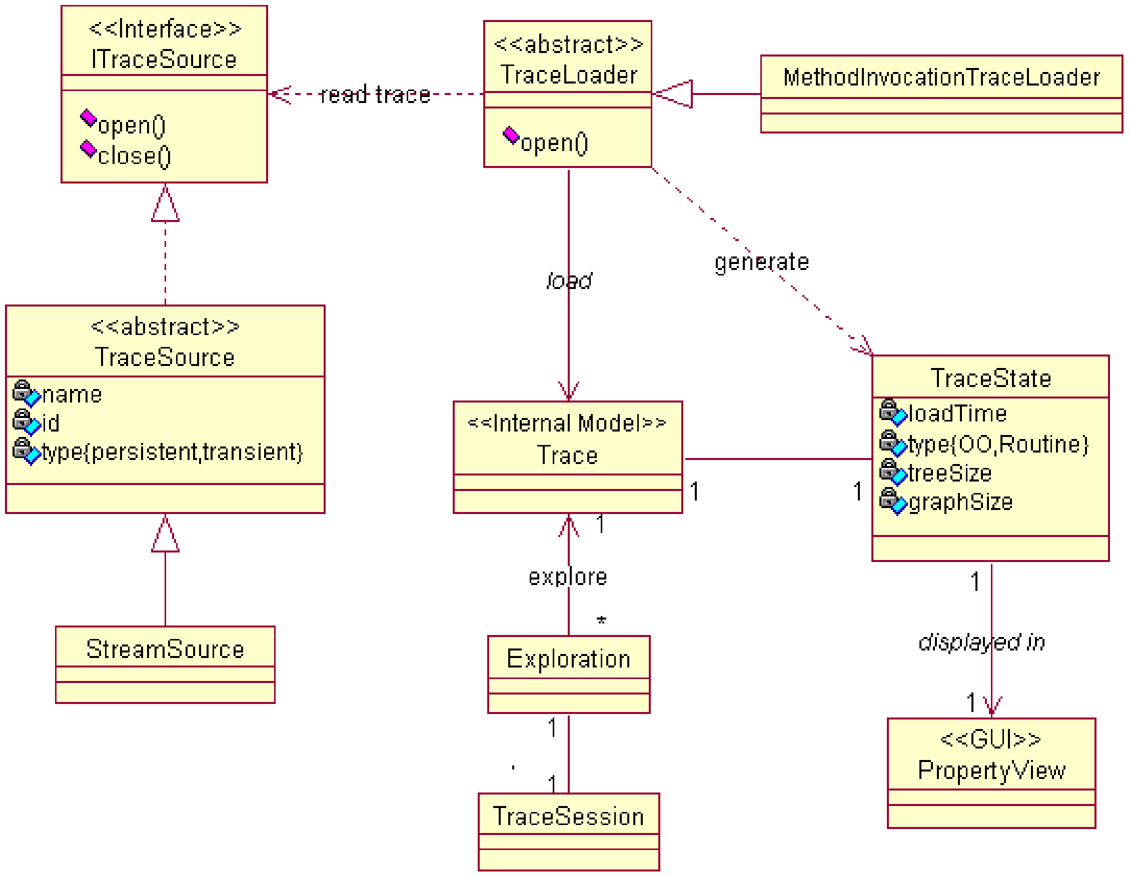


Figure 16 Class diagram for trace I/O

The input trace can come from different sources. As discussed in [83], possible sources include a flat file, a TCP stream, a database, etc. The interface *ITraceSource* defines the general signature to which a source implementation should conform. A basic implementation, *TraceSource*, is provided with abstract *open()* and *close()* methods. Therefore, a further extension like *StreamSource* has to be provided. *StreamSource* takes an *InputStream* when constructed, disregarding where the stream comes from. So a file stream, a memory byte stream or a TCP stream can benefit from this implementation. A trace delegates its initialization and data loading to an abstract *TraceLoader* through its *open()* method. A *TraceLoader* is aware of the format of a trace and will be responsible to construct a CTF trace model.

There are various ways to construct a CTF trace model by a *TraceLoader* and the construction is done through coordination of a *TraceLoader* and an *ITraceSource*. By default, a subclass of *TraceLoader*, called *MethodInvocationTraceLoader* will load traces in CTF format from a stream interface which is created by a CTF-aware trace capture system. However, typically a trace capture system will encode traces sequentially as events or method calls when they are generated as the system runs. Traces saved in this way need to be converted to CTF format using converters if they are to be loaded by the default loader. Another way is subclassing of *TraceLoader* so the new loader can construct a CTF trace model directly from sequential traces. With either approach, the algorithm that does the transformation is linear if the degree of the tree is a constant [1]. We have developed a trace converter to convert sequential traces to CTF traces with O(n) complexity based on subtree signatures. Some experiments were done on an IBM P4 2.8G PC with 760M RAM. The following table lists the size of traces and the time used to convert them to CTF format.

No.	Original Trace File Size (KB)	Total Method Calls	Conversion Time (sec)
1	3,345	85,406	2
2	15,718	219,507	2
3	14,636	385,434	6
4	22,705	631,530	4
5	29,127	742,812	4
6	94,862	2,409,740	14

Table 1 Trace size and conversion time

Besides construction of Trace model in memory, a *TraceLoader* can also gather information about a trace and save it in a *TraceState*. A *TraceSession* is used to save the current user session and recover it when reloaded. *Exploration* is used to support different user interactions and contains mostly state information so multiple explorations of the same trace are supported. A session will be created for each *Exploration*.

4.5.2 Algorithm Component

The algorithm component deals with the visual representation of all the filtering algorithms from the model. SEAT defines an Eclipse extension point called "seat.algorithm" so additional algorithms can be easily added without modifying the source code.

A newly developed algorithm must implement the *IAlgorithm* interface so it can both apply the filtering algorithm to filter the trace and revoke the filtering. An abstract class, called *Algorithm* which implements *IAlgorithm,* is provided to ease integration of a new algorithm into the framework (See Figure 17). If an algorithm has the capability to process a subtree and not just an entire trace, it can also implement *ICompressNode* interface. But that is not compulsory.

After the algorithm has been developed, the next step is to describe its attributes and parameters in a "seat.algorithm" extension point. The basic attributes of an algorithm include id, name, and implementation class, icon representing itself, description, category and an index. The category attribute identifies the kind of trace an algorithm can handle as either object-oriented or routine or both. The index represents a unique id of the algorithm and is used to identify the algorithm so exploration states of the current trace can be restored between persistent sessions. For example all filtering algorithms that are effective can be saved and restored at the next session easily. An improvement of the current approach would to calculate the unique index based on some attributes of an algorithm; the calculation would need to return same index all the time. An optional attribute for an algorithm is *parameter*, with which a user can define several input parameters for a given algorithm. Other optional attributes are pre-process class and post-process class for the algorithm. They can include any supplementary processing including UI related settings for the algorithm and will be loaded automatically. A schema for the "seat.algorithm" extension point definition can be found in Appendix B.

When a new filtering algorithm is developed and added as an extension to the extension point described above, it will be represented as an *IConfigurationElement* in the Eclipse platform. *AlgorithmDescription* will extract attributes from *IConfigurationElement* and it stands for a high level representation of an algorithm extension. The parameters of an algorithm that are defined in the extension will be parsed into *Parameter* objects.

For user interface, an algorithm will be visually displayed as an *AlgorithmAction*, which extends Eclipse *Action* class and can be displayed as both a menu item and a toolbar button. Also we aggregate all the *AlgorithmDescription*s into an *AlgorithmList* so their states can be managed in a central place. Another interface aspect is to allow the user to load parameters of an algorithm into a preference page and modify them. *Parameter*s of each algorithm are grouped into an instance of the *AlgorithmPreference* class.

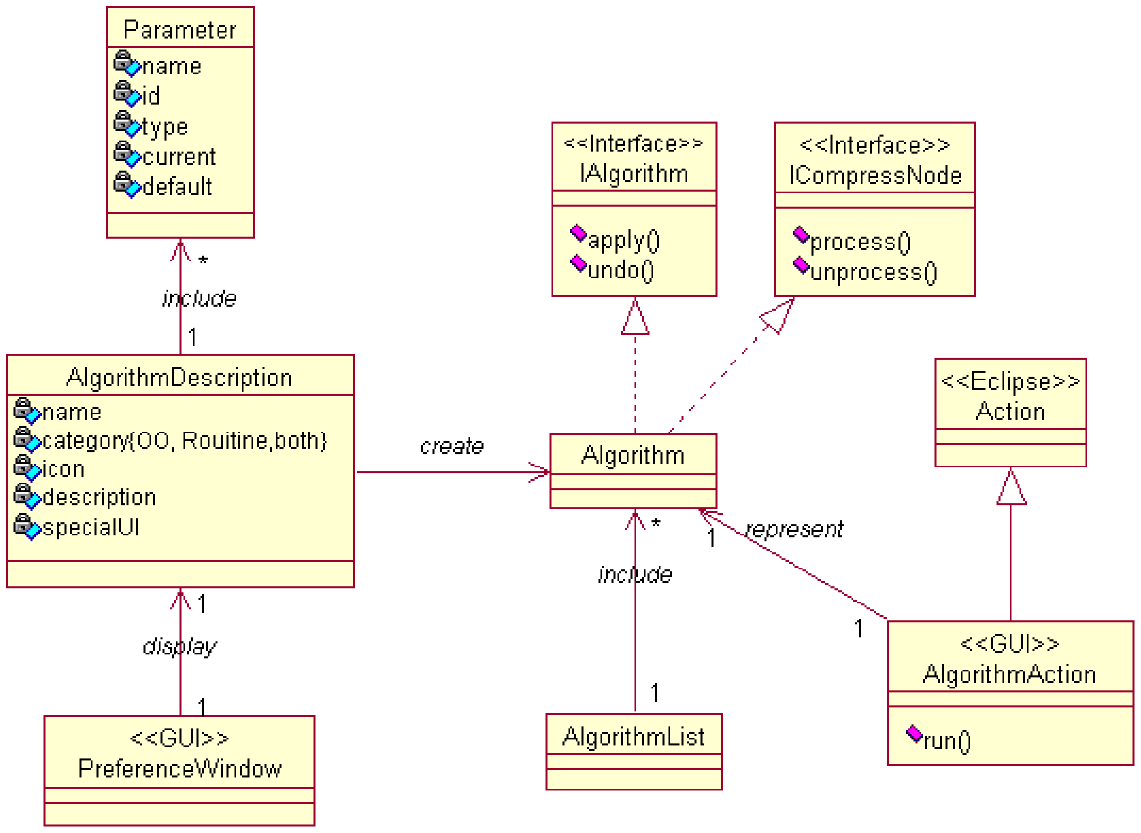


Figure 17 Class diagram for algorithm loading

4.5.3 Dynamic Trace Loading

TreeNode (See Figure 18) contains basic interface-related attributes, such as name, type, icon name, and description. It also realizes the parent/children relation. However, in a CTF trace model, all repeated nodes and subtrees will be only created once. Therefore a single node will have several parents and represent events that occur in different places in a tree. So a node object from a CTF model cannot clearly identify which exact event it stands for. This leads us to employ an edge-based approach to uniquely identify an event from the trace model. We use *UniquePath* to represent an event that occurs at a specific place in a tree. *UniquePath* essentially comprises a series of edges from the root node to a given node in a CTF model. The dynamic loading algorithm will identify all nodes that should be visible in the current window, create corresponding *UniquePath* representations of these tree nodes, and construct a *PartialTree* with the *UniquePath*s. This *PartialTree* can be further visualized by our new widget. Further explanation of the *PartialTree* and dynamic loading algorithm will be presented in Chapter 5, Dynamic Trace Loading Algorithm.

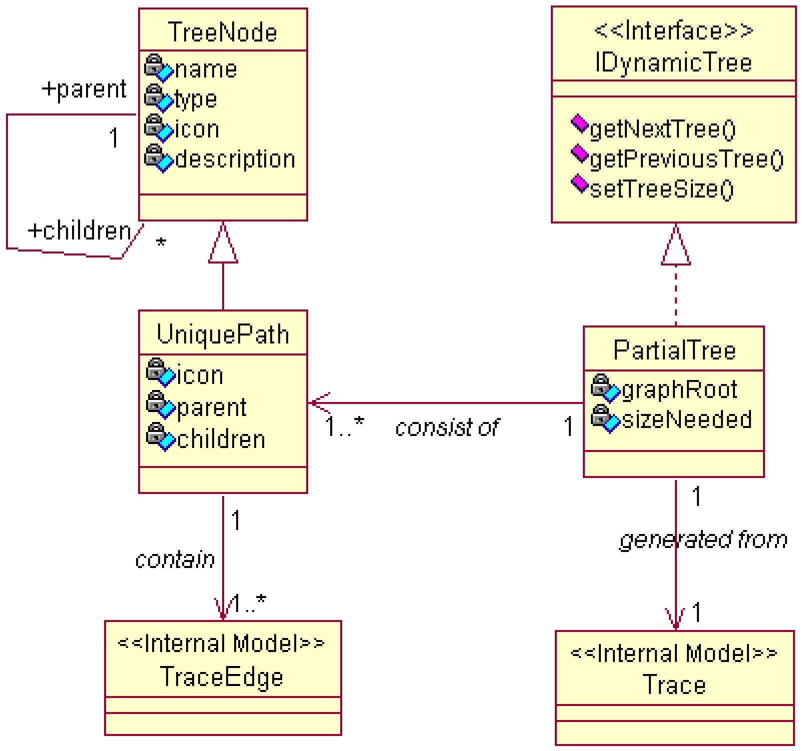
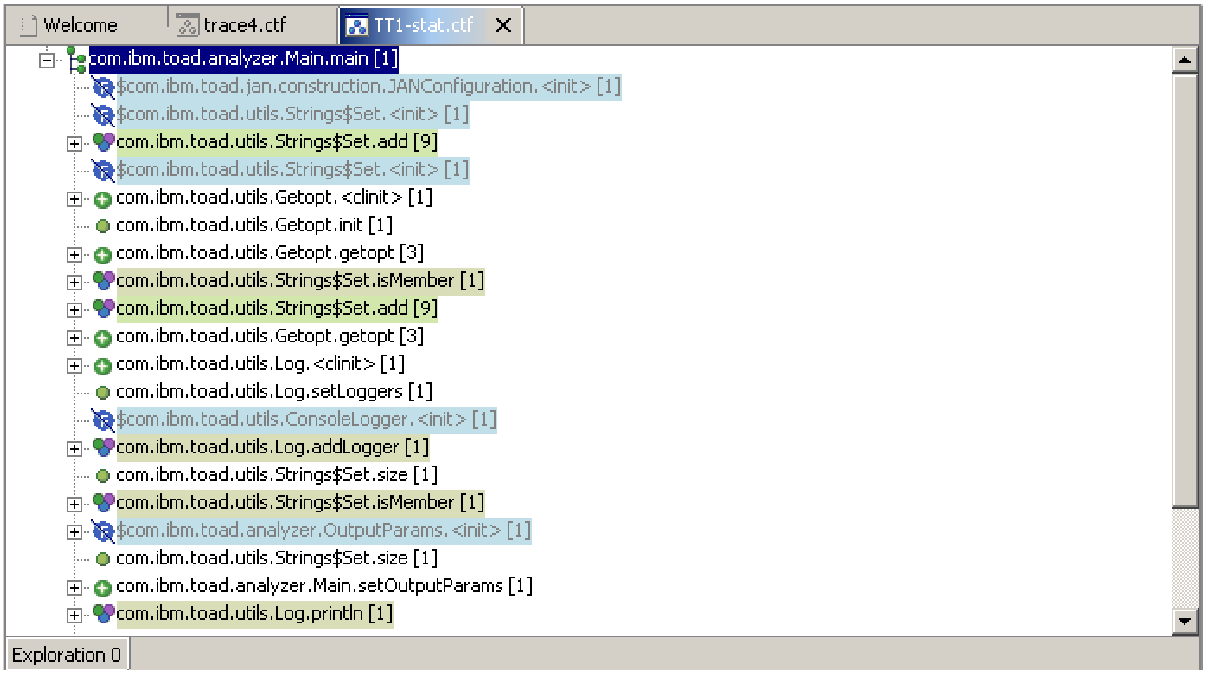


Figure 18 Class diagram for dynamic trace loading

4.5.4 Editor

Visualization of SEAT is driven by the Eclipse platform. SEAT uses multiple page editors (*MultiTraceEditor* class derived from *MultiPageEditorPart* class in the Eclipse platform) for trace exploration. Each editor corresponds to a trace and each page contained in the editor corresponds to an exploration. Because the Eclipse platform only allows opening exactly one editor for one set of input data, we use each page within a multiple page editor as a standalone exploration. Exploration statuses are cached in the page. A trace is displayed in a *TraceViewer* (See Figure 19), which manages exploration states, label providers and listeners to support trace exploration and visualization. To visualize the different statuses of trace nodes currently in the *TraceViewer*, customizable labelling and colouring techniques are supported. A user can choose whether to add special characters to the filtered nodes, or which colours and images to use to display nodes with different attributes. This is accomplished through standard decoration technique found in the Eclipse platform.



```
com.ibm.toad.analyzer.Main.main [1]
  $com.ibm.toad.jan.construction.JANConfiguration.<init> [1]
  $com.ibm.toad.utils.Strings$Set.<init> [1]
  com.ibm.toad.utils.Strings$Set.add [9]
  $com.ibm.toad.utils.Strings$Set.<init> [1]
  com.ibm.toad.utils.Getopt.<clinit> [1]
  com.ibm.toad.utils.Getopt.init [1]
  com.ibm.toad.utils.Getopt.getopt [3]
  com.ibm.toad.utils.Strings$Set.isMember [1]
  com.ibm.toad.utils.Strings$Set.add [9]
  com.ibm.toad.utils.Getopt.getopt [3]
  com.ibm.toad.utils.Log.<clinit> [1]
  com.ibm.toad.utils.Log.setLoggers [1]
  $com.ibm.toad.utils.ConsoleLogger.<init> [1]
  com.ibm.toad.utils.Log.addLogger [1]
  com.ibm.toad.utils.Strings$Set.size [1]
  com.ibm.toad.utils.Strings$Set.isMember [1]
  $com.ibm.toad.analyzer.OutputParams.<init> [1]
  com.ibm.toad.utils.Strings$Set.size [1]
  com.ibm.toad.analyzer.Main.setOutputParams [1]
  com.ibm.toad.utils.Log.println [1]
```

Figure 19 An editor displaying a trace with filtered tree nodes highlighted

TraceViewer extends another high-level JFace viewer, called *PartialTreeViewer*. *PartialTreeViewer* bridges between the low level *PictureTree* rendering widget and the data model, in this case *PartialTree*. For example, it supports selection of tree nodes, accepts a new *PartialTree* and delegates refresh of the screen to the *PictureTree* widget. *PictureTree* contains a rendering algorithm that draws the tree with exactly the same appearance as a traditional tree widget on the screen. Further explanation of *PartialTreeViewer* and *PictureTree* can be found in Chapter 6, New Tree Widget.

Other classes include parsers for different languages; they are called by the editor when the first exploration of a trace is loaded and are essential in the correct construction of the trace model. The parsers are also used to link a method call in a trace to the source code and extract data, such as comments for a method, from the source code. The diagram for editor and viewer are show in Figure 20.

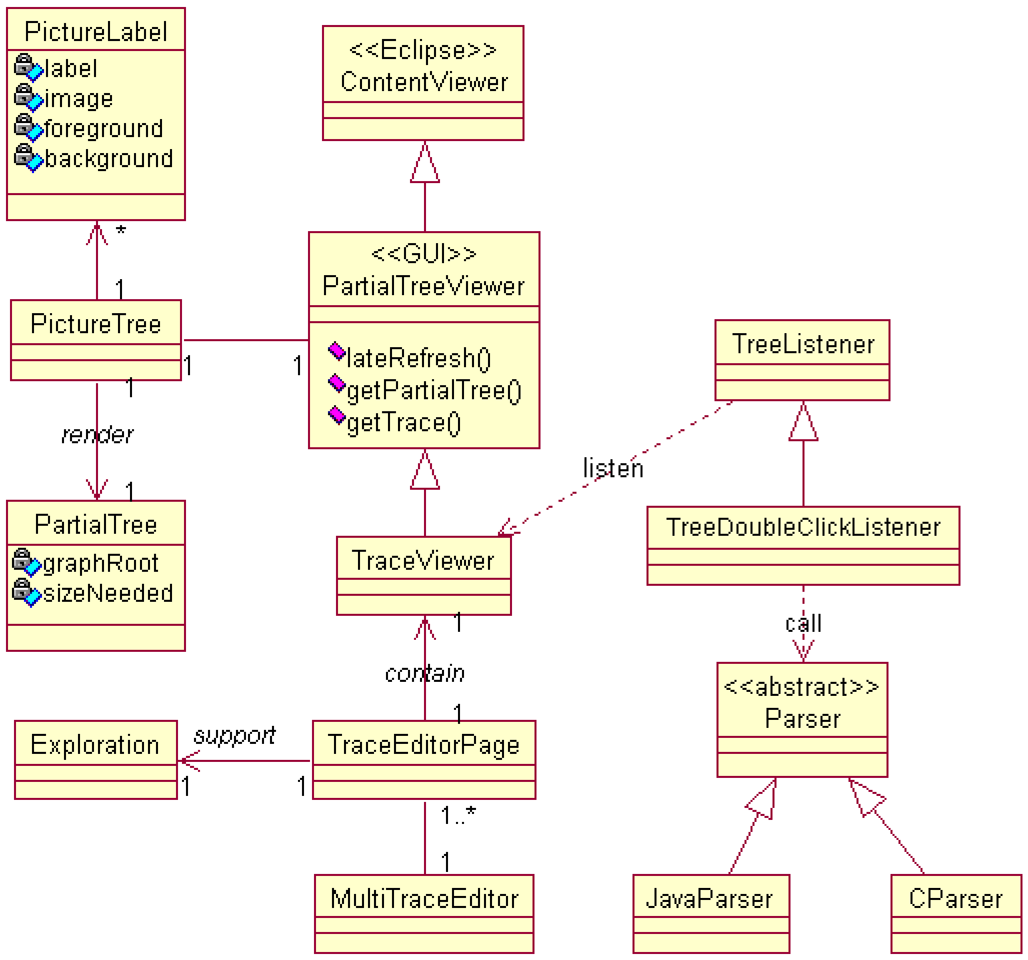
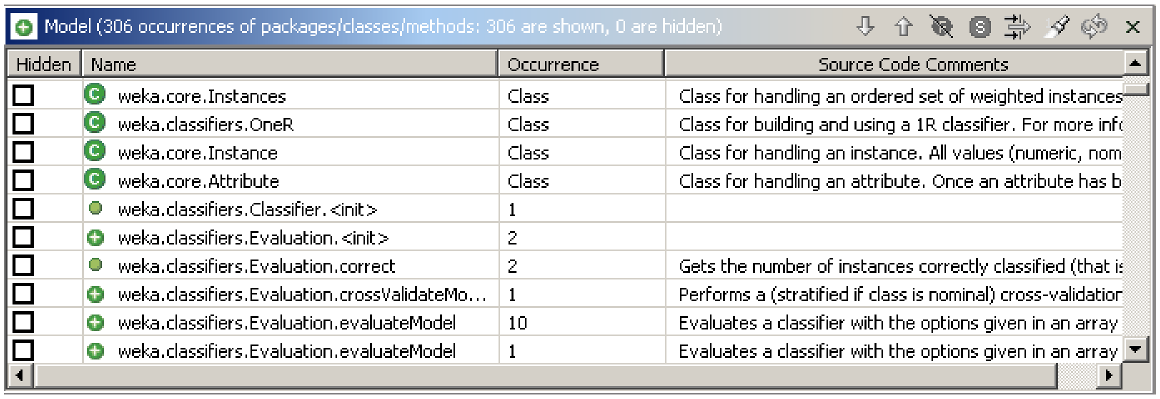


Figure 20 Class diagram for trace editor

4.5.5 Views

Views are typically used to navigate a hierarchy of information, open an editor, or display properties for the active editor.

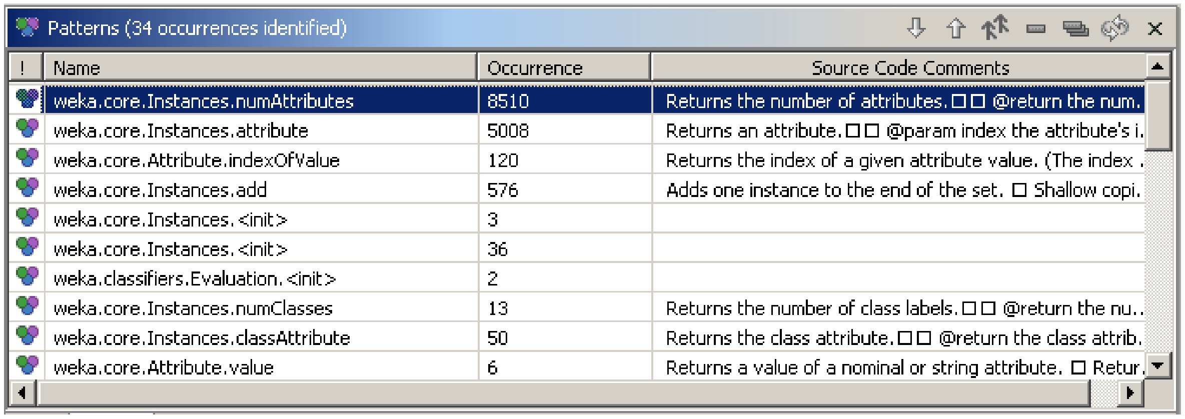
SEAT has a *Model* view to display distinct nodes from a CTF model and the current states of these nodes (See Figure 21). The *Model* view is also used as a kind of feedback when a user explores a trace, such as by applying an algorithm. Nodes in a *Model* view can also be hidden and shown directly and changes will be reflected in the editor so all visual tree nodes corresponding to the selected model nodes will be refreshed.


Model (306 occurrences of packages/classes/methods: 306 are shown, 0 are hidden)

Hidden	Name	Occurrence	Source Code Comments
☐	weka.core.Instances	Class	Class for handling an ordered set of weighted instances
☐	weka.classifiers.OneR	Class	Class for building and using a 1R classifier. For more info
☐	weka.core.Instance	Class	Class for handling an instance. All values (numeric, nom
☐	weka.core.Attribute	Class	Class for handling an attribute. Once an attribute has b
☐	weka.classifiers.Classifier.<init>	1	
☐	weka.classifiers.Evaluation.<init>	2	
☐	weka.classifiers.Evaluation.correct	2	Gets the number of instances correctly classified (that is
☐	weka.classifiers.Evaluation.crossValidateMo...	1	Performs a (stratified if class is nominal) cross-validation
☐	weka.classifiers.Evaluation.evaluateModel	10	Evaluates a classifier with the options given in an array
☐	weka.classifiers.Evaluation.evaluateModel	1	Evaluates a classifier with the options given in an array

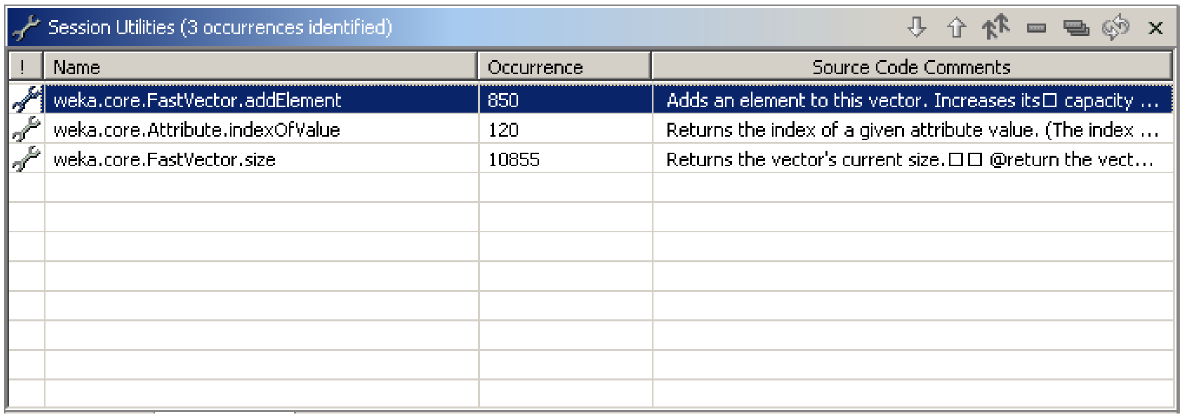
Figure 21 A Model view displaying nodes from a CTF model

Pattern view (See Figure 22) is used to display reoccurrence patterns, that is repeated subtrees, identified during an exploration, and *Utility view* (See Figure 23) is used for displaying utility methods. Patterns and utilities can be promoted to the global level so different traces can share the results of an exploration. Where a trace is first loaded for exploration, it will check patterns and utility methods existing at the global level and will use them to filter nodes in current trace.


Patterns (34 occurrences identified)

!	Name	Occurrence	Source Code Comments
	weka.core.Instances.numAttributes	8510	Returns the number of attributes.□□ @return the num.
	weka.core.Instances.attribute	5008	Returns an attribute.□□ @param index the attribute's i.
	weka.core.Attribute.indexOfValue	120	Returns the index of a given attribute value. (The index .
	weka.core.Instances.add	576	Adds one instance to the end of the set. □ Shallow copi.
	weka.core.Instances.<init>	3	
	weka.core.Instances.<init>	36	
	weka.classifiers.Evaluation.<init>	2	
	weka.core.Instances.numClasses	13	Returns the number of class labels.□□ @return the nu..
	weka.core.Instances.classAttribute	50	Returns the class attribute.□□ @return the class attrib.
	weka.core.Attribute.value	6	Returns a value of a nominal or string attribute. □ Retur.

Figure 22 A Pattern view displaying repeated node and subtrees identified in a trace

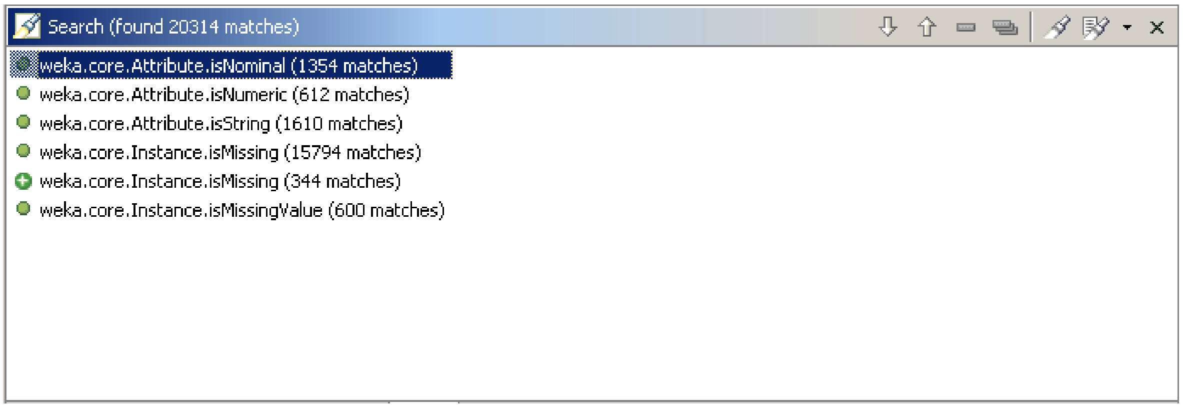


Session Utilities (3 occurrences identified)

!	Name	Occurrence	Source Code Comments
	weka.core.FastVector.addElement	850	Adds an element to this vector. Increases its☐ capacity ...
	weka.core.Attribute.indexOfValue	120	Returns the index of a given attribute value. (The index ...
	weka.core.FastVector.size	10855	Returns the vector's current size.☐☐ @return the vect...

Figure 23 A Utility view displaying utility methods identified for the current session

SEAT provides search ability by incorporating the *Search* view from the Eclipse search framework (See Figure 24). It supports both 'exact' and 'contains' matching. An intelligent search capability was developed by Liu [88]; this scheme can help in assisting user exploration. Search history is recorded automatically and the user can easily return to a previous result.

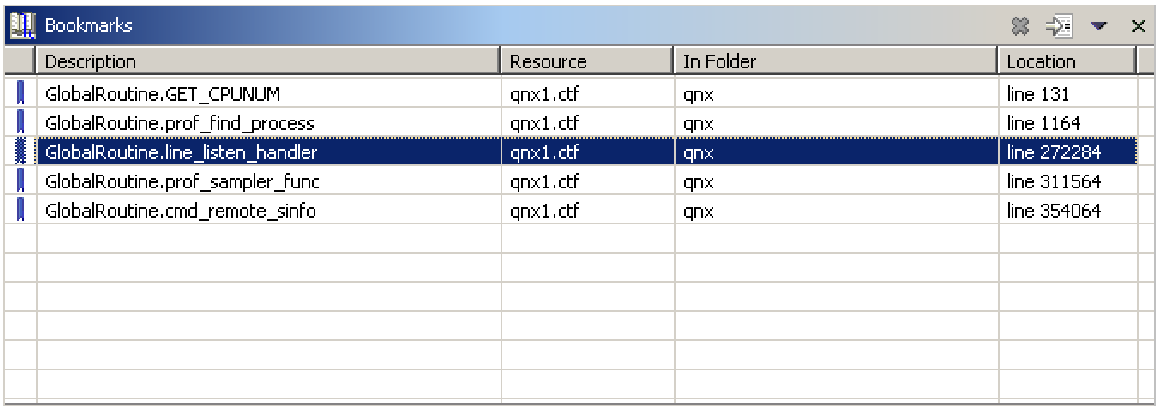


Search (found 20314 matches)

```
weka.core.Attribute.isNominal (1354 matches)
weka.core.Attribute.isNumeric (612 matches)
weka.core.Attribute.isString (1610 matches)
weka.core.Instance.isMissing (15794 matches)
weka.core.Instance.isMissing (344 matches)
weka.core.Instance.isMissingValue (600 matches)
```

Figure 24 A Search view displaying all trace methods that starts with 'is'

A *Bookmark* view is also provided so the user can record the locations previously visited and return to it when necessary (See Figure 25).


Bookmarks

Description	Resource	In Folder	Location
GlobalRoutine.GET_CPUNUM	qnx1.ctf	qnx	line 131
GlobalRoutine.prof_find_process	qnx1.ctf	qnx	line 1164
GlobalRoutine.line_listen_handler	qnx1.ctf	qnx	line 272284
GlobalRoutine.prof_sampler_func	qnx1.ctf	qnx	line 311564
GlobalRoutine.cmd_remote_sinfo	qnx1.ctf	qnx	line 354064

Figure 25 A Bookmark view displaying saved locations during an exploration

All views described above support navigation in the editor. When a node in a view is selected, the editor will be refreshed automatically to reveal the first matched tree node. The user can further select to navigate to other matched tree nodes in turn if there is a multiple match. *Model* view, *Patte*rn view, *Utility* view, *Search* view and *Bookmark* view correspond to *ModelView*, *PatternView*, *UtilityView*, *SearchResultView* and *BookMark-View* class respectively in the diagram shown in Figure 26.

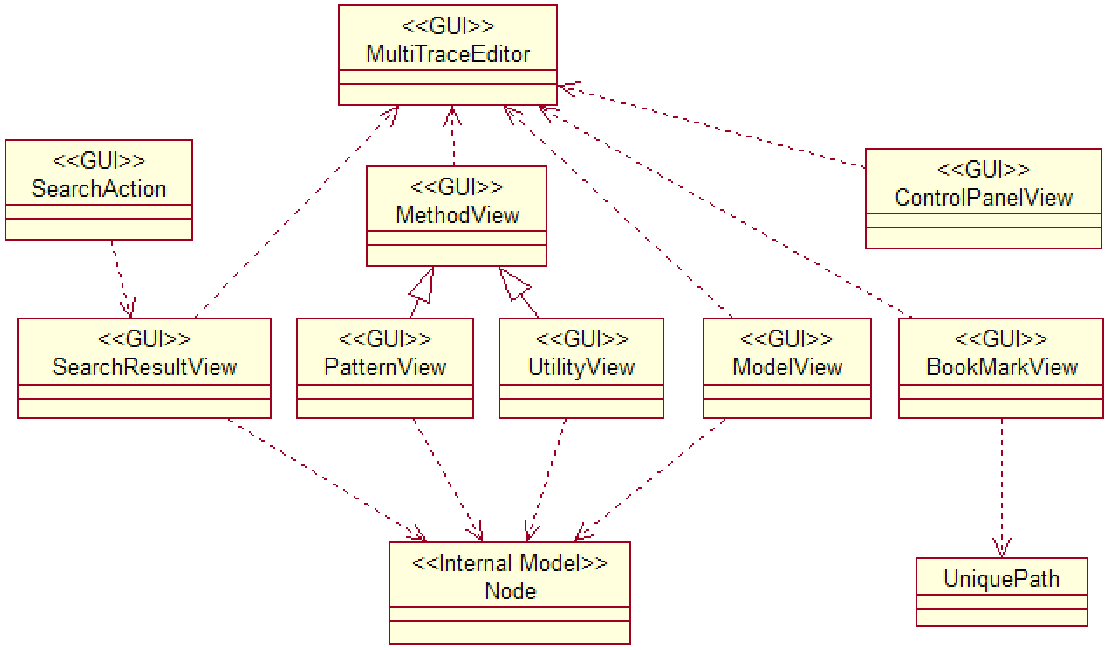


Figure 26 Class diagram for different views in SEAT

Chapter 5 Dynamic Trace Loading Algorithm

In Chapter 4, we introduced the architecture of SEAT and different subcomponents. In this chapter we will concern ourselves with trace loading techniques.

In essence, a trace can be regarded as a tree structure. Trees are one of the most important data structures in real-world applications; and many types of data can be converted to trees. Widgets for displaying a tree exist in all major GUI toolkits, however they have a major problem: They require the whole tree structure to be modelled and passed to the widget. This leads to two inherent problems: time and space constraints. Specifically, those tree nodes that are currently invisible in the window need also to be processed.

Our proposed approach to this problem is to only retrieve currently visible nodes for the UI tree using a dynamic trace-loading algorithm. The direct result of the algorithm is a tree that is only comprised of visible tree nodes and their linking parents. We call this tree, a *PartialTree*.

5.1 Design Rationale

The design rationale for *PartialTree* is program locality, i.e., many programs exhibit locality when accessing memory [89]: certain data are referred to repeatedly for a while, and then the programs shift focus to other parts of the data. The currently focused data is called the “working set”. In the working set model, the working set of a process refers to the minimum collection of its pages that must reside in the main memory for the process to run without ‘un-necessary’ page faults. This concept is utilized by many operating systems for performance enhancement.

5.1.1.1 Why not cache?

It is difficult to determine which part of the tree to cache. With trees, unlike with linear data structures, such as lists, nodes of interest change quickly as users expand and collapse nodes. Nodes can also be filtered by a filtering algorithm sparsely. Therefore, nodes in a cache will become obsolete quickly and the cache will be useless.

The trace model or underlying storage may already have provided some kinds low-level cache for improved performance. For example, when retrieving nodes from a database, the database management system may already provide such facilities.

Therefore, dynamic loading is a better choice.

5.2 Concepts

Node: A node is a vertex in a graph. A node, also called a model node, represents a distinct entity in a CTF model.

Edge: If a node i is associated with node j, the association is called an edge from initial node i to the terminal node j. Extra information can be encoded in an edge, such as the number of repetition for a looped method call.

Path: A path is a traversal of consecutive nodes along a sequence of edges in a graph. The node at the end of one edge in the sequence is also the node at the beginning of the next edge in the sequence. The first node of a path is called root node and the last node in a path is called terminal node.

UniquePath: A *UniquePath* is an implementation of *path* in Java using a list of edges. In a CTF trace model, a node can stand for occurrences of method calls in different places in a call tree. By giving only a node, one cannot clearly identify its exact location. Therefore, we use *UniquePath* to represent a distinguishable tree node in a tree. In the following sections, we use path and *UniquePath* interchangeably.

Subtree: A subtree of a tree is a tree node in that tree together with all its descendents. When a tree node in a tree has no children, it is called a tree node.

PartialTree: A *PartialTree* is a tree generated as the user explores a trace dynamically, and contains only paths that are visible in the current view. It may also contain intermediate parent paths of visible paths that are used to link the tree structure, but these parent paths should be invisible to the user when visualized.

Dynamic Loading Algorithm: Algorithm that constructs a *PartialTree* as the user is exploring the trace.

5.3 Design of the Dynamic Loading Algorithm

5.3.1 Overview of Design

Dynamic loading is an independent layer between the trace model and trace visualization. Though it closely interacts with these two layers, our design guideline is to try to make it as reusable as possible. So it can communicate with different models and be rendered with various visualization techniques.

5.3.2 UniquePath

A *UniquePath* uses a list to remember the edges from root node to a terminal node in a model. Because in a model representation of a trace, such as in CTF, a node itself cannot be used to identify its location in a tree, *UniquePath* solves this problem by incorporating information of all edges from the root node to the terminal node.

The most important methods of *UniquePath* are *getPreviousSibling()* and *getNextSibling()*. A sibling path is the path that has same parent as the current path. These methods are used to extract data from the model and to construct a *PartialTree* when exploration is made. The current implementation uses the index of last edge in the list of a *UniquePath* to find its neighbouring edge and creates the required path. *NULL* will be returned when the desired edge does not exist. For example, the first edge will not have an edge before it. The other two methods are *createParentPath()* and *createChildPath()* that are used to navigate through parent/child relation. The parent path of a *UniquePath* can be easily created by simply removing the last edge in the list of current *UniquePath*. A child path will be created by adding the child edge to the end of the edge list. If models other than CTF are used, these methods should be overridden.

5.3.3 PartialTree

IDynamicTree defines the interface for a dynamically loaded tree so that different implementations can be realized. *PartialTree* implements this interface and encapsulates the dynamic loading algorithm. We will first introduce the basic pieces of functionality provided in *PartialTree* to support dynamic data loading.

5.3.3.1 Determine whether a path will be included in PartialTree construction.

Visually hiding or showing a path depends on a combination of four conditions: a tree global setting that determines whether the filtered paths will be shown, a parent path setting that overrides the global settings, the hide/shown state of the path itself, and user specified filter criteria at the model level.

5.3.3.2 Cache expanded/collapsed state of a path.

A hashtable is used to record those paths that are expanded so their children will be shown in the tree. Also the state of expansion is identified as either fully expanded or partially expanded. The state is important in supporting additional features of the new widget that will be discussed later.

5.3.3.3 Calculate the size of a complete tree based on the expand/collapse states of paths.

The size is the total number of nodes of a tree if dynamic loading is not used. As will be seen in the design of the new widget, this size can be used to determine the presentation of the scrolling to help correct visualization. The calculation is based on the hashtable for expanded nodes.

5.3.3.4 Calculate offset of current path to the root.

The offset determines how many nodes are “before” this path starting from the root path. It is also used to help visualization by indicating the current position of user scrolling.

The algorithm iterates through the expanded nodes list and uses comparison to determine which node is before the current path and which is after or equal to it.

5.3.4 Dynamic Loading Algorithm

From a high-level view, the dynamic loading algorithm will search children, siblings and parent nodes from a model in turn to retrieve candidate paths and create the *PartialTree* using the retrieved paths. To construct a *PartialTree*, the dynamic loading algorithm uses the following steps:

5.3.4.1 Forward skip n paths from a given path.

Skipping is used to relocate the starting point for the next *PartialTree*. From the given path, *n* paths will be skipped to reach the new starting point and the new *PartialTree* will be constructed from the new position. In case there are not enough paths to be skipped, the algorithm will skip the maximum number that is available and return that number. Forward skipping consists of three consecutive steps and successive steps will only continue if a previous step has not found enough paths.

1. Skip current path as a subtree. Each child path of current path will be skipped. The child path is treated as a subtree so the skipping is done recursively. This method will return with either the desired number of paths or the maximum number of path skipped in the subtree.

2. Skip sibling paths. Sibling paths that are after the current path are skipped if not enough paths have been skipped in the previous step. Each sibling path is treated as a subtree and skipped using the skipping path method in step one.

3. Skip sibling paths of all parent paths if not enough paths have been linked in previous step. Each parent path is skipped using the skip sibling method in step two until a given number of paths have been skipped or the root path has been searched. The sibling paths must be after the parent path. Repeat this step until the root path is done if no enough paths have been linked.

4. The number of paths skipped from these steps will be returned.

5.3.4.2 Backward skip n paths from a given path.

Backward skipping is used to relocate the starting point in the reverse direction. That is from the current location, it will find *n* paths before it and use that location as the starting point for the new *PartialTree*. The algorithm will also skip the maximum number that is available and return that number. This is comparable to the forward skipping which has three consecutive steps; backward skipping has only two steps since backward skipping will not skip the current path as a subtree as the first step.

1. Skip sibling paths that are before the current path if not enough paths have been skipped. Each sibling path is treated as a subtree and calls skipping child path method described below.

2. Skip sibling paths of all parent paths if not enough paths have been linked in the previous step. Each parent path is skipped using the skip sibling method in the previous step until a give n number of paths have been skipped or the root path has been found. The sibling paths must be before the parent path; Repeat this step until the root path is found if not enough paths have been linked.

3. Skip child path. The method will skip a child path from largest index first instead of smallest index as in forward skipping. Each child path of the current path will also be recursively counted and skipped. This skipping path method will be called by the skipping sibling path method.

4. Return the total number of paths skipped backwards after all the skipping calls.

5.3.4.3 Construct a New PartialTree from a given path.

This is the core method of the loading algorithm and it calls other methods described in this section to construct a *PartialTree* from a given path. The method takes three parameters: the starting path (the path that is used as a reference point), an offset from the starting path, and number of paths to be retrieved. The algorithm will use the starting path as a reference point, skip the number of paths indicated by the offset, and construct a *Partial-Tree* with the given number of paths.

1. Determine the new loading point. Depending on the offset, which can be positive or negative, the forward skip or backward skip method is called and current path will be move to a new location. All other paths will be retrieved starting from this location.

2. Link paths to construct the PartialTree. The linking process is the same as the forward skipping process except that it actually creates and links paths to construct the partial tree instead of counting the numbers. Similarly the linking process also has three constituent steps which include **linking the current path as a subtree**, **linking sibling nodes and linking parent nodes** until the desired number of paths is constructed. If there are not enough paths to be linked, the algorithm will process the maximum available.

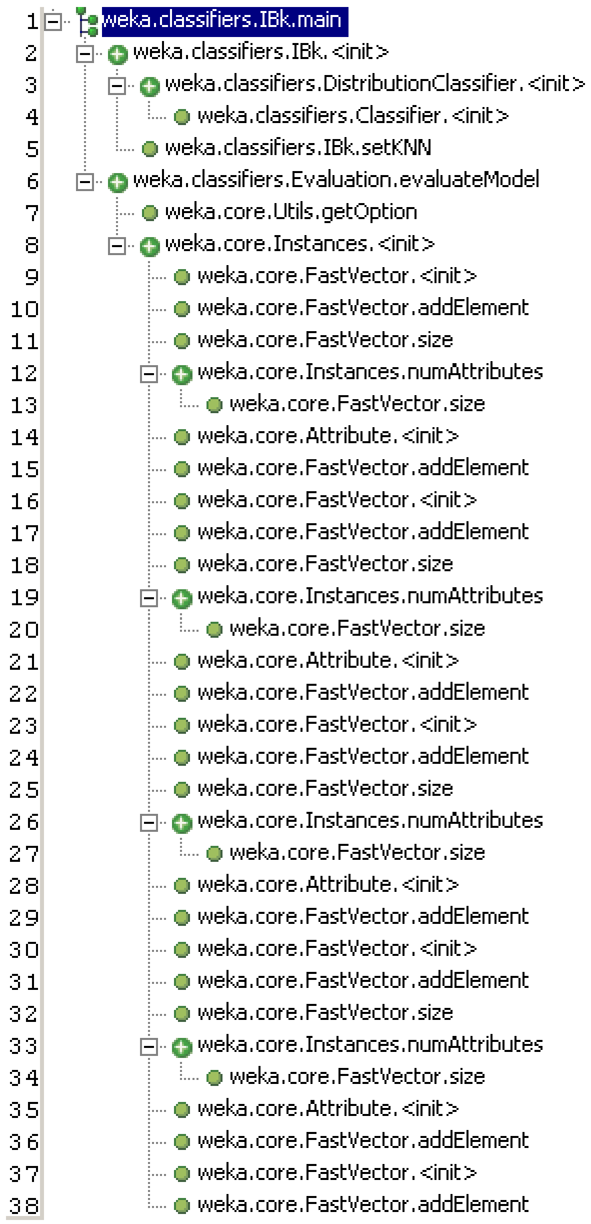
3. Return the root of the PartialTree created.

5.4 Illustration of Dynamic Loading Algorithm

To help better understanding the dynamic loading algorithm, a sample trace is given to show how a *PartialTree* is dynamically constructed.

5.4.1 Sample Trace

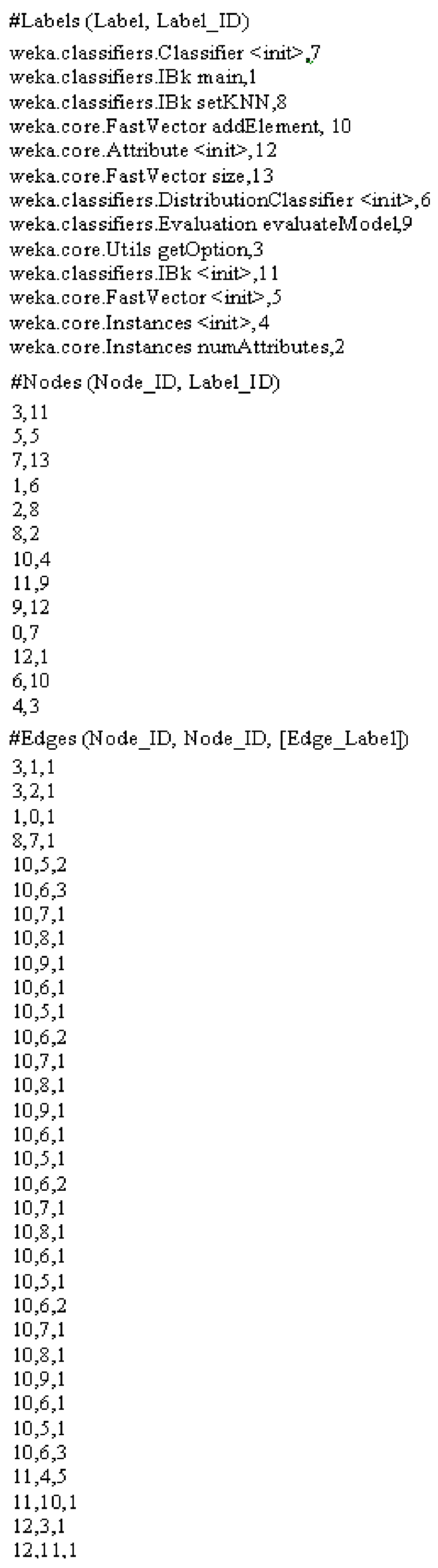
The sample trace is abridged from a real single-threaded trace without timing information (See Figure 27).



```
 1 weka.classifiers.IBk.main
 2   weka.classifiers.IBk.<init>
 3     weka.classifiers.DistributionClassifier.<init>
 4       weka.classifiers.Classifier.<init>
 5     weka.classifiers.IBk.setKNN
 6   weka.classifiers.Evaluation.evaluateModel
 7     weka.core.Utils.getOption
 8     weka.core.Instances.<init>
 9       weka.core.FastVector.<init>
10       weka.core.FastVector.addElement
11       weka.core.FastVector.size
12       weka.core.Instances.numAttributes
13         weka.core.FastVector.size
14       weka.core.Attribute.<init>
15       weka.core.FastVector.addElement
16       weka.core.FastVector.<init>
17       weka.core.FastVector.addElement
18       weka.core.FastVector.size
19       weka.core.Instances.numAttributes
20         weka.core.FastVector.size
21       weka.core.Attribute.<init>
22       weka.core.FastVector.addElement
23       weka.core.FastVector.<init>
24       weka.core.FastVector.addElement
25       weka.core.FastVector.size
26       weka.core.Instances.numAttributes
27         weka.core.FastVector.size
28       weka.core.Attribute.<init>
29       weka.core.FastVector.addElement
30       weka.core.FastVector.<init>
31       weka.core.FastVector.addElement
32       weka.core.FastVector.size
33       weka.core.Instances.numAttributes
34         weka.core.FastVector.size
35       weka.core.Attribute.<init>
36       weka.core.FastVector.addElement
37       weka.core.FastVector.<init>
38       weka.core.FastVector.addElement
```

Figure 27 A sample trace

The CTF formatted trace without timing information is shown in Figure 28:



```
#Labels (Label, Label_ID)
weka.classifiers.Classifier <init>,7
weka.classifiers.IBk main,1
weka.classifiers.IBk setKNN,8
weka.core.FastVector addElement, 10
weka.core.Attribute <init>,12
weka.core.FastVector size,13
weka.classifiers.DistributionClassifier <init>,6
weka.classifiers.Evaluation evaluateModel,9
weka.core.Utils getOption,3
weka.classifiers.IBk <init>,11
weka.core.FastVector <init>,5
weka.core.Instances <init>,4
weka.core.Instances numAttributes,2
#Nodes (Node_ID, Label_ID)
3,11
5,5
7,13
1,6
2,8
8,2
10,4
11,9
9,12
0,7
12,1
6,10
4,3
#Edges (Node_ID, Node_ID, [Edge_Label])
3,1,1
3,2,1
1,0,1
8,7,1
10,5,2
10,6,3
10,7,1
10,8,1
10,9,1
10,6,1
10,5,1
10,6,2
10,7,1
10,8,1
10,9,1
10,6,1
10,5,1
10,6,2
10,7,1
10,8,1
10,6,1
10,5,1
10,6,2
10,7,1
10,8,1
10,9,1
10,6,1
10,5,1
10,6,3
11,4,5
11,10,1
12,3,1
12,11,1
```

Figure 28 Sample trace in CTF format without timing information

5.4.2 Illustration

We make the following assumptions to simplify the illustration steps. First the tree is fully expanded, that is all the subtrees are expanded and all leaf tree nodes are visible as shown in Figure 27. Second we assume that the current window can display 10 tree nodes at most, so the *PartialTree* to be constructed will consist of 10 visible *UniquePath*s. We also use the word *"node"* to refer to the tree node for simplifying illustration only. It should not be confused with "*model node*" defined in 5.2.

5.4.2.1 Construct the Initial PartialTree

Initially, the *PartialTree* will be constructed from the root node. Therefore, node 1 is first included and its child nodes will be linked recursively. Because node 1 has two children, node 2 and node 6, they are retrieved recursively. Node 2 is then included. Node 3 is child of node 2 so both itself and its only child node 4 will be linked. After node 3 is done, its sibling node 5 is linked to node 2. Till now subtree node 2 is finished. Because there are still not enough nodes, the siblings of node 2 will be searched. Node 6 will also be linked recursively until 10 nodes are found. This initial *PartialTree* is marked as *PartialTree* 1 in Figure 29.

5.4.2.2 Forward Skip 6 Lines and Construct New PartialTree

Next we will skip 6 lines so the starting point for the new *PartialTree* is node 7. Forward skipping is similar to linking. Node 1 is first included and counted as 1. Then it child node 2 will be counted recursively. After node 2 is finished, the counter will be 5. Node 6 will then be processed recursively to get extra node. Here only one extra node is needed to skip 6 nodes in all, so node 6 will be counted. After the desired number of nodes is counted, one extra node is skipped to reach the next starting point. Therefore, node 7, the first child of node 6, is set as the starting point of new *PartialTree*. Loading of new *PartialTree* will start from node 7.

Before construct new *PartialTree*, all intermediate parents of node 7 is first linked. They include node 1 and node 6. Next node 7 is linked to its directly parent node 6. Because node 7 has no children, its sibling, node 8, will be linked recursively until node 16 is reached. The process is similar to that described in 5.4.2.1. The new *PartialTree* is indicated as *PartialTree* 2. Node 1 and node 6 are also marked with number 2 and indicate that they also belong to *PartialTree* 2.

Long jump to the last page is also supported. The number of long jump steps is passed from user interface, such as scrollbar. Then new starting point from root node is first calculated. The calculated value minus current starting point is the number of nodes to be skipped. The remaining steps will be the same as previously introduced. The *PartialTree* and intermediate linking nodes are marked by 3.

5.4.2.3 Backward Skip and Construct New PartialTree

Suppose a *PartialTree* starting from node 14 is displayed and we will scroll backward so the new starting point is node 3. In this scenario, 11 nodes will be skipped in a backward move.

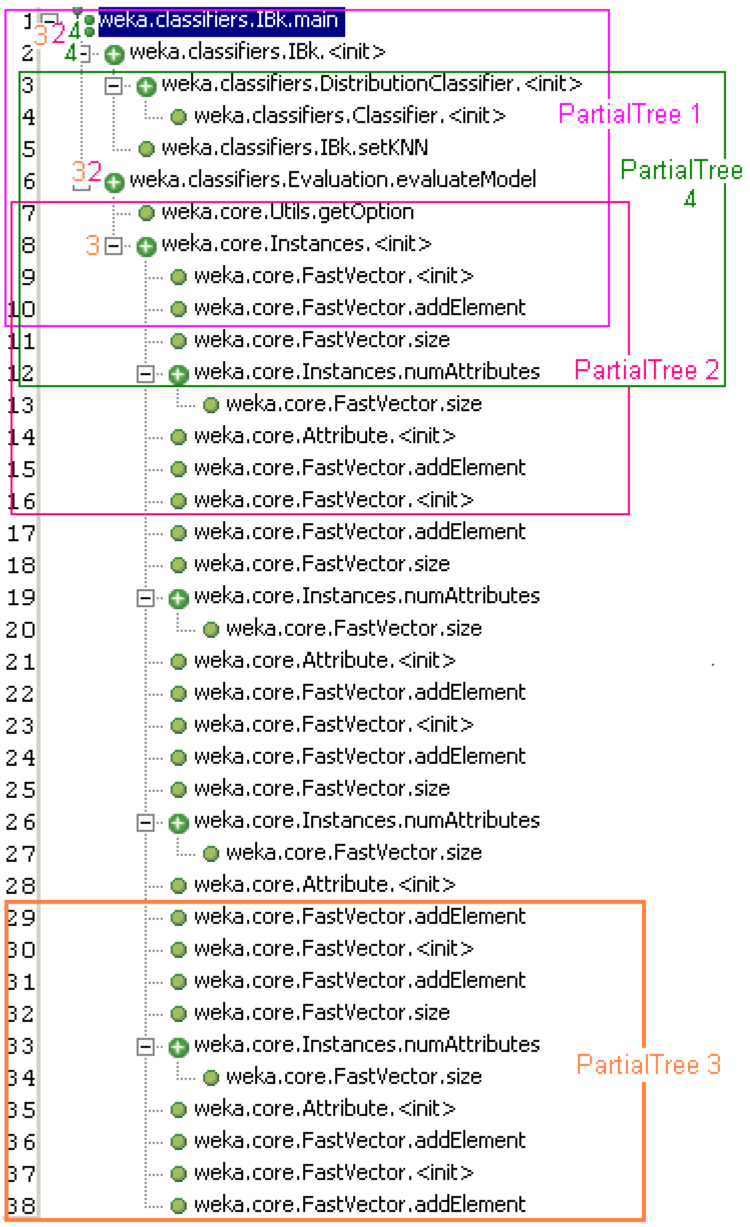
The first step is to skip previous sibling as a subtree. In this case, node 12 is processed and its children will be skipped recursively starting from highest index to lowest index. This will result in that the last child deep in the subtree structure will be counted first. Node 12 only has one child, node 13. Node 13 will be counted as one. Then node 12 is counted. All siblings before node 12 is counted in turn until the parent, node 8, is reached. Because no enough nodes have been found, node 7 and node 6 will be skipped as sibling node and parent node of node 8. Then node 2 will be skipped as the sibling of node 6 and the search is done by skipping the last child, node 5. Node 3 will also be recursively skipped by skipping its last child first. The only one, node 4, is counted, and the new starting point is positioned to node 3.

From node 3, a new *PartialTree* will be constructed by forward linking. The result is *PartialTree* 4. Node 1 and node 2 are also marked as being the intermediate parents in the *PartialTree*.

5.4.3 When New PartialTree Construction Occurs

There are different actions from user interface that may cause a new *PartialTree* being constructed. For example, when a user navigate in the trace editor, using either keyboard or mouse, a *PartialTree* will be created and rendered for each movement. Expanding and collapsing a subtree will also cause new tree nodes to be displayed or hidden. When filtering algorithms or user defined filtering criteria are applied or invoked, a new retrieval will occur to reflect changes. A global setting change, such as do not hide filtered tree nodes, will also cause the editor to be refreshed and thus results in a new *PartialTree* to be created.

No matter which way causes a new *PartialTree* construction, the process is same. The dynamic loading algorithm will first test whether a model node will be included in the retrieval or not. Because the algorithm works on model nodes, the memory footprint of *PartialTree* is very small and performance of overall tree manipulation is improved.



```
1  weka.classifiers.IBk.main
2    weka.classifiers.IBk.<init>
3      weka.classifiers.DistributionClassifier.<init>
4        weka.classifiers.Classifier.<init>
5      weka.classifiers.IBk.setKNN
6    weka.classifiers.Evaluation.evaluateModel
7      weka.core.Utils.getOption
8      weka.core.Instances.<init>
9        weka.core.FastVector.<init>
10       weka.core.FastVector.addElement
11       weka.core.FastVector.size
12       weka.core.Instances.numAttributes
13         weka.core.FastVector.size
14       weka.core.Attribute.<init>
15       weka.core.FastVector.addElement
16       weka.core.FastVector.<init>
17       weka.core.FastVector.addElement
18       weka.core.FastVector.size
19       weka.core.Instances.numAttributes
20         weka.core.FastVector.size
21       weka.core.Attribute.<init>
22       weka.core.FastVector.addElement
23       weka.core.FastVector.<init>
24       weka.core.FastVector.addElement
25       weka.core.FastVector.size
26       weka.core.Instances.numAttributes
27         weka.core.FastVector.size
28       weka.core.Attribute.<init>
29       weka.core.FastVector.addElement
30       weka.core.FastVector.<init>
31       weka.core.FastVector.addElement
32       weka.core.FastVector.size
33       weka.core.Instances.numAttributes
34         weka.core.FastVector.size
35       weka.core.Attribute.<init>
36       weka.core.FastVector.addElement
37       weka.core.FastVector.<init>
38       weka.core.FastVector.addElement
```

Figure 29 Sample trace with different PartialTrees marked for illustration

Chapter 6 New Tree Widget

Trees are a convenient hierarchical structure to organize large data sets. By partitioning data into different level of groups, we can examine data sets at different level of abstraction methodically [90]. We can move down the hierarchy if interesting features appear, or go up if sufficient information about sub trees has been collected. This chapter will describe the design of a new tree widget called *PictureTree* that utilizes the dynamic data loading techniques presented in Chapter 5.

6.1 Classification of Tree Layout Possibilities

There are many visualization methods for a tree structure depending on the requirements and domain. Some of these are described below and illustrated in Figure 27.

- Simple text layout: indented list, like Windows Explorer.
- Traditional graphic layout: parent-child link drawing
- Containment tree: Instead of representing a tree as nodes and links, a tree is represented as a containment-TreeMap [38]. In this representation, children nodes are displayed in the space of a parent node. Space for a parent node is divided horizontally and vertically for all direct children and the division is repeated until leaf nodes are reached. TreeMaps are effective in presenting unusual patterns at the leaf level [51].
- Radial layout: This involves displaying the root node at the center with all children scatter around. Layout algorithms can be applied to place different level of children. This view can be zoomed out to display a large number of nodes, but the

drawback is that it cannot clearly display the hierarchy of the tree as intuitively as other approaches.

- Collapsible Cylindrical Trees [91]: Child nodes are drawn on the surface of rotating cylinders, which will be automatically displayed, or hidden depending on their three-dimension position. Each label is drawn on a facet. If the number of child nodes outnumbers the facets, the remaining nodes will be mapped and drawn dynamically. At any time, only one expanded cylinder is drawn on the right side of the current cylinder. The right side cylinder is the set of child nodes of the focused node from the parent cylinder, and all other cylinders are squeezed.

- 3D ConeTree: This uses cones to represent subtrees and draws one node at the apex and all its children nodes in the circular base of the cone. Refer to the Chapter 2 for details.

Auxiliary techniques such as animation [45], zooming and focus + context [32] are also often used.

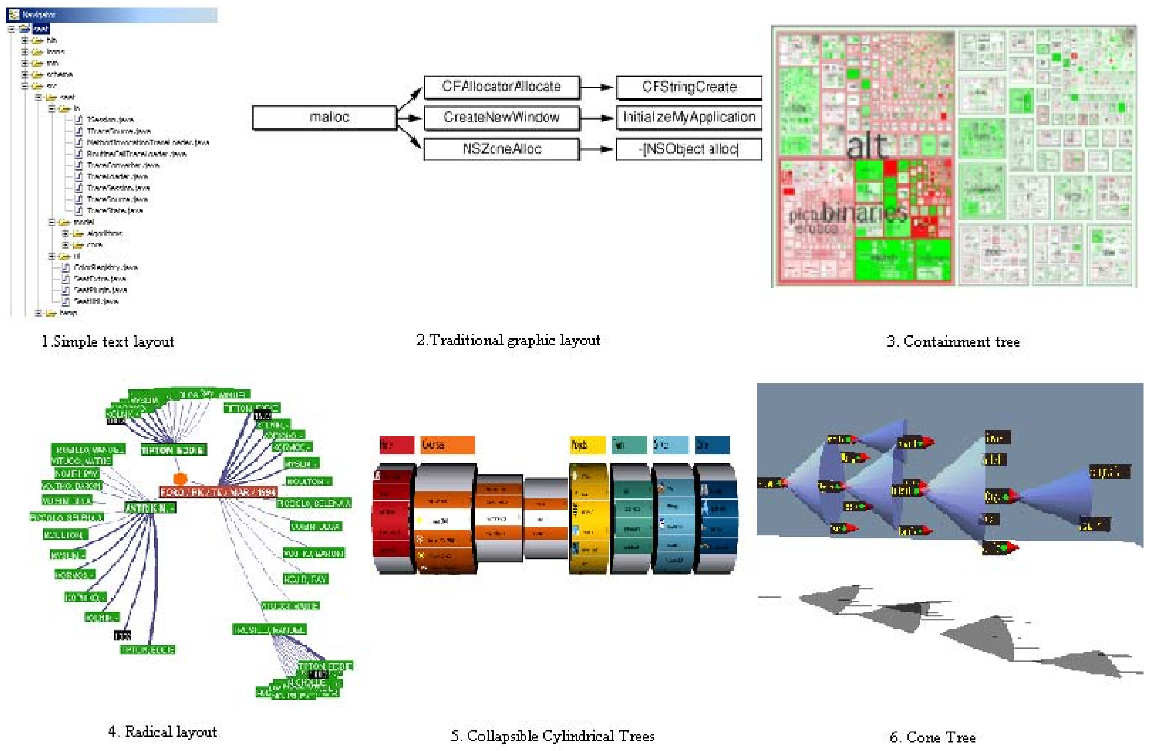


Figure 30 Different tree layout and visualization techniques

6.2 Requirements for Tree Layout

The most influential guidelines for drawing trees are those originally proposed for drawing binary trees, but which are applicable to all trees [92]:

- A parent should be drawn above its children.
- Nodes at the same level should lie along a horizontal line or vertical line.
- A sub tree should look the same regardless where it is drawn.

Another set of requirements for aesthetically displaying hierarchical information such as tree is summarized as follows [93]:

- Allow adequate space between nodes to display readable information
- Allow users to understand the relationship between a node and its surrounding context.
- Allow users to find elements in the hierarchy quickly.

- Allow the whole information space to fit into a bounded region. This requirement is desirable to eliminate the need for scrolling.

6.3 Existing Solutions

An example of a tree-browsing tool that avoids loading all the data into memory is Virtual TreeView [94]. It is a tree control based on the event concept and defers data loading until absolutely necessary. Though it is very efficient in loading thousands of nodes, when the number of nodes reaches the millions, the widget ceases to response when it is evaluated.

SpaceTree [95] is based on the conventional tree node-link layout, but adds dynamic zooming features that lay out branches of a tree to best fit the available screen space. Branches that cannot fit on to the screen are summarized and indicated as a triangle at the side of the parent nodes. The shading of the triangle corresponds to the children's size, and the height corresponds to the depth of the sub tree; the width indicates the average width. Animation techniques are employed to reflect change of focus. When a new focus is selected, the SpaceTree evaluates how many new branches will be opened, trims the branches that overlap with the new branch and then centers the new tree. To help user orientation, SpaceTree uses all the previous focus nodes as landmarks, and highlights this path using the colour blue. Users can navigate the tree by clicking the triangle to show branches, or by clicking arrows to move among parent, children and siblings. However, SpaceTree loads all the nodes required for visualization into the memory at once.

6.4 Design Decisions for the New Widget

The following are some problems with existing widgets on the Eclipse platform motivate us to develop a new tree widget instead of using the old ones.

- The existing technology cannot handle large data. Filtering is done at the user interface level, not the backend model level. When filtering needs to be done in a user interface widget, data are first created. The result is slow response and relatively high memory requirements.

- The behaviour of a tree control can not be fully controlled as desired by users because some operations on tree control are delegated to the underlying operating system and the operating system gets the priority to do processing first. Therefore, described feature cannot be realized.
- Operations during trace exploration require extra states to be represented by a node that are not supported by traditional widgets. Most technologies either consider a node to either have all its children hidden, or all of them displayed (i.e. using the plus/minus icons). As discussed earlier, there is a need to show in addition that a subset of children are displayed.

Different layouts for tree rendering were compared and investigated to create the new tree widget. However, our decision had to be based on supporting the expected tasks [94]. Because of the limitations of screen space, we decided that only a portion of a tree will be displayed. Other nodes may be out of the view, pruned to be invisible or displayed in the background. We decided to use the simple text layout that is most familiar to the users and is widely used. Another consideration is that we want to be able to substitute the new tree widget as transparently for the users as possible.

6.5 Design of PictureTree

The *PictureTree* class is developed in Eclipse platform using SWT. SWT is a set of components that deliver native look and feel functionality for the Eclipse Platform. Applications developed using SWT API are fully portable to different platforms because SWT is implemented in Java. Another benefit of SWT is that SWT-based applications can interact with platform-specific features.

6.5.1 Tree Navigation

By studying the existing tree widgets in different platforms, we determined that users navigate tree widgets by using the following keys and mouse clicks. (The term *page* used hereafter refers to a fixed amount of tree nodes that is rendered in the window of a widget).

- Home: display the first page starting from the first node (root).
- End: display the last page to the last node.
- PageUp: reveal the previous page, if not enough nodes can be retrieved, display the first page from the root.
- PageDown: reveal the next page, if not enough nodes can be retrieved, display the last page.
- ArrowUp: Move the current selection up if it is not the first visible node in the page; otherwise, move the page up so one previous node in the tree is visible, and select it.
- ArrowDown: Move the current selection down if it is not the last visible node in the page; otherwise, move the page down by one so one subsequent node in the tree is made visible, and select it.
- ArrowLeft: If current node is an expanded subtree, collapse the subtree; otherwise, move the selection to the parent node; if the parent node is not visible in the current window, make it the first visible node in the window.
- ArrowRight: If current node is not a subtree, do nothing. If the current node is an expanded subtree, go to its first child node; otherwise, expand the current node.
- Backspace: move to the parent node of the current node; reveal it as the first visible node if it was not already visible.
- Mouse Click: If the user clicks on a label, select it as the current node; if the user clicks on an expand/collapse icon, expand or collapse the subtree respectively.
- Mouse Click+Ctrl: Make a selection of multiple non-contiguous nodes; the node selected by mouse click is marked as the current node.
- Mouse Click+Shift: make selection on multiple contiguous nodes from a previous selected node to the node selected by mouse click.

PictureTree will support all the common features described above.

6.5.2 New Features

One novel feature of this new widget is supporting subtree level filtering through an additional expanded state, called 'partially expanded'. We made the decision to implement this based on the observation that software engineers need constantly to change their ideas and assumptions when exploring traces. For example, a software engineer may want to hide all the filtered tree nodes. When exploring a specific subtree, he or she may wish to see all the tree nodes of the subtree, including those marked filtered so he or she can have all the interaction details.

When a subtree is first displayed, it is in collapsed state. When a user wants to expand the subtree to visit its children nodes, the icon that represents the new state will change to either '–' icon or '~' icon depending 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 indicated some children cannot be seen. By using Ctrl+Click, the user can switch between these two icons, and refresh children nodes respectively.

The states of a subtree node in the new widget are depicted as in Figure 28. A leaf node does not have these states.

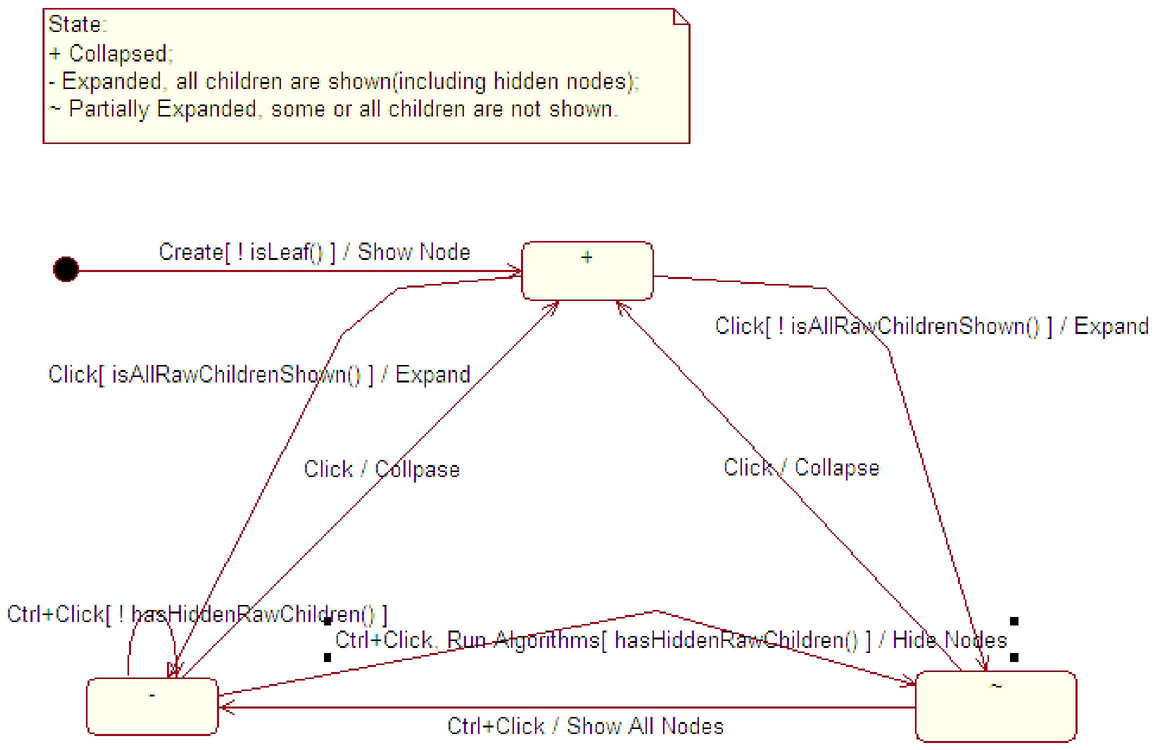

State:
+ Collapsed;
- Expanded, all children are shown(including hidden nodes);
~ Partially Expanded, some or all children are not shown.
Create[! isLeaf()] / Show Node
+
Click[! isAllRawChildrenShown()] / Expand
Click[isAllRawChildrenShown()] / Expand
Click / Collpase
Click / Collapse
Ctrl+Click[! hasHiddenRawChildren()]
Ctrl+Click, Run Algorithms[hasHiddenRawChildren()] / Hide Nodes
-
~
Ctrl+Click / Show All Nodes

Figure 31 State diagram of a subtree node

6.5.3 PictureLabel

PictureLabel is a basic visual item used to render a *UniquePath* in a *PartialTree*. It contains the icon representing the expanded state, the icon standing for the path, the foreground colour, the background colour and the path as its associated data elements. *PictureLabel* only contains information for visualization so it is independent of any rendering method. If another representation of the tree is to be used, such as a drawing algorithm that is different as *PictureTree* in 6.5.4, no change in *PictureLabel* is needed.

6.5.4 PictureTree

The new tree widget is called *PictureTree* and it extends the *Canvas* class of SWT on the Eclipse Platform. *Canvas* is the starting point for a drawing-related widget that is portable to different platforms. *PictureTree* is designed to simulate the interface of a *Tree* widget in SWT. It supports selection of tree nodes, querying of tree attributes like first item and node count, and adding listeners. Mouse/Keyboard functionalities as described in 6.5.1 are supported so it looks exactly like a traditional *Tree* widget provided by the Eclipse platform. State changes of a node are captured using the *onMouseDown()* method which switches the icon of a subtree between +, - and ~.

To ease the incorporation of other visualization methods, all the code that renders the tree is encapsulated into a single *onPaint()* method. Input to the drawing method is the *PartialTree*, which contains all the nodes to be drawn by a layout. To reduce blinking between two drawings, double buffering is used.

One of the challenges in rendering is correct positioning of the scrollbar to indicate current location according to the size of the entire tree. Accurate calculation of position is necessary to ensure the user has a proper understanding of where he or she is in the tree. Because the functionality of a *PictureTree* is visualization, it delegates the calculation to the model – *PartialTree* so other visualizations may also benefit from this model/view separation. *PictureTree* will query *PartialTree* to get the offset of the first node to the root and adjust the position the *syncScrollBars()* method.

6.6 Design of PartialTreeViewer

PartialViewer is a JFace viewer which is based on the *PictureTree* widget.

6.6.1 Difference between A Viewer and A Widget

A viewer acts as an adapter of a low level widget, provides additional feature such as sorting, filtering, and event listeners. All viewers belong to the JFace package instead of SWT. A JFace viewer consists of an SWT widget (e.g. Tree, Table, etc), plus domain objects and acts as a bridge between them. A viewer is able to sort and filter domain objects, as well as updates the widget when your domain objects change. This is realized by inheriting and re-implementing *ViewerSorter* and *ViewerFilter* classes provided by the platform and adding them to the viewer. If you use a SWT widget directly, you have to convert your objects into the strings and images expected by the widget API because SWT widgets have no knowledge of your domain.

6.6.2 Eclipse TreeViewer Architecture

The base class for all high level JFace viewers is derived from the *Viewer* class, which defines an input of data to be displayed, and selection of support capabilities. Each viewer has an associated SWT widget. This widget can be created automatically in the constructor of the *Viewer* class, or explicitly by creating it first and supplying it to the viewer in its constructor. The *Viewer* class is defined as abstract because the main control type, input and selection cannot be determined at this point. When a new input is set, the *inputChanged()* method will be called which defaults to empty.

Based on *Viewer*, the class *ContentViewer* handles the model data through *IContent-Provider* and *ILabelProvider* interfaces. *IContentProvider* mediates between the viewer's model and the viewer itself so any data change in the model will be notified to the viewer. One of the main reasons to provide a content implementation is to avoid data duplication. For example, if the data to be rendered exists in a model, a content provider can be used to retrieve them instead of creating them. *ILabelProvider* provides interfaces to get the text and image for the label of a given element in the model. The *Viewer* class defines a generic infrastructure for handling model input, updates and selections.

StructuredViewer is an abstract implementation of structure-oriented viewers, such as trees, tables, lists. It supports user defined sorting, filtering and updating. Starting from *StructuredViewer*, all subclasses of *Viewer* class begin to have a hashtable that contains a map from model elements to visible interface items. But by default this cache capability is turned off. When turned on, the hashtable can improve the performance by caching visible items; but at the same time it will increase memory requirements for huge data set. Raw data will be returned by the *getElements()* method defined in the *IStructuredContentProvider* interface. The *filter()* method will check and decide whether a visual element can pass a list of filters before rendering it.

For a *TreeViewer*, when first created, the required child nodes are queried from *ITreeContentProvider* of the model, depending on the level of nodes to be expanded. *Tree* control in *TreeViewer* will create interface elements by wrapping visual attributes such as images and colours around the model children. The interface element is called *TreeItem*. In the next step, these *TreeItem* children will be filtered and sorted if a sorter exists. *Tree* control further delegates the actual rendering to the underlying operating system.

Inside the *Tree* control, an array is used to keep track of all the expanded nodes and their children including those filtered. Collapsing a node and will not delete its nodes. In the case of a structural change, such as deletion of some nodes, the corresponding data in the array will be emptied but the array will not reduce in size. This incremental array can be one of the problem sources for large trees.

The *Tree* control is designed to closely depend on the operating system, and passes all the data to the operating system. The operating system needs the whole tree for rendering. As a result the problem dealing with a large tree will eventually occur.

6.6.3 PartialTreeViewer

We decided therefore to extend *ContentViewer* and create a viewer called *PartialTreeViewer* for *PictureTree*. Though from a high-level view, *PartialTreeViewer* is a kind of

TreeViewer, it is from *StructuredViewer* in the *Viewer* hierarchy, that the filter functionality is added into the framework.

PartialTreeViewer adds support for double click listeners, selection, and refreshing. It also supports label providers, colour providers, and expand icon providers to adjust and control the rendering of tree on the screen.

Based on *PartialTreeViewer*, we implement another viewer called *TraceViewer*. *TraceViewer* is developed with the direct association with the trace model based on CTF format. Because we need to support explorations of different parts of a trace at the same time, we use an *Exploration* class internally to store all the state information of the current trace. Each *TraceViewer* has an associated *Exploration* and a dynamic tree will be displayed in it. A *TraceViewer* also has a label provider to supply decorated label names, images for its nodes, and a color provider to determine foreground and background colors of nodes. Images and colors are determined by attributes from the trace model, such as whether a node is filtered, is a pattern, is a utility, or is highlighted.

TraceViewer will be used in an editor for trace exploration and visualization.

Chapter 7 Evaluation

This chapter discusses the experiment used to evaluate the effectiveness of the new widget and the tool.

7.1 Goal of Evaluation

As described in Chapter 3, our current research focuses on effective handling of large traces, especially those with millions of lines. Our goal is to test whether SEAT can help a software engineer easily understand and manipulate large traces. To be specific, this user evaluation will address the following three questions:

1. Identify the usefulness of the new widget and any potential usability problems that exist in the tool.
2. Obtain evidence indicating whether trace processing with the tool can reach an acceptable performance level (based on user tasks performed and time taken).
3. Identify whether the tool can help understanding.

7.2 Methodology

Our study followed general guidelines and approaches for usability testing [84]. These included designing training material, tasks and the questionnaire, recruiting test participants, conducting testing, and analyzing test results.

7.2.1 Test Participants

In order to identify whether the tool can effectively help software engineers understand a system, we decided to recruit participants and asked them to manipulate the traces from these systems using our tool. According to usability studies, at least five participants are needed to generate a reliable result [84].

We recruited 10 software developers; some were graduate students and others were software engineers from industry. We called them ‘participants’. All the participants had software engineering, graphical interface design and programming experience.

7.2.2 Test Preparation

Before the study started, we developed a quick guide for the tool that a participant can refer to during the study. In this guide, basic concepts are explained so novice participants can familiarize themselves with the tool.

Tasks can be divided into three main categories to cater to the goal described in previous section:

- Featured oriented tasks. These tasks allow a participant to explore various functionalities provides by the tool.

- Performance oriented tasks. These tasks include exploring traces using both the traditional tree widget and the newly designed widget and comparing their performance.

- Goal oriented tasks. We provide a high level goal, such as understanding what a subtree is doing, and let a participant determine answers by exploring in whatever way they see fit.

We designed 11 questions for the questionnaire. Most of them related to the tasks that the test participants had conducted. We also provided some general open-ended questions to get their subjective opinions of the tool and their overall exploration experience. The total list of the questions can be found in Table 4.

7.2.3 Study Process

Participants performed three activities: training, using the tool, and completing a questionnaire.

First the participants were asked to sign the informed consent form (See Appendix C). Then in the training phase, we provided them a copy of the instruction guide and showed them how to use the software for about 10 minutes; they were allowed to ask any questions they wished.

Participants were then asked to explore several traces using this tool and were asked to do certain tasks such as selecting different filtering algorithms to compare their effectiveness, and finding patterns inside the trace.

Each session took approximately 45 minutes to complete and was videotaped. We also took notes. If a participant spent more than four minutes on a task and could not find a solution, we considered that the task failed and went to the next one. Participants could also ask questions about the tool and the tasks during any activity.

At the end of each session, participants were asked to complete a questionnaire in which they were asked to state their opinions about several assertions using a typical Likert Scale with options: Strongly Agree, Agree, Neutral, Disagree, Strongly Disagree.
The questionnaire also included questions about the background of participants, including:

- The number of years they have spent as a software developer.
- Whether they have had a little, a moderate amount, or a lot of experience in the Eclipse platform.
- Whether they have had a little, a moderate amount, or a lot of experience working with traces or dynamic analysis.

The participants were also asked orally to give any other feedback they may have to the researcher.

7.2.4 Traces and Systems being traced

In our evaluation, we collected traces from three subject systems. One is a remote test and debug environment developed by C. Two other systems are java based, one is a open source data mining tool and the other is a test and monitor tool.

Traces of various sizes are selected for the test. The following table lists some of the traces we specifically used in the test.

No.	Original Trace File Size (KB)	Total Method Calls	Trace File Size in CTF Format (KB)	Distinct Method Calls
1	3,345	85,406	344	225
2	15,718	219,507	522	850
3	14,636	385,434	1,536	294
4	22,705	631,530	270	1,734
5	29,127	742,812	3,796	72
6	94,862	2,409,740	10,070	344

Table 2 Trace size

We also obtained other traces with total method calls greater than 1 million. But in the experiment some smaller traces are chosen instead because the tool using the *TreeViewer* provided by Eclipse platform will sometimes cause "Out of Memory" error and crash.

7.3 Analysis of Study Results

7.3.1 Time to Perform the Tasks

On average, the participants have 5.8 years of experience in software development, with little to moderate knowledge of the Eclipse IDE and dynamic analysis concepts. The test was conducted on an IBM P4 2.8G PC with 760M RAM. The version of Eclipse was 2.1.1. The result for tasks completion time is shown in the following table. During the test design, we intended the participant to finish the test within 30 minutes. Therefore each task would take about one and half minutes on average. Considering the different difficulties of the tasks, we allowed a maximum of four minutes for a task. If a participant could not finish a task or needed assistance to finish a task, we consider the task as failed. Table 3 is a list of tasks with their completion time and 95% confidence intervals for the true mean time the tasks would take.

No.	Task Category	Task	Average Completion Time(sec)	Standard Deviation	Low limit of 95% confidence	High limit of 95% confidence	No. of participants failed
1	Feature Oriented Tasks (1-14)	Open **Trace Exploration** perspective and open the trace called **trace3.ctf** in *weka* project	26	5	14	37	0
2		Identify for the nodes labeled "**weka.classifiers.IBk.buildClassifier**" using **Model** view**.** Hide the nodes labeled "**weka.classifiers.IBk.buildClassifier**" from the tree	113	16	78	148	0
3		Restore the hidden nodes in previous step	11	2	6	16	0
4		Hide the methods that belong to the class "**weka.core.Instances**"	97	25	39	155	1
5		Search for the nodes that contain methods that start with **"is"** in the tree	97	10	74	120	0
6		Select any method and open the source code that corresponds to it	42	14	10	74	0
7		Find out the number of nodes the trace contains as well as the number of distinct methods	125	13	95	155	2
8		Hide all accessing methods by executing the filtering algorithm called **"Hide Accessing Methods"**	44	13	13	74	0
9		Cancel the effect of the previous filtering algorithm (i.e. show all accessing methods)	13	2	7	19	0
10		Hide all constructors by executing the filtering algorithm called **"Hide Constructors and Destructors"**	7	2	4	11	0
11		Cancel the effect of the previous filtering algorithm (i.e. show all constructors of the trace)	8	3	2	14	0
12		Detect patterns of execution that occur in the trace	24	6	12	37	0
13		Change the system's settings so nodes that are marked as hidden will not be shown by default	118	16	83	153	0
14		Select subtree called "**weka.classifiers.Evaluation.evaluateModel**" and show all the children nodes including those marked hidden	92	29	25	159	2
15	Per-	Open the following traces in both versions of SEAT: **qnx1.ctf** in *qnx* project, **TT1-stat.ctf** in *toad* project, and **trace5.ctf** in *weka* project.	109	10	86	132	0

16	formance Oriented Tasks (15-19)	Close all the traces in both versions of SEAT. Open **trace4.ctf** in *weka* project. Search for methods starting with “**evaluate”** and jump to the source code of an occurrence of a “**evaluate…”** method using both versions of SEAT	108	17	71	145	0
17		Apply the following filtering algorithms on **trace4.ctf** using both versions of SEAT: **“Hide Accessing Methods**”,“**Hide Implementation of Polymorphic Methods**”, and “**Detect Patterns**”	76	21	29	123	0
18		In both versions of SEAT, try to hide enough nodes so only the very top levels of the tree are visible (i.e. about 30-40 nodes) and all the rest hidden	87	21	38	136	1
19		Close all the opened traces, restart Eclipse and open **trace6.ctf(2 million nodes)** in *weka* project using both versions of SEAT.	24	5	12	35	0
20	Goal Oriented Tasks (20-21)	Imagine you are trying to explain the trace to someone: Use whatever mechanisms explained to you during the training session that can help you get rid of nodes you think are the least important, resulting in a display that shows the 'essence' of the trace.	134	20	81	181	0
21		Now, explain to the investigator what the system was doing when the trace was collected. You may need to show or hide nodes as you do this - use whatever mechanisms are available in the tool.	N/A	N/A	N/A	N/A	N/A

Table 3 Tasks and their completion time.

Task 21 in Table 3 relates to the previous experience of a participant on the target system that the trace is gathered. The aim of this task is to test whether a participant can explain what the traced system is doing so it is not included in the calculation of the average completion time. For the remaining 20 tasks, their average completion time is 68 seconds. As we can see from Table 3, the standard deviations of some tasks are very high, which indicates a great difference of task completion for different participants. This is due to the different experience of participants, while some are familiar with Eclipse platform, others are not. The high limits of the confidence interval for all tasks are below 200 seconds, therefore, we can say that we are 95% confident that participants will finish tasks within about 3 minutes.

For the last task in the list, Task 21, only those participants familiar with the target system being traced are required to finish it. Five participants who do have previous experience with the target system can give a brief description on what the trace is doing. Because we cannot collect data for Task 21 from all participants, there is potential threat to the validity of test result, especially for the result of Task 21. But we think the threat is minimized because five participants performed the task and according to usability studies, five participants is the minimum number to generate a reliable result [84].

7.3.2 Usability Problems

During the experiment, some usability problems are found and they are summarized in Table 4.

We categorize the usability problems using four scales [96]:

1. *Severe*. The problem prevents the accomplishment of a task.
2. *Moderate*. The problem causes measurable delay and user frustration.
3. *Minor*. The problem has some adverse effect on usage.
4. *Suggestive*. The problem suggests an improvement of the system.

No.	Usability Problem	Severity
1	Order of nodes in Model view is not clear.	minor
2	Multiple occurrences of same method name (different subtrees) in Model view are causing confusion.	moderate
3	Search scope is not clear. It is better to add more options.	minor
4	Statistics of trace in Property view is hard to find.	severe
5	Refresh should appear in main menu instead of context menu, it's hard to find.	minor
6	After a setting changes, users prefer nodes to be hidden immediately instead of current approach, which keeps current subtree status.	suggestive
7	Scope difference of an algorithm on main menu and context menu is not obvious.	moderate
8	A legend of the meaning of colors should be put in Control Panel View.	suggestive
9	Feedback to users after an operation needs to be more obvious, besides node changes in the Model view.	minor
10	Some users have difficulties in finding where to make setting changes.	suggestive
11	Using Ctrl+Click to switch hidden children of a subtree is not obvious to first time user.	moderate

Table 4 Usability problems and their severity scale

There is one usability problem about the property view in the 'severe' category. We use the view to display properties of traces. However, some of participants took a long time to initially find the desired information and they experienced frustration. Three usability problems fall in the moderate category. They include: lack of distinction of subtrees with the same method name, lack of indication of the scope of an algorithm applied (a subtree or the whole tree), and switching hidden children using Ctrl+Click. There were also four minor problems and three suggestive problems. The results can be used to improve the future version of the tool.

7.3.3 Performance Issues

In order to test the performance of the tool, an initial prototype that uses JFace Tree-Viewer provided by Eclipse Platform was compared with SEAT. The purpose was to investigate and compare which mechanism exhibits better performance. The tasks corresponding to test performance are numbered from 15 to 19 in the task list.

However, performance data of the old version of the tool cannot be exactly gathered because the old version constantly experienced “Out of Memory” errors, although some of traces are well below one million items. Of the five tasks that are used to compare performance, four tasks therefore cannot be done (effectively they take infinite time). On the other hand, looking at the average completion time for tasks15 to 19 in Table 3, we can see that the version that uses the new widget can accomplish all the tasks in two minutes. Especially for task 19 and also task 20, a trace (No. 6 in Table 2) with more than two million nodes was used. From task completion time in Table 3, we can see that the high limit of 95% confidence for task completion time is 181 seconds.

Below, we will also raise performance issued arising from the subjective questionnaire.

7.3.4 Questionnaire Analysis

In the background survey, we found participants have 5.8 years of experience in software development on average, with a standard deviation of 3.8. Four participants had development experience less than five years, four other participants had experience between 5 and 10 and the last two had more than years experience. Most of them were not advanced users of the Eclipse platform. Five had little experience on the platform and five are intermediate level users. Lastly they had little to moderate knowledge of dynamic analysis. Four participants had little knowledge of dynamic analysis and six participants had moderate knowledge of it.

The questionnaire was formulated using multiple-choice questions and the answers of participants are divided into five possible responses on a scale from totally disagree to totally agree. We assigned these a numerical value to allow simple analysis:

1. Totally disagree
2. Disagree
3. Average
4. Agree
5. Totally agree

We decided that the desired value for an answer we considered positive should be at least 4, whereas desired value for a negative answer is at most 2. The descriptive statistics resulting from the questionnaire are presented in Table 5. Note that the upper bound of the confidence interval would necessarily be 5; data items above this are marked as *. The real upper bound would be 5.

No.	Question	Average Scale	Standard Deviation	Low Limit of Conf. Interval	High Limit of Conf. Interval
1	I feel very comfortable when using this tool.	4.2	0.25	3.64	4.76
2	Overall, I feel the tool is very easy to learn and to use.	4.0	0.33	3.25	4.75
3	I think the tool needs to be made more efficient.	3.1	0.23	2.57	3.63
4	I think the tool feedback - after performing the operations - is clear.	4.2	0.20	3.75	4.65
5	The new widget has a faster response time than the old widget.	4.8	0.13	4.50	5.10*
6	The old widget is easier to use than the new widget.	1.9	0.28	1.27	2.53
7	I am able to effectively manipulate the trace using the new widget.	4.0	0.39	3.11	4.89
8	The new widget allows me to more quickly explore a trace than the old.	4.7	0.21	4.22	5.18*
9	I prefer the old widget because it is standard.	1.8	0.25	1.24	2.36
10	I would prefer to use the new widget for trace exploration, as opposed to the old.	4.7	0.16	4.23	4.97
11	The new widget adds very little new capability.	2.8	0.36	1.99	3.6

Table 5 Result of questionnaire.

As we can see from Table 5, three questions achieve a low confidence interval limit greater than 4 showing a high level of certainty that the typical user would respond positively to them. These questions relate to the fast response time of new widget, and performance gains of the new widgets. Four questions have a low confidence interval limit above 3, which also indicate a high user preference. They mostly deal with the overall experience of participants in using the tool.

For questions regarding the capabilities added by the new widget (such as 6, 9 and 11), the results show low values. However these questions were deliberately negatively expressed regarding the new widget so participants needed to think before answering. We can invert the data and conclude that the new widget did add interesting features. Considering all the factors, we can conclude that new widget will have a scale above 3 from subjective opinions of test participants.

For question 3, we only obtained an average rating, which indicates that the users would like the tool to be more efficient. We think this is partly because some usability problems prevent some of the users from effectively accomplishing the tasks.

7.4 Summary

The above results can be summarized as follows. In terms of objective performance data (tasks completion time), the tool can meet performance requirements. Most tasks can be finished within three minutes on average. For a large trace with more than two million nodes, the tool can handle it in appropriate time.

For subjective data, we compute confidence interval at 95%. For most questions, we obtain responses that indicate the participants strongly agree with positive statements (or strongly disagree with negative statements) about the tool. The result is also supplemented by the fact that the old widget would be impractical for handling large traces because of out of memory errors and its slow response.

The overall results indicates that the performance issue is well addressed by the results of this thesis, but there are still some relatively small usability improvements to consider that could make the tool more usable.

Chapter 8 Conclusion and Future Work

In the previous chapters, we have presented our solution in dealing with large traces. This chapter summarizes our research and findings. Future work for improvement is also presented.

8.1 Review of the Research

Our research is motivated by difficulties in handling large traces when performing software dynamic analysis. Currently problems exist both in modeling large traces and visualizing them.

To effectively support exploration and visualization of large traces, a trace format called Compact Trace Format was proposed in our research lab. Our first step in addressing largeness is to represents traces using the CTF model; this involves sharing common sub trees and converting the trace to a directed acyclic graph.

Next, we developed a dynamic loading algorithm which loads trace data as the user explores the trace. The algorithm will generate a new dynamic tree called *PartialTree* in each step as the user traverses the trace. The *PartialTree* reflects the current focus of the user and only include nodes for current window.

We also built a new tree widget on the Eclipse platform to visualize the dynamic tree. A trace exploration and visualization tool, called SEAT, was developed to verify our research findings. The tool is developed according to guideline requirements we believe to be imperative for any reverse engineering tool. We obtained positive results about the tool through a basic user evaluation.

8.2 Functionalities of SEAT That Address the Requirements

SEAT is designed with consideration of the tool requirements described in Section 1 of Chapter 4. SEAT supports user-oriented exploration. During exploration, a user can explicitly hide and show nodes, identify utility methods and expand and collapse subtrees. We will see how SEAT addresses the tool requirements in the following paragraphs.

Requirement 1: *The tool needs to support high level abstraction, intermediate level abstraction and support source code navigation.*

Currently SEAT does not have a view that represents high-level abstraction of a trace, such as a use case diagram or a sequence diagram [97]. But these diagrams are can be easily obtained through reverse engineering the source code and are often provided by the IDE if source code that generates traces is available. UML plugins from other providers can fulfill this requirement [98]. A simplified trace with less-important nodes filtered by different algorithms after explorations and patterns have been identified can be viewed as the intermediate level of abstraction. SEAT can automatically parse source code if it is available and supports source code navigation. That is, the user can directly jump to a method definition in the Eclipse Java editor from a trace editor window.

Requirement 2: *The tool needs to support broad search capabilities, including basic search, customizable search, wildcard search, regular expression search and with search history being traced.*

SEAT supports the Eclipse search framework and automatically keeps a search history. Instead of searching the huge graph space, SEAT accelerates search using the internally saved distinct node set for CTF trace model and the set is very small. After a node is located, all the occurrences of that node in the graph are identified using the inverted tree property of the trace DAG as described in Section 2.9.

Requirement 3*: The tool needs to support a global view of the trace being explored and various views of data slices from different perspectives.*

As the focus of our research, SEAT currently only supports a view of a portion of the trace tree using our specifically designed widget. A global view of the trace is not supported in this prototype yet, except to the extent that the user can use different algorithms that filter away almost all the methods of the trace, except the essence. One possible approach is to use an "Information Mural"-like view [53] and map the trace to it.

Requirement 4*: The tool needs to support filtering of events, highlighting events that have specific attributes using colours or shapes.*

SEAT supports quick filtering of traces through filtering algorithms. Different colouring and labelling schemes are supported to provide visual rendering of different nodes.

Requirement 5*: The tool needs to support the user's orientation, such as the position in the global view; this concept is also known as landmarks.*

Same as the disadvantage described in Requirement 3, SEAT does not have a global view for a trace and it only has a bookmark view used for this purpose. At any time during the exploration, interesting points can be bookmarked explicitly and these points can be returned to easily. The bookmark view may also be extended to allow automatically adding points of interest.

Requirement 6: *The tool needs to minimize view switching when a user is carrying out a task.*

Currently, as an Eclipse plugin, SEAT takes the advantage of Eclipse's editor-centered strategy. An editor is the main working area for trace exploration and navigation. All the user interactions are initiated and visualized in the editor. The trace editor also supports switching to source code when desired. There are different supplementary views that help exploration, but they can be turned off. Because views in the Eclipse platform are arranged in a way that they will never overlap and hide editor area, a user can always concentrate on the trace and source code editor for an exploration task.

***Requirement** 7: The tool needs to be extensible to support different sources and destinations.*

SEAT defines the *ITraceSource* interface to support different trace sources. By default, SEAT currently supports input from a stream, such as file input. The stream input interface is available to facilitate development of a variety of input forms. Data from stream can be in pre-calculated CTF format, or raw trace format. As seen in the Class diagram for trace I/O (Figure 16), the actually trace construction is delegated to a trace loader.

***Requirement 8**: The tool needs to support workspace and session persistence so a user can continue previous work at a later time.*

SEAT has a *TraceSession* class devoted exclusively for trace persistence. States of an exploration can be easily gathered and recreated when the trace is loaded again. For example, filtering algorithms that are effective on the current trace can be recorded and reapplied when trace is loaded. However, a concrete persistence implementation is not realized yet.

***Requirement 9**: The tool needs to be able to handle traces with millions of lines of data easily and effectively, including loading and saving.*

Our main research has been motivated and focused on the handling of large traces. We address the problem by representing a trace using the CTF model. Evaluation of the performance of SEAT is discussed in Chapter 7 – Evaluation.

***Requirement 10**: The tool needs to adopt user centered design and can be integrated easily with an IDE and other tools to increase its adoption.*

We implement SEAT in the Eclipse open source platform so it can be easily integrated with other tools. SEAT itself defined an extension point for filtering algorithms as well so other developers can contribute their algorithms easily. Also we have called in an Eclipse expert to comment on and suggest improvements for the interface of our tool.

8.3 Contributions in a nutshell

To conquer the size problem encountered when exploring large traces, we developed a novel data loading algorithm and a tree browsing widget that can both be used in any scenario where a user needs to work with a massive tree. We developed these in the context of an Eclipse based tool and verified that they operate with an appropriate response time. The algorithm and widget can be used in user interface design when there is a large tree-structured data set that needs to be processed.

8.4 Future Work

Our research has been focusing on single thread traces. This limits its applicability in multi-threaded, distributed environments. Future studies could investigate how to extend the current trace model and adapt it to contain multi-thread support. For example, a future system could include more signatures, such as machine, process, and thread names.

The new widget and the tool, SEAT, based on our research need further experimental evaluation. The effectiveness of novice features of the widget needs to be further tested in an industry environment where software engineers are exploring real traces.

Another enhancement to both current and future versions of SEAT would be to integrate with the Hyades project. A direct benefit would be seamlessly integration with other trace tools.

Responding to model changes, visualization also needs to address how to display multiple threads at same time, and how to compare multiple traces using dynamic data–loading techniques.

References

[1] Hamou-Lhadj, A.; Lethbridge, T.; 2003, *The Compact Trace Format*, Proceedings of the 1st International Workshop on Meta-models and Schemas for Reverse Engineering (ATEM), Co-located with the 10th Working Conference on Reverse Engineering (WCRE), Victoria, BC, Canada, November 2003.

[2] Freitag, F.; Caubet, J.; Labarta. J.; 2002, *A Trace-Scaling Agent for Parallel Application Tracing*, IEEE International Conference on Tools with Artificial Intelligence (ICTAI), pp. 494-499, Washington D.C., November 2002.

[3] Munzner, T.; Guimbretiere, F.; Tasiran, S.; Zhang, L.; and Zhou, Y.; 2003, *TreeJuxtaposer: Scalable Tree Comparison using Focus + Context with Guaranteed Visibility*, SIGGRAPH 2003, pp. 435-462.

[4] Fjelstad, R.K.; Hamlen, W.T.; 1983*, Application Program Maintenance Study: Report to Our Respondents*, Tutorial on software maintenance, April (1983).

[5] Von Mayrhauser, A.; Vans, A.M.; 1995, *Program Comprehension During Software Maintenance and Evolution*, Computer, Volume: 28, Issue: 8, pp. 44-55, Aug. 1995.

[6] Ahrens, J.; Brislawn, K.; Martin, K.; Geveci, B.; Law, C.C.; Papka, M.; 2001, *Large-Scale Data Visualization Using Parallel Data Streaming*, Computer Graphics and Applications, IEEE, Volume: 21, Issue: 4, pp. 34-41, July-Aug. 2001.

[7] Novotny, M.; 2004, *Visually Effective Information Visualization of Large Data*, Technical Report, Research Center for VrVis, 2004.

[8] Miller, L.; Honavar, V.; Barta, T.A.; 1997, *Warehousing Structured and Unstructured Data for Data Mining*, Proceedings of the American Society for Information Science Annual Meeting (ASIS 97). Washington, D.C.

[9] Ball, T. A.; Eick, S. G. 1996, *Software Visualization in the Large*. IEEE Computer, 29(4): pp. 33-43, April 1996.

[10] Grant, C.; 1999, *Software Visualization in Prolog*, Technical Report UCAM-CL-TR-511, Computer Science Department, University of Cambridge, December 1999.

[11] Chikofsky, I.; Cross, J.H.; 1990, *Reverse Engineering and Design Recovery: A Taxonomy*, IEEE Software, vol.7, no.1, Jan. 1990.

[12] Favre, J.; 1997, *Understanding-In-The-Large*, Proceedings of the 5th International Workshop on Program Comprehension, pp. 29-38, May 28-30, 1997, Dearborn, MI.

[13] Müller, H.A.; Jahnke, J.H.; Smith, D.B.; Storey, M; Tilley, S.R.; Wong. K.; 2000, *Reverse Engineering: A Roadmap*, May 2000, Proceedings of the Conference on The Future of Software Engineering, pp. 47–60.

[14] *http://www.computerdictionary.info*.

[15] Ernst, M.D.; 2003, *Static and Dynamic Analysis: Synergy and Duality*, WODA 2003: ICSE Workshop on Dynamic Analysis, pp. 24-27, (Portland, OR), May 9, 2003.

[16] Systa, T.; 2000, *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*, PhD Thesis, University of Tampere, Dept. of Computer and Information Sciences, Report A-2000-4, 2000.

[17] Miller, E.; 1984, *Quality Managment Technology: Practical Applications*, Software Validation 1984, pp. 255-266.

[18] Wilde, N.; Scully, M.; 1995, *Software Reconnaissance: Mapping Program Features to Code*, Journal of Software Maintenance: Research and Practice, Vol. 7, No. 1, January 1995.

[19] Marburger, A.; Westfechtel, B.; 2003, *Tools for Understanding the Behavior of Telecommunication System*, 25th International Conference on Software Engineering, May 2003.

[20] Wong, K.; 1994, *Software Understanding Through Integrated Structural and Run-time Analysis*, Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative Research, Toronto, Ontario, Canada.

[21] Lakhotia, A.; 1993, *Understanding Someone Else's Code: Analysis of Ex*perience, Journal of systems and software, vol. 26, 1993.

[22] Canfora, G.; Mancini, L.; Tortorella, M.; 1996, *A Workbench for Program Comprehension During Software Maintenance*, Proceedings of Program Comprehension, pp. 30-39, Fourth Workshop, 29-31 March 1996.

[23] Müller, H.A.; Tilley, S.R.; Wong K.; 1993, *Understanding Software Systems Using Reverse Engineering Technology: Perspectives from the Rigi Project*, Proceedings of CASCON '93, pp. 217-226, Toronto, Ontario, October 25-28, 1993.

[24] Soloway, E.; Adelson, B.; Ehrlich, K.; 1988, *Knowledge and Processes in the Comprehension of Computer Programs*, The Natare of Expertise**,** pp. 129-152, Lawrence Erlbaum Associates.

[25] Brooks, R.; 1983, *Towards a Theory of the Comprehension of Computer Programs*, International Journal of Man-Machine Studies, Vol. 18, pp. 543-554, 1983.

[26] Pennington, N.; 1987, *Comprehension Strategies in Programming*, Empirical studies of programmers: Second workshop Human/computer interaction, Vol. 7. Norwood, NJ, Ablex Publishing, 1987

[27] Von Mayrhauser, A.; Marie Vans, A.; 1997, *Program Understanding Behavior During Debugging of Large Scale Software*, Papers presented at the seventh workshop on Empirical studies of programmers, October 1997.

[28] Price, B.A.; Baecker, R.M.; Small, I.S.; 1993, *A Principled Taxonomy of Software Visualization*, Journal of Visual Languages and Computing 4(3), pp. 211-266, 1993.

[29] Hendrix,T.D.; Cross II,J.H; Maghsoodloo S.; McKinney, M.L.; 2000, *Do Visualizations Improve Program Comprehensibility? Experiments With Control Structure Diagrams for Java*, Proceedings of SIGCSE 2000, ACM

[30] Eick, S.; Steffen, J.L.; Summer, E.E.; 1992, *Seesoft - A Tool For Visualizing Line Oriented Software Statistics*, IEEE TSE, vol. 18, no. 11, pp. 957-968, November 1992.

[31] Marcus, A.; Feng, L.; Maletic, J.I.; 2003, *3D Representations for Software Visualization*, Proceedings of the ACM Symposium on Software Visualization (SoftVis 2003), pp. 27-36, San Diego, CA.

[32] Stuart K.; Card, S.K.; MacKinlay, J.D.; Shneiderman, B.; 1999, *Readings in Information Visualization-Using Vision to Think*, Series in Interactive Technologies, Morgan Kaufmann, Morgan Kaufmann Publishers, 1999. ISBN 1-55860-533-9.

[33] Pirolli, P.; 2001, *Visual Information Foraging in a Focus + Context Visualization*, Proceedings of the SIGCHI, March 2001.

[34] Beaudoin, L.; Parent, M.A.; Vroomen, L.C.; 1996, *Cheops: A Compact Explorer for Complex Hierarchies*, Proceedings of Visualization'96, San Francisco.

[35] Baldonado, M.; Woodruf, A; Kuchinsky, A.; 2000, *Guidelines for Using Multiple Views in Information Visualization*, Proceedings of the Working Conference on Advanced Visual Interfaces, pp. 110-119, Palermo, Italy, ACM Press.

[36] Roberts, J.C.; 1998, *On Encouraging Multiple Views for Visualization*, IEEE Information Visualization, pp. 8-14, July 1998.

[37] Wu, J.; Storey, M.D.; 2000, *A Multi-Perspective Software Visualization Environment*, Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research, Mississauga, Ontario, Canada

[38] Johnson, B.; Shneiderman, B.; 1991, *Tree-maps: A Space-filling Approach to the Visualization of Hierarchical Information Structures*, Proceedings of Visualization '91, pp.284-291, IEEE Conference, 22-25 Oct. 1991.

[39] Robertson, G.; 1991, *Cone Trees: Animated 3D Visualizations of Hierarchical Information*, Proceedings of the SIGCHI, March 1991.

[40] Carriere, J.; Kazman, R.; 1995*, Research Report: Interacting with Huge Hierarchies: Beyond Cone Trees*, Proceedings on Information Visualization, October 30 - 31, 1995, Atlanta, Georgia, USA.

[41] Cockburn, A.; McKenzie, B.; 2001, *3D or not 3D?: Evaluating the Effect of the Third Dimension in a Document Management System*, Proceedings of the SIGCHI conference on Human factors in computing systems, March 2001.

[42] Nguyen, T.D.; Ho, T.B.; Shimodaira, H.; 2000, *A Visualization Tool for Interactive Learning of Large Decision Trees*, Proceedings of the 12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'00), 2000.

[43] Storey, M.; Best, C.; Michaud, J.; Rayside, D.; Litoiu, M.;Musen, M.; 2002, *SHriMP Views: an Interactive Environment for Information Visualization and Navigation*, CHI '02 extended abstracts on Human factors in computing systems, April 2002.

[44] Tudoreanu, M.E.; 2003, *Algorithm Animation Evaluation: Designing Effective Program Visualization Tools for Reducing User's Cognitive Effort*, Proceedings of the 2003 ACM symposium on Software visualization, June 2003.

[45] Wang, L.; 2002, *Animated Exploring of Huge Software Systems*, Master thesis, http://www.site.uottawa.ca/~tcl/gradtheses/lwang/.

[46] *http://www.cepba.upc.es/paraver/.*

[47] Malony, A.D.; Larson, J.L.; Reed, D.A.; 1990, *Tracing Application Program Execution on the Cray X-MP and Cray 2*, Proceedings of the 1990 conference on Supercomputing, pp. 60-73, 1990.

[48] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W.G.; 2001, *An Overview of AspectJ*, Proceedings of ECOOP 2001.

[49] Malony, A.; Hammerslag, D. ; Jabalonski. D.; 1991, *Traceview: A Trace Visualization Tool*, IEEE Software, pp 19-28, September 1991.

[50] Moe, J.; Carr, D.; 2001, *Understanding Distributed Systems via Execution Trace Data*, Proceedings of the Ninth International Workshop on Program Comprehension, 60-67, Toronto, Canada. 12-13 May 2001.

[51] Orso,A.; Jones,J.; Harrold, M.J.; 2003, *Visualization for Program Understanding: Visualization of Program-execution data for Deployed Software*, Proceedings of the 2003 ACM symposium on Software visualization, June 2003.

[52] Reiss, S.P.; Renieris, M.; 2001, *Encoding Program Executions***,** 23rd International Conference on Software Engineering (ICSE'01), May 12-19, 2001.

[53] Jerding, D.J.; Stansko, J.T.; Ball, T.; 1997, *Visualizing Interactions in Program Executions*, Proceedings of the 1997 International Conference of Software Engineering (ICSE '97), pp. 360-370, 1997.

[54] Reniers, M.; Reiss, S.P.; 1999, *ALMOST: Exploring Program Trace,* Workshop on New Paradigms in Information Visualization and Manipulation, pp. 70-77, 1999.

[55] *http://www.eclipse.org/hyades*.

[56] Marshall, S.; Jackson, K.; McGavin, M.; Duignan, M.; Biddle, R.; Tempero, E.; 2001, *Visualising Reusable Software Over The Web*, Information Visualisation 2001: Australian Symposium on Information Visualisation, December 2001

[57] Gschwind,T.; Oberleitner, J.; Pinzger, M.; 2003, *Using Run-Time Data for Program Comprehension*, Proceedings of the 11th IEEE International Workshop on Program Comprehension, pp. 245.

[58] Pacione, M. J.; Roper, M.; Wood, M.; 2003, *A Comparative Evaluation of Dynamic Visualisation Tools*, 10th Working Conference on Reverse Engineering**,** November 13-17, 2003**,** Victoria, B.C., Canada

[59] Hamou-Lhadj, A.; Lethbridge, T; 2004, *A Survey of Trace Exploration Tools and Techniques*, the 14th Annual IBM Centers for Advanced Studies Conferences (CASCON), Toronto, Canada, October 2004

[60] Dauphin, P.; Hofmann, R.; Lemmen, F.; Mohr, B.; 1996, *Simple, a Universal Tool Box for Event Trace Analysis*, 2nd International Computer Performance and Dependability Symposium (IPDS '96), September 1996

[61] Aydt, R.A.; 1994, *SDDF: The Pablo Self-describing Data Format.* Tech. Rep., Department of Computer Science, University of Illinois, April 1994.

[62] *http://www.gupro.de/GXL/.*

[63] Walker, R.J.; Murphy, G.C.; Steinbok,J.; Robillard, M.P.; 2000, *Efficient Mapping of Software System Traces to Architectural Views*, Proceedings of CASCON 2000, pp. 31-40, 2000.

[64] Das, S.; Johnson, E.E.; 1995*, Accuracy of Filtered Traces*, Proceedings of 1995 IEEE International Phoenix Conference on Computers and Communications, IEEE, April 1995.

[65] Fu, J.W.C.; Patel, J.H.; 1994, *Trace–driven Simulation Using Sampled Traces*, Proceedings of 27th Ann. Hawaii Int'l. Conf. on System Sciences, pp. 211–220, 1994.

[66] Freitag, F.; Caubet, J.; Labarta, J.; 2002, *A Trace-Scaling Agent for Parallel Application Tracing*, IEEE International Conference on Tools with Artificial Intelligence (ICTAI), pp. 494-499, November 2002.

[67] Eeckhout, L.; De Bosschere, K.; 2004, *Efficient Simulation of Trace Samples on Parallel Machines*, Parallel Computing, Elsevier, Vol. 30, 2004, pp. 317- 335.

[68] Johnson, E.E.; Jiheng Ha; Baqar Zaidi, M.; 2001, *Lossless Trace Compression*, Computers, IEEE Transactions on, Volume: 50, Issue: 2, pp. 158-173, Feb. 2001.

[69] Hamou-Lhadj, A.; Lethbridge, T.; 2002, *Compression Techniques to Simplify the Analysis of Large Execution Traces*, Proceedings of the 10th IEEE International Workshop on Program Comprehension (IWPC), pp. 159-168, Paris, France, June 2002.

[70] Reiss, S. P.; Renieris, M.; 2000, *Generating Java Trace Data*, Proceedings of the ACM 2000 conference on Java Grande, pp. 71 – 77, 2000.

[71] Larus, J.R.; 1999, *Whole Program Paths*, Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation, pp. 259-269,1999.

[72] Hamou-Lhadj, A.; Lethbridge, T.; 2003, *Techniques for Reducing the Complexity of Object-Oriented Execution Traces*, Proceedings of Vissoft 2003, 2nd IEEE International Workshop on Visualizing Software for Understanding and Analysis, Co-located with the 19th International Conference on Software Maintenance (ICSM), pp. 35-40, Amsterdam, The Netherlands, October 2003.

[73] Kannan,S.; Warnow, T.; Yooseph, S.; 1998, *Computing the Local Consensus of Trees*, SIAM Journal on Computing, Volume 27, Number 6, pp. 1695-1724.

[74] Anquetil, N.; Lethbridge, T.C.; 1998, *Assessing the Relevance of Identifier Names in a Legacy Software System*, Cascon 1998, pp 213-222.

[75] Jackson, K.; Biddle, R; Tempero, E.; 2000, *Understanding Frameworks through Visualisation*, 37th International Conference on Technology of Object-Oriented Languages and Systems, November 2000.

[76] Ferenc, R.; Magyar, F.; Beszedes, A.; Kiss, A.; Tarkiainen, M.; 2001, *Columbus - Tool for Reverse Engineering Large Object Oriented Software Systems*, Proceedings of the Seventh Symposium on Programming Languages and Software Tools, pp. 16-27, June 2001.
[77] Gueheneuc, Y.; 2004, A Reverse Engineering Tool for Precise Class Diagrams, CASCON, 2004.

[78] Wu, C.E.; Bolmarcich, A.; Snir, M.; Wootton, D.; Parpia, F.; Chen, A.; Lusk, E.; Gropp, W.; 2000, *From Trace Generation to Visualization: A Performance Framework for Distributed Parallel Systems*, Supercomputing 2000.

[79] Lintern R.; Michaud,J.; Storey, M.A.; Wu,X.; 2003, *Plugging-in Visualization: Experiences Integrating a Visualization Tool with Eclipse*, ACM Symposium on Software Visualization, (Softvis'2003), pp. 47-56, and p. 209, San Diego, June 2003.

[80] Redmiles, D.; 1993, *Observations On Using Empirical Studies in Developing a Knowledge-Based Software Engineering Tool*, Proceedings of the 8th Annual Knowledge-Based Software Engineering (KBSE-93) Conference, pp. 170-177, IEEE Computer Society Press, Los Alamitos, CA, September 1993.

[81] Lethbridge, T.; 2004, *Value Assessment by Potential Tool Adopters: Towards a Model that Considers Costs and Risks of Adoption*, Proceedings of ACSE 2004, Edinburgh, Scotland, UK.

[82] Gershon, N.; Eick, S.G.; Card, S.; 1998, *Information Visualization*, Interactions, Volume 5 Issue 2 pp. 9–15, March 1998

[83] Hamou-Lhadj, A.; Lethbridge, T.; Fu, L.; 2004, *Challenges and Requirements for an Effective Trace Exploration Tool*, Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC), pp. 70-78, Bari, Italy, June 2004.

[84] Nielsen, J.; 1994, *Usability Engineering*, published by Morgan Kaufmann, San Francisco, 1994, ISBN 0-12-518406-9.

[85] Lethbridge, T.C.; Anquetil, N.; 1997, *Architecture of A Source Code Exploration Tool: A Software Engineering Case Study*, University of Ottawa. Computer Science Technical Report TR97-07.

[86] Pirolli, P.; Card, S.K.; Van Der Wege, M.; 2000, *The Effect of Information Scent on Searching Information Visualizations of Large Tree Structures*, Proceedings of Advanced Visual Interfaces Conference (AVI 2000), pp.161-172, 2000.

[87] Janzen, D.; De Volder, K.; 2003, *Navigating and Querying Code Without Getting Lost*, Proceedings of ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications, 2003.

[88] Liu, H.; 2001, *Intelligent Search Techniques for Large Software Systems*, M.Sc. Computer Science, 2001, http://www.site.uottawa.ca/~tcl/gradtheses/hliu/.

[89] Vitter, J.S.; 2001, *External Memory Algorithms and Data Structures: Dealing with Massive Data*, ACM Computing Surveys, 33 (2): pp. 209-271, June 2001.

[90] Fua, Y.; Ward, M.O.; Rundensteiner, E. A.; 1999, *Navigating Hierarchies with Structure-based Brushes*, Proceedings of the IEEE Symposium on Information Visualization, pages 58-64, San Francisco, California, October 1999.

[91] Dachselt, R.; Ebert, J.; 2001, *Collapsible Cylindrical Trees: A Fast Hierarchical Navigation Technique*, Information Visualization, pp. 79-86, 2001.

[92] Moen, S.; 1990*, Drawing Dynamic Trees*, IEEE Software 7(4): pp. 21-28, July 1990.

[93] Card, S. K.; Nation, D. 2002, *Degree-of-Interest Trees: A Component of an Attention-reactive User Interface*, Advanced Visual Interfaces '02, Trento, Italy, 2002.

[94] *http://www.delphi-gems.com/VirtualTreeview/.*

[95] Grosjean, J.; Plaisant, C.; Bederson, B.; 2002, *SpaceTree: Supporting Exploration in Large Node Link Tree, Design Evolution and Empirical Evaluation*, Procedings of IEEE Symposium on Information Visualization, pp. 57 -64, Boston, October 2002

[96] Ebling, M.R.; John, B.E.; 2000, *On the Contributions of Different Empirical Data in Usability Testing*, ACM 1-58113-219-0/00/0008

[97] *http://www.uml.org/.*

[98] *http://www.omondo.com/.*

Appendix A: SEAT Evaluation Instructions

Introduction

SEAT (Software Exploration and Analysis Tool) is tool that aims to help software maintainers understand a large software system by analyzing its execution traces.

SEAT user interface is based on Eclipse platform and consists mainly of a multiple-page trace editor and a set of auxiliary views. The trace is displayed in the trace editor in the form of a tree structure. The auxiliary views are used to display different kinds of information such as pattern, statistical data and so on. Auxiliary views can be opened automatically in trace perspective or by selecting **Window > Show View** menu.

SEAT implements a variety of algorithms that can be used to filter the displayed trace. Some of these algorithms require the setting of some parameters. This can be done by selecting **Window > Preferences > Trace Preference**.

Purpose

The purpose of the testing is to evaluate the effectiveness of the tool. The result will be used to improve SEAT and guide future research.

Concepts

Primary concepts requiring some explanation for this user study include:

- **Execution Trace**: The result of executing a software system. There are different types of traces. SEAT supports traces of method calls.
- **Filtering Algorithm**: An algorithm that filters out some components from the trace depending on a set of criteria. For example, a user can choose to filter out utility components.
- **Compact Trace Format (CTF):** Represents the format used to store trace data.

- **Trace Pattern**: Similar sequences of events that occur in a non-contiguous way in the trace.
- **View**: An Eclipse window.

Operations

Typical software operations for SEAT include:

- *Open Trace Perspective*
  - Select **Window > Open Perspective > Other**
  - In “Select Perspective” dialog, select **Trace Exploration**
- *Open a view*
  - Select **Window > Show View > Other**
  - In “Show View” dialog, click **Trace Exploration** and select desired view
- *Open a trace*
  - In “Navigator” view, double click on a **.ctf** trace file
- *Apply/Cancel an algorithm on the trace*
  - When a ctf file is opened in a trace editor, the **Algorithm** toolbar and the **Algorithms** menu list available algorithms.
  - To apply an algorithm (an operation that filters nodes):
    - Select **Algorithms > “An Algorithm”**
    - Click on an algorithm icon in the **Algorithm** toolbar
  - To cancel an algorithm, simply deselect that algorithm in either menu or toolbar
- *Apply an algorithm on a subtree of trace*
  - Right-click on a subtree node
  - From the node’s context menu, select **“**An Algorithm”
- *Hide a group of nodes*
  - Select **Algorithms > Hide Nodes** or
  - Select **Hide Nodes** from **Algorithm** toolbar
  - In the “Hide Matched Nodes” dialog, enter the filter condition and click OK
- *Hide selected nodes*

    - Select node(s)
    - From the nodes' context menu select **Hide**
- *Hide/Show nodes through Model view*
    - Select model node(s)
    - Check/uncheck **Hidden** field of selected nodes
- *Identify patterns*
    - Select **Algorithms > Detect Patterns** or
    - Click on Detect Patterns icon in the **Algorithm** toolbar
- *Mark utility*
    - Select node(s)
    - From the nodes' context menu, select **Mark as Utility Method**
- *Remove utility*
    - In "Utility" view, select nodes to be removed
    - Click **Remove Selected** icon in local toolbar of the view
- *Add bookmark*
    - Select nodes to be book marked
    - From the nodes' context menu, select **Add Bookmark**
- *Search*
    - Select **Search** > **Search** menu
    - In "Search" dialog, select **Trace Search** page
    - Enter *condition* and click **Search** button
- *Open a new exploration*
    - Right click in editor and select **New Exploration on Same Trace**
- *Open source code*
    - Double-click on the **method name** of a node if source code is available
- *Show trace and node properties*
    - Open "Property" view and relevant data will be displayed
- *Refresh editor*
    - Right click in editor and select **Refresh**
- *Change settings*
    - Select **Window > Preferences > Trace Preference**

    - Refresh editor to reflect changes
- *Switch between fully/partially expand state of a subtree*:
    - Select a subtree node, and **Ctrl+Click**

Appendix B: Schema of "seat.algorithm" Extension Point

```
<?xml version='1.0' encoding='UTF-8'?>
<!-- Schema file written by PDE -->
<schema targetNamespace="seat">
<annotation>
    <appInfo>
      <meta.schema plugin="seat" id="algorithms" name="Trace Compression Algo-
rithms"/>
    </appInfo>
    <documentation>
      [Enter description of this extension point.]
    </documentation>
  </annotation>

  <element name="extension">
    <complexType>
      <sequence>
        <element ref="algorithm" minOccurs="1" maxOccurs="unbounded"/>
      </sequence>
      <attribute name="point" type="string" use="required">
        <annotation>
          <documentation>


          </documentation>
        </annotation>
      </attribute>
```

```
        <attribute name="id" type="string">
          <annotation>
            <documentation>

            </documentation>
          </annotation>
        </attribute>
        <attribute name="name" type="string">
          <annotation>
            <documentation>

            </documentation>
          </annotation>
        </attribute>
      </complexType>
    </element>

    <element name="algorithm">
      <complexType>
        <sequence>
          <element ref="parameter" minOccurs="1" maxOccurs="unbounded"/>
          <element ref="description"/>
        </sequence>
        <attribute name="name" type="string" use="required">
          <annotation>
            <documentation>

            </documentation>
          </annotation>
        </attribute>
        <attribute name="id" type="string" use="required">
          <annotation>
            <documentation>

            </documentation>
          </annotation>
        </attribute>
        <attribute name="index" type="string" use="required">
          <annotation>
            <documentation>

            </documentation>
          </annotation>
        </attribute>
        <attribute name="class" type="string" use="required">
          <annotation>
            <documentation>

            </documentation>
          </annotation>
        </attribute>
        <attribute name="category" type="string">
          <annotation>
            <documentation>

            </documentation>
          </annotation>
        </attribute>
        <attribute name="preprocess" type="string">
          <annotation>
            <documentation>
```

```
            </documentation>
          </annotation>
        </attribute>
        <attribute name="postprocess" type="string">
          <annotation>
            <documentation>

            </documentation>
          </annotation>
        </attribute>
        <attribute name="icon" type="string">
          <annotation>
            <documentation>

            </documentation>
          </annotation>
        </attribute>
        <attribute name="targetView" type="string">
          <annotation>
            <documentation>

            </documentation>
          </annotation>
        </attribute>
      </complexType>
    </element>

    <element name="parameter">
      <complexType>
        <attribute name="name" type="string" use="required">
          <annotation>
            <documentation>

            </documentation>
          </annotation>
        </attribute>
        <attribute name="id" type="string" use="required">
          <annotation>
            <documentation>

            </documentation>
          </annotation>
        </attribute>
        <attribute name="type" type="string" use="required">
          <annotation>
            <documentation>

            </documentation>
          </annotation>
        </attribute>
        <attribute name="default" type="string" use="required">
          <annotation>
            <documentation>

            </documentation>
          </annotation>
        </attribute>
      </complexType>
    </element>

    <element name="description" type="string">
    </element>
```

```
<annotation>
  <appInfo>
    <meta.section type="since"/>
  </appInfo>
  <documentation>
    [Enter the first release in which this extension point appears.]
  </documentation>
</annotation>

<annotation>
  <appInfo>
    <meta.section type="examples"/>
  </appInfo>
  <documentation>
    [Enter extension point usage example here.]
  </documentation>
</annotation>

<annotation>
  <appInfo>
    <meta.section type="apiInfo"/>
  </appInfo>
  <documentation>
    [Enter API information here.]
  </documentation>
</annotation>

<annotation>
  <appInfo>
    <meta.section type="implementation"/>
```

```
      </appInfo>
      <documentation>
         [Enter information about supplied implementation of this extension point.]
      </documentation>
   </annotation>

   <annotation>
      <appInfo>
         <meta.section type="copyright"/>
      </appInfo>
      <documentation>

      </documentation>
   </annotation>

</schema>
```

Appendix C: Informed Consent Form

Title of the study: Visualization and Navigation of Large Traces.

Name of Researcher: **Lianjiang Fu**	Phone: **(613) 562-5800 x6428**,
Department/School: **SITE**	
Faculty: **Engineering**	E-mail: lfu@site.uottawa.ca
Institute: University of Ottawa	
Name of Researcher: **Abdelwahab Hamou-Lhadj**	Phone: **(613) 562-5800 x6688**,
Department/School: **SITE**	
Faculty: **Engineering**	E-mail: ahamou@site.uottawa.ca
Institute: University of Ottawa	
Supervisor: **Dr. Timothy Lethbridge**	Phone: **(613) 562-5800 x6685**,
Department/School: **SITE**	Fax.: **(613) 822-5473**
Faculty: **Engineering**	E-mail: tcl@site.uottawa.ca
Institute: University of Ottawa	Position: Associate Professor

Invitation to Participate: I am invited to participate in the research study conducted by Lianjiang Fu, master's student, who is supervised by Dr. Timothy Lethbridge, of the Faculty of Engineering, and assisted by Abdelwahab Hamou-Lhadj, PhD student. The project is funded by NCIT and QNX Software Systems.

Purpose of Study: I understand that the purpose of the study is to evaluate the usability of, and identify problems in, a software system entitled Software Exploration and Analysis Tool (SEAT).

Participation: The study is conducted in English and my participation will consist essentially of the following: **I will be trained** for about 10 minutes to use the software. Then **I will be asked to use a software system for about 30 minutes**, performing specific tasks with the software, as requested by the students. During this time, **I will be asked to talk out loud to the best of my ability, describing what I am thinking and doing**. At the end of the session **I will be asked some questions** about my experiences and background for about 5 minutes; some of questions will be on paper. **The session will be videotaped.** The session will be conducted individually and not in a group format.

Risks: I understand that **this activity may cause me mild frustration and stress if the software proves difficult to use or does not perform as expected**. I have been assured by the researchers that every effort will be made to minimize these occurrences by stopping a task if it becomes too difficult. I understand that it is **the software and not me that is being evaluated**, and that when I have difficulties, it is the fault of the software, not me.

Voluntary Participation: I understand that participation is strictly voluntary. I am free to withdraw from the study at any time. I may also refuse to perform any task or answer any question. I may also request that the video camera be turned off at any time. If I choose to withdraw, all data gathered until the time of withdrawal will be completely erased.

Confidentiality: I have received assurance from the researchers that the information I will share will remain strictly confidential. I understand that the contents will be used only for identifying usability problems of SEAT and that my confidentiality will be protected. The videotapes will only be viewed by the researchers listed above and Dr. Lethbridge. The videotapes will be erased upon completion of the researcher's thesis, and all other data will be deleted after five years**.**

Anonymity: My anonymity will be protected at all times. The researchers assure me that I will not be identified in any publications. If any quotes are used, neither my name nor any identifying contextual information will be mentioned.

If I am a QNX employee, I understand that my manager has given permission for me to participate. I also understand that my manager will not be informed of whether I participate or not.

Conservation of data: The videotapes and questionnaires will be initially stored by Mr. Fu in his office at the university. Following analysis, they will be stored in Dr. Lethbridge's office until completion of the thesis. The derived data will be stored on Dr. Lethbridge's computer in his university officefor 5 years after the time of publication. The investigators, Lianjiang Fu and Abdelwahab Hamou-Lhadj will have access to the data along with their supervisor Dr. Lethbridge.

Acceptance: I agree to participate in the above research. I understand that by accepting to participate I am in no way waiving my right to withdraw from the study.

If I have any questions, I may contact any of the researchers. If I have any ethical concerns regarding my participation in this study, I may contact the Protocol Officer for Ethics in Research, University of Ottawa, Tabaret Hall, 550 Cumberland Street, Room 159, Ottawa, ON (613) 562-5841 or ethics@uottawa.ca.

There are two copies of the consent form, one of which is mine to keep.

Participant's signature:	*(Signature)*	Date: *(Date)*
Researcher's signature:	*(Signature)*	Date: *(Date)*