Techniques to Simplify the Analysis of Execution Traces for Program Comprehension

by

Abdelwahab Hamou-Lhadj

Thesis submitted to the Faculty of Graduate and Post-Doctoral Studies in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

Ottawa-Carleton Institute for Computer Science
School of Information Technology and Engineering
University of Ottawa,
Ottawa Ontario Canada

Abstract

Understanding a large execution trace is not easy task due to the size and complexity of typical traces. In this thesis, we present various techniques that tackle this problem.

Firstly, we present a set of metrics for measuring various properties of an execution trace in order to assess the work required for understanding its content. We show the result of applying these metrics to thirty traces generated from three different software systems. We discuss how these metrics can be supported by tools to facilitate the exploration of traces based on their complexity.

Secondly, we present a novel technique for manipulating traces called trace summarization, which consists of taking a trace as input and return a summary of its main content as output. Traces summaries can be used to enable top-down analysis of traces as well as the recovery of the system behavioural models. In this thesis, we present a trace summarization algorithm that is based on successive filtering of implementation details from traces. An analysis of the concept of implementation details such as utilities is also presented.

Thirdly, we have developed a scalable exchange format called the Compact Trace Format (CTF) in order to enable sharing and reusing of traces. The design of CTF satisfies well-known requirements for a standard exchange format.

Finally, this thesis includes a survey of eight trace analysis tools. A study of the advantages and limitations of the techniques supported by these tools is provided.

The approaches presented in this thesis have been applied to real software systems. The obtained results demonstrate the effectiveness and usefulness of our techniques.

Acknowledgments

I would like to express my gratitude to my Ph.D. supervisor Dr. Timothy Lethbridge. His enthusiasm, vast knowledge, and great skills to explain things clearly and simply added considerably to my graduate experience. Throughout my studies, Tim provided encouragement, sound advice, prompt feedback, good company, and lots of great ideas. I thank him for that.

I would also like to thank the members of my PhD committee for the feedback they provided at all levels of this research during the candidacy paper defence as well the Ph.D. oral defence.

I must also acknowledge the software engineers at QNX Software Systems for their participation to the brainstorming session. In particular, I appreciate the support and interest expressed by Thomas Fletcher from QNX to the content of this research. Also, I want to thank the software engineers of Mitel Networks for motivating the necessity to have efficient techniques for manipulating execution traces. They also provided me with lengthy execution traces that allowed me to conduct initial analysis that form the foundation of the work presented in this thesis.

My appreciation also goes to the software developers of the Weka system for creating this great tool and making it open, and for evaluating some of the results presented in this thesis.

This research would not have been possible without the financial assistance of QNX, Mitel Networks, CSER, NCIT, NSERC, and the University of Ottawa School of Graduate and Postgraduate Studies. Thanks to all these organisations.

I would also like to thank my whole family for the support they provided me through my entire life. In particular, I would like to thank my parents and my brothers Fateh and AbdelKrim for their continual support and encouragement.

A special thanks goes to my wife Gligorka for always believing in me and giving me the extra strength, motivation, and love necessary to complete this thesis.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

The objective of this thesis is to develop and evaluate techniques to facilitate the analysis and understanding of the content of large execution traces.

The particular techniques that we propose will be of use are, 1) capabilities to compact and summarize traces that involve removing details such as utilities; 2) an ability to measure traces so their size and complexity can be more easily seen, and 3) an ability to exchange information among trace analysis tools in a standard and efficient way.

In the remainder of this chapter we will motivate the thesis and summarize the contributions in more detail.

1.1 Problem and Motivations

Maintaining[1] large software systems is not an easy task. The difficulties encountered by maintainers are partially attributable to the fact that changes made to the implementation of systems are usually not reflected in the design documentation. This can be due to various reasons including time-to-market constraints, the cost of changing the documentation not justifying the benefit, the initial documentation being too poor to modify, etc. As a result, the gap between a system's implementation and its design models becomes large.

Without consistent or adequately complete documentation, maintainers are faced with the inevitable problem of understanding how the system is implemented prior to undertaking any

[1] By 'maintenance' we mean any change to software beyond its first release or iteration; i.e. development of software where there is already an existing system that is to be changed. This is the broadest possible meaning of the term. It includes adding new features, creating new iterations, as well as classic adaptive and corrective maintenance.

maintenance task. Research into the discipline of *program comprehension* aims to reduce the impact of this problem.

The understanding of the dynamics of a program can be made easier if dynamic analysis techniques are used. Dynamic analysis typically involves the analysis of traces generated from executing the features of the software system under study [Ball 99].

The usefulness of analysing execution traces to help perform software maintenance tasks has been the topic of many studies [De Pauw 93, Jerding 97a , Systä 00a]. For example, in [Jerding 97a], Jerding et al. showed how adding a new feature to a software system can be made easier if trace analysis is used. The authors conducted a case study that consisted of enhancing the NCSA Mosaic web browser [MOSAIC] to support user-configurable external viewers. In order to achieve their goal, they used their trace analysis tool called ISVis to examine the way Mosaic deals with its built-in external viewers. As a result, the authors were capable to quickly uncover the components of the system that needed to be modified and hence insert the changes. Another example would be the study conducted by Systä [Systä 00a]. The author used a combination of static and dynamic analysis to recover the behaviour of part of a software system called FUJABA [Rockel 98]. The study focused on analysing a particular feature of FUJABA that was known to have a defect in it. The trace corresponding to this feature was generated and used to successfully uncover the causes of the buggy behaviour.

However, the large amount of data generated from the execution of a software system complicates the process of applying dynamic analysis techniques. To reduce the impact of this issue, most existing tools turn to specific visualization techniques [De Pauw 93, Jerding 97b, Koskimies 96a, Lange 97, Richner 02 , Walker 98, Zayour 02]. As a result, the problem of exploring the content of traces is often seen as the problem of developing usable visualization tools. However, due to the complex nature of most interesting traces, most existing tools recognise the fact that there is a need for more advanced trace analysis techniques.

In this thesis, we present a set of techniques that aim to simplify the analysis of large traces. These techniques are independent of any visualization scheme. In other words, we investigate what makes traces hard to use and design solutions to reduce this complexity. The techniques presented in this thesis range from exploring the content of traces to developing a scalable model for sharing and reusing large traces.

This chapter is organized as follows: in the next section, we discuss the type of traces used in the thesis. In Section 1.3, we present the thesis contributions. Finally, the detailed thesis outline is presented in Section 1.4.

1.2 The Focus on Traces of Routine Calls

We chose, in this thesis, to focus on traces of *routine calls*[2]. Such traces are at an intermediate level of abstraction between highly detailed *statement-level* traces and traces of *interactions*, such as messages, among high level system components. The former are rarely used since they tend to produce vastly more data than needed, whereas the latter can only reveal the architectural characteristics of the system. Most of the techniques discussed in this thesis could, however, be extended to any type of traces.

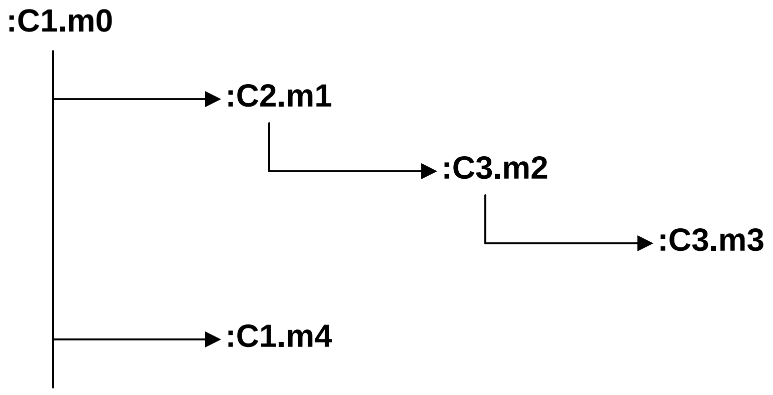


Figure 1.1. Tree representation of a trace of routine calls

We will put particular emphasis on a special class of routine call traces: traces of *method calls* in object-oriented (OO) systems. It is very important to note that the adaptation of the techniques presented in this thesis to procedural systems can be done easily. To allow

[2] We will use the term 'call' and 'invocation' synonymously.

generalization of the concepts presented here to OO as well as procedural systems, we use the term 'routine' to refer to any routine, function, or procedure whether it is a method of a class or not.

Traces of routine calls can easily be depicted using a tree structure as illustrated in Figure 1.1. The figure shows an example of interactions among three objects of the classes C1, C2 and C3 respectively.

To reproduce the execution of an object-oriented system, we need to collect at least the events related to object construction and destruction, as well as method entry and exit [De Pauw 93]. Additional information can be collected such as events related to thread execution.

It is very common that traces, once generated, are saved in text files. A trace file usually contains a sequence of lines in which each line represents an event. An example of this representation is given by Richner and Ducasse in [Richner 02]. Each line records: The class of the sender, the identity of the sender, the class of the receiver, the identity of the receiver and the method invoked. The order of calls can be maintained in two ways, either each entry and exit of the method is recorded, which results is a very large trace file, or an integer is added to represent the nesting level of the calls. In this case, we do not need to record the exit event of a method as shown by the following illustration.

Assuming that obj1 is a unique identifier of the object :C1 and that obj2 represents :C2 and obj3 represents :C3, the trace file that corresponds to the trace of Figure 1.1 should have the following events:

Sender Class	Sender ID	Receiver Class	Receiver ID	Called Method	Nesting Level
c1	obj1	c2	obj2	m1	1
c2	obj2	c3	obj3	m2	2
c3	obj3	c3	obj3	m3	3
c1	obj1	c1	obj1	m4	1

There exist various techniques for generating traces. The first technique is based on instrumenting the source code, which consists of inserting probes (e.g. print statements) at appropriate locations in the source code. In the context of object-oriented systems, probes are usually inserted at each entry and optionally each exit of every method. Instrumentation is usually done automatically.

Another technique for collecting run-time information consists of instrumenting the execution environment in which the system runs. For example, the Java Virtual Machine can be instrumented to generate events of interest. The advantage of this technique is that it does not require the modification of the source code.

Finally, it is also possible to run the system under the control of a debugger. In this case, breakpoints are set at locations of interest (e.g. entry and exit of a method). This technique has the advantage of modifying neither the source code nor the environment; however, it can slow down considerably the execution of the system.

1.3 Research Contributions

The major research contributions of this thesis are:

- A survey of trace analysis tools and techniques.
- A set of metrics for measuring various properties of an execution trace.
- A set of techniques for trace summarization that can be used to enable top-down analysis of an execution trace, and recover the system's behavioural design models.
- An exchange format for exchanging and reusing traces of routine (method) calls.

The remainder of this section elaborates on these contributions. The subsequent section presents the outline of the thesis.

1.3.1 A Survey of Trace Analysis Tools and Techniques

We studied the techniques implemented in eight trace analysis tools. The contribution of this part the thesis is to understand the advantages and limitations of the supported features, particularly techniques used to cope with the large size of execution traces; the levels of granularity of the analysis permitted by the existing techniques; and the internal formats used to represent traces.

1.3.2 Trace Metrics

Using existing trace analysis tools, an analyst may apply many operations to a particular trace and still be left with a large amount of data to analyze. This problem is mainly due to the fact that none of the existing tools is built upon specific metrics for assessing the work required for analyzing traces.

To address this issue, we propose that if various aspects that contribute to a trace's complexity could be measured and if this information could be used by tools, then trace analysis could be facilitated. For this purpose, we present a set of simple and practical metrics that aim to measure various properties of execution traces. We also show the results of applying these metrics to traces of three software systems and suggest how the results could be used to improve existing trace analysis tools.

1.3.3 Summarizing the Content of Large Traces

In this thesis, we introduce the concept of trace summarization and discuss how it can be used to (a) enable top-down analysis of traces, and (b) recover high-level behavioural design models from traces. Similar to text summarization, where abstracts can be extracted from large documents, the aim of trace summarization is to take an execution trace as input and return a summary of its main content as output. The process is performed is a semi-automatic way. The summary can then be converted into a UML sequence diagram and used as a high-level behavioural model. Our approach to trace summarization is based on the way software engineers use traces in an industrial setting. After a discussion that took place at QNX Software Systems (the company that supported some of this research), the participants argued that when exploring traces, they would like to have the ability to look at the 'big picture' first and then dig into the details. They referred to the elements of the trace that can be filtered out as implementation details such as utilities.

Trace summarization is based on filtering traces by removing implementation details such as utilities. To achieve this goal, we first show how fan-in analysis can be used to detect the utility components of the system. Then, we present a trace summarization algorithm that uses fan-in analysis as its main mechanism. The algorithm also assumes that software engineers will manipulate the resulting summary in order to adjust its content to their specific needs. We applied our approach to a trace generated from an object-oriented system called Weka [Weka, Witten 99] that initially contains 97413 method calls. We succeeded to extract a summary from this trace that contains 453 calls. According to the developers of the Weka system, the resulting summary is an adequate good high-level representation of the main interactions of the traced scenario.

1.3.4 An Exchange Format for Representing Execution Traces

Existing trace analysis tools use different formats for representing traces, which hinders interoperability. To allow for better synergies among trace analysis tools, it would be beneficial to develop a standard format for exchanging traces.

An exchange format consists of two main components: Firstly, a metamodel that represents the entities to exchange and their interconnections, and secondly the syntactic form of the file that will contain the information to exchange [Bowman 99, Jin 02]. In this research, we introduce a metamodel for representing traces of routine (method) calls referred to as the Compact Trace Format (CTF). We also validate CTF with respect to well-known requirements for a standard exchange format. We discuss how existing syntactic forms namely GXL (Graph eXchange Language) [Holt 00] and TA (Tuple Attribute Language) [Holt 98] can be used to 'carry' the data represented by CTF.

1.4 Thesis Outline

The remaining chapters of this thesis are:

Chapter 2 – Background

This chapter starts by presenting the different areas that are related to our research, namely software maintenance, reverse engineering, and program comprehension. The remainder of the chapter surveys the literature on the various trace analysis techniques that are implemented in eight trace analysis tools. The chapter continues with a detailed analysis of these techniques and concludes by classifying the studied tools according to the trace analysis features they support.

Chapter 3 – Trace Metrics

This chapter starts with explaining the usefulness of having well defined metrics of measuring properties of traces. It proceeds with an introduction of key concepts that will be used for the rest of the thesis namely the concept of comprehension units. The chapter continues by presenting a set of metrics that are designed to characterize the work required to understand the traces. The design of these metrics is compliant with the Goals/Questions/Metrics (GQM) model [Basili 94]. An empirical study of several traces of three different object-oriented systems using these metrics is then presented. The chapter continues by briefly discussing how these metrics should be supported by trace analysis tools. Some of the metrics presented in this thesis rely on the fact that any tree structure can be turned into an ordered directed acyclic graph (DAG) by representing similar subtrees only once. This motivated us to include in this chapter an algorithm that we developed for transforming a call tree into a DAG. We want to note that many other sections of this thesis will also refer to this algorithm.

Chapter 4 – Trace Summarization

This chapter starts by discussing the concept of trace summarization, its applications, and its similarities with the concept of text summarization. An approach for achieving trace summarization is then presented. The chapter proceeds by discussing the concept of implementation details including utility components. After this, a utility detection technique based on fan-in analysis is presented.

Chapter 5 – Case Study

This chapter focuses on a case study used to evaluate the trace summarization concepts presented in the previous chapter. In the beginning of the chapter, we present the target system from which we generated the trace to summarise. The chapter continues by describing the evaluation process. The quantitative results of applying the trace summarization process are then presented and discussed. The chapter proceeds with a questionnaire-based evaluation that aims to capture the feedback of the developers of the system under study.

Chapter 6 – The Compact Trace Format

The chapter starts by stating the advantages of having a standard exchange format for exchanging execution traces. An introduction of the Compact Trace Format (CTF) is then presented. The chapter continues with presenting related work. It proceeds by citing the requirements used to guide the design of CTF. The remaining sections discuss the CTF metamodel, semantics, and syntactic form. In the end of the chapter, we discuss the adoption of CTF.

Chapter 7 – Conclusions

This chapter starts by discussing the contributions of the thesis as well as opportunities for future research. Closing remarks are then presented at the end of the chapter.

Chapter 2. Background

2.1 Introduction

Our research is intended to help software engineers understand the content of large execution traces. In the next section, we present background information and terminology that are necessary to understand this thesis, and situate it in context. In Section 2.3, we present a survey of existing trace analysis tools and techniques. A discussion of the advantages and limitations of these techniques is the subject of Section 2.4. Finally, in Section 2.5, we summarise the content of the survey by listing the various trace analysis techniques and classify the tools chosen in this study.

2.2 Related Topics

2.2.1 Software Maintenance

Software maintenance is defined as the modification of a software system after delivery [ANSI/IEEE Std]. These modifications are usually grouped into four categories [Pfleeger 98]:

- **Corrective maintenance:** This involves fixing faults that caused the system to fail.
- **Preventive maintenance:** This is concerned with preventing failures before they even occur. For example, a software engineer might add fault handlers if the possibility of a potential fault is noticed.
- **Adaptive maintenance:** This consists of making changes in existing software to accommodate new requirements.

- **Perfective maintenance:** This involves making improvements to the existing system in order to make it easier to extend and add new features in the future.

The case studies conducted by Jerding et al. [Jerding 97a] and Systä [Systä 00a] and that are discussed in the introductory chapter of this thesis show how traces can help perform adaptive and corrective maintenance tasks. However, maintaining a poorly documented system is a hard task and requires the understanding of its various artefacts including the source code, run-time information, etc. In this thesis, we focus on techniques for simplifying the exploration and understanding of large execution traces.

2.2.2 Program Comprehension

Making changes to a software system requires a prior understanding of how it works [Chapin 88]. Understanding a program involves usually four main strategies also referred to as program comprehension models [Pennington 87, Storey 97, Von Mayrhauser 95]:

Bottom-up Model:

Using the bottom-up model, a software engineer proceeds with comprehending the source code by building abstractions from it. This strategy usually involves reading the source code and mentally grouping together low-level programming details, called *chunks*, in the form of higher-level domain concepts. This process is repeated until an adequate understanding of the program is gained [Pennington 87, Storey 97, Von Mayrhauser 95].

Similarly, the bottom-up understanding of a trace of routine calls consists of exploring the content of various subtrees of the trace. This exploration typically involves expanding/collapsing subtrees, searching for similar execution patterns, etc. as shown by Jerding et al. [Jerding 97a]. Most existing trace analysis tools offer a variety of operations that support this strategy (see Section 2.5).

Top-down Model:

The top-down model is a hypothesis-driven comprehension process. Using this strategy, a programmer first formulates hypotheses about the system functionality. Then, he or she verifies whether these hypotheses are valid or not [Brooks 83]. This process usually leads to creating other hypotheses, forming a hierarchy of hypotheses. This continues until the low-level hypotheses are matched to the source code and proven to be valid or not. Top-down analysis is usually performed by software engineers who have some knowledge of the system under study.

By analogy, we can think of top-down analysis of a trace as a process that consists of two steps: a) formulating hypotheses about the content of a trace in term of what it does, and b) validating these hypotheses by matching them to the actual content of the trace.

Formulating hypotheses ought to be easy for software engineers who have some knowledge of the software under study. However, the level of details represented in a trace can render the second step (i.e. validating the hypotheses) hard to perform. The problem is that software engineers do not have a simplified view of a trace they can readily work with. One of the main contributions of this thesis is to enable top-down analysis by extracting summaries from large traces. This concept is discussed in more detail in Chapter 4.

The Integrated Model:

Von Mayrhauser and Vans present a model that combines the top-down and bottom-up approaches that they refer to as the *integrated model* [Von Mayrhauser 94, Von Mayrhauser 95]. The authors conducted several experiments with real world systems and presented empirical results that cover a variety of maintenance activities including adaptive maintenance, corrective maintenance and reengineering [Von Mayrhauser 96, Von Mayrhauser 97, Von Mayrhauser 98]. One of the most important findings of their research is that maintainers tend to switch among the different comprehension strategies depending on the code under investigation and their expertise with the system. We believe that this applies also to understanding execution traces.

Partial Comprehension:

Erdos and Sneed argue that it is not necessary to understand the whole system if only part of it needs to be maintained [Erdos 98]. They suggest that most software maintenance tasks can be met by answering a set of basic questions, which are:

- How does control flow reach a particular location?
- Where is a particular subroutine or procedure invoked?
- What are the arguments and results of a function?
- Where is a particular variable set, used or queried?
- Where is a particular variable declared?
- What are the input and output of a particular module?
- Where are data objects accessed?

The two first questions are clearly addressed by analyzing traces of routine calls. The remaining questions can be answered using traces in various ways: extending the analysis of routine-call traces to consider routine arguments and return values; combining trace analysis techniques with static analysis of the source code, etc. In this thesis, we only focus on understanding the flow of execution of a system by analyzing the routine calls generated from executing the software features, leaving the other aspects of this section as a line of future research.

2.2.3 Reverse Engineering

Program comprehension can be made easier if reverse engineering tools are used. Reverse engineering can be defined as: "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 at a higher level of abstraction" [Chikofsky 90]. The objectives of reverse engineering include coping with the increasing complexity of software, recovering lost information, recovering high-level models of the system, etc. [Chikofsky 90, Biggerstaff 89].

Tilley identifies three basic activities that are involved in the reverse engineering process [Tilley 96]:

- **Data gathering:** This activity covers the techniques that are used to gather the data. This can be done either by performing static analysis or dynamic analysis. In the context of this thesis, traces are generated using instrumentation techniques.
- **Knowledge organization:** Once the data is gathered, we need to structure it in order to ease its storage and retrieval. In this thesis, we introduce the Compact Trace Format

(CTF) as a metamodel for representing traces of routine calls. CTF is discussed in more detail in Chapter 6.

- **Information exploration:** Reverse engineering tools need to implement exploration techniques to allow fast analysis of the data. Most of the concepts presented in this thesis such as trace metrics (Chapter 3), trace summarization (Chapter 4), and CTF (Chapter 6) are designed to help explore the content of traces in an efficient way. However, in order to be effective, these concepts need to be integrated into a tool suite.

2.3 A Survey of Trace Analysis Tools and Techniques

In this section, we present a survey of existing trace analysis tools and techniques. We selected several tools in order to achieve good coverage of the types of features available; we did not attempt to examine in detail all the tools that exist in the literature. Some of the tools described are not freely available, so our analysis is based on the scientific publications that describe them. For simplicity, we exclude from our analysis those tools that deal with distributed systems and multi-threaded traces.

The tools selected for this study are: Shimba, ISVis, Ovation, Jinsight, Program Explorer, AVID, Scene, and The Collaboration Browser. The next subsections describe these tools and the techniques they support in more detail. A discussion about the advantages and limitations of these techniques is presented in Section 2.4. A summary of this study is presented in Section 2.5.

Pacione et al. [Pacione 03] conducted a study in which they evaluated five general purpose dynamic analysis tools including debuggers based on the way these tools enable a number of reverse engineering tasks such as identifying the system architecture, identifying design patterns, etc. The study presented in this section focuses on the way tools support the analysis of execution traces to help with program comprehension, which represents a more specific scope than the work presented by Pacione et al.

Much of the material in this section is adapted and expanded from a paper published in the 14th IBM Conference of the Centre for Advanced Studies on Collaborative Research (CASCON), 2004 [Hamou-Lhadj 04a].

2.3.1 Shimba

Systä presents a reverse engineering environment, called Shimba, which combines static and dynamic analysis to understand the behaviour of Java software systems [Systä 01, Systä 99, Systä 00a]. Static analysis is used to select a set of components that need to be examined later during dynamic analysis. Systä's approach is based on the assumption that a software engineer does not need to trace the whole system if only a specific part needs to be analyzed.

Shimba extracts the system artefacts and their interdependencies from the Java class files and enables them to be viewed using a reverse engineering tool called Rigi [Müller 88]. In Rigi, the artefacts are shown as nodes, and the dependencies are shown as directed edges among the nodes. Shimba considers the following system artefacts: classes, interfaces, methods, constructors, variables, and static initialization blocks. The dependencies among these artefacts include inheritance relationships, containment relationships (e.g. a class contains a method), call relationships and so on. Using Rigi, a software engineer can run a few scripts to exclude the nodes that are not of interest and keep only those he or she wants to investigate. Breakpoints are then set at events of interest (e.g. the entry of a method or a constructor) of the selected classes. The target system is executed under a customized debugger and the trace is collected.

The next step is to analyze the trace. For this purpose, a software engineering tool called SCED is used [Koskimies 96b]. SCED is a forward engineering tool that permits the creation and manipulation of scenario diagrams, which are similar in principle to UML sequence diagrams. SCED also has the ability to synthesise state machines given several scenario diagrams.

However, neither Rigi nor SCED allow direct manipulation of the content of traces. By manipulation, we mean searching for specific components, collapsing and expanding parts of the traces, etc.

Although the execution trace represents only the classes that were selected using static analysis, Systä recognizes the fact that these traces may still be large. To overcome this problem, she applies the Boyer-Moore string matching algorithm [Boyer 97] to SCED scenario diagrams in order to detect repeated sequences of identical events that she refers to as behavioural patterns. She distinguishes between two kinds of behavioural patterns: The first type involves contiguous repetitions of sequences of events due to loops. These patterns are shown using a repetition construct that exists in SCED. The second type consists of behavioural patterns that occur in a non-contiguous way in the trace. They are represented using *subscenario* constructs, which consist of boxes that are added to SCED scenario diagrams. A subscenario box encapsulates the events of an instance of the behavioural pattern. A user can double click on a subscenario box to display the detailed pattern information.

2.3.2 ISVis

ISVis is a visualization tool that supports analysis of execution traces generated from object-oriented systems [Jerding 97a, Jerding 97b]. ISVis is based on the idea that large execution traces consist of recurring patterns, referred to as *interaction patterns*, and that visualizing these patterns is useful for reverse engineering. Interaction patterns are in fact the same as the behavioural patterns used in Shimba.

The execution trace is visualized using two kinds of diagrams: the *information mural* and the *temporal message-flow diagram* (a variant of UML sequence diagrams). The two diagrams are connected and are presented in one view called the scenario view as shown in Figure 2.1. The information mural uses visualization techniques to create a miniature representation of the entire trace that can easily show repeated sequences of events. The temporal message-flow diagram is used to display the detailed content of the trace. The software engineer can spot a pattern on the information mural view, select it and investigate its content using the temporal message-flow diagram.

To deal with the size explosion problem, ISVis uses an algorithm that detects patterns of identical sequences of calls. Given a pattern, the user can search in the trace for an exact match, an interleaved match, a contained exact match (components in the trace that contain components in the pattern) and a contained interleaved match. Additionally, the user can use wildcards to formulate more general search queries.

Another important feature of ISVis is that trace events can be abstracted out using the containment relationship. For example, a user can decide to hide the classes that belong to the same subsystem and only show the interactions between this subsystem and the other components of the trace.

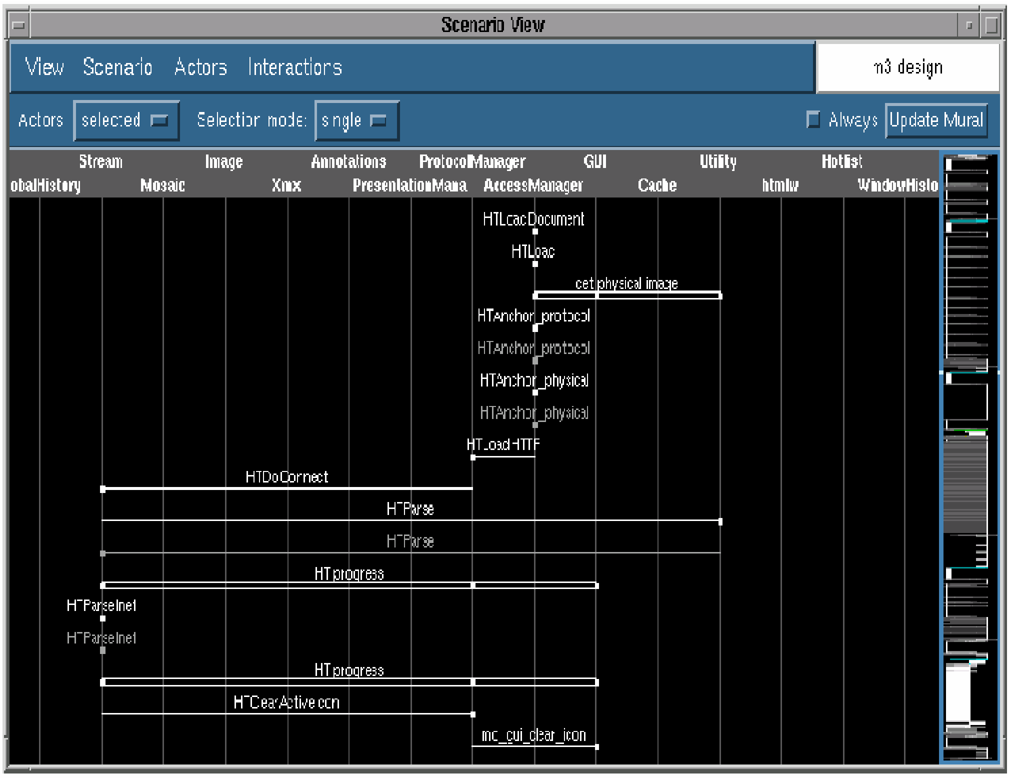


Figure 2.1. ISVis scenario view which consists of the information mural view (on the right) and the temporal message-flow diagram (center).

In [Jerding 97b], Jerding et al. describe a data structure for the internal representation of traces. This is based on the idea that a trace of method calls, which is a tree structure, can be transformed into its compact form resulting in an ordered directed acyclic graph where the same subtrees are represented only once. This representation allows ISVis to scale up to very large traces.

2.3.3 Ovation

De Pauw et al. introduce a tool called Ovation [De Pauw 98]. Unlike Shimba and ISVis, Ovation visualizes traces using a view based on tree structures called the execution pattern view (Figure 2.2). According to the authors, the execution pattern view is less cumbersome than UML sequence diagrams.

This view lets the user browse the program execution at various levels of detail. For example the user can collapse and expand subtrees, show only messages sent to a particular object, remove contiguous repetitions of sequences of calls, zoom in and out the trace panel and many other useful visualization features.

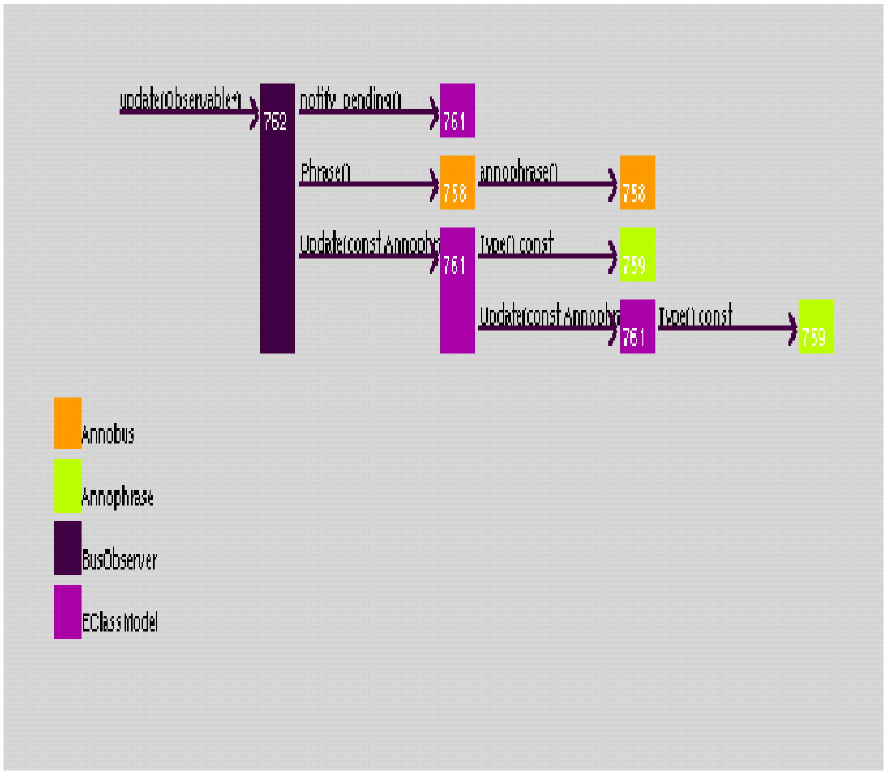


Figure 2.2. The Execution Pattern View of Ovation

To overcome the size explosion problem, similar sequences of events are shown as instances of the same pattern. The patterns are then color coded to allow software engineers to notice them easily.

However, the authors notice that *exact matching* is too restrictive and does not reduce the size problem very much. They therefore turn to inexact matching: To achieve this, they present a set of matching criteria that can be used to decide when two sequences of events can be considered equivalent [De Pauw 98]. The main matching criteria are the following:

- **Identity:** Two sequences of calls are considered instances of the same pattern if they have the same topology: same order, objects, methods, and so on. This is the base case, i.e. exact matching.
- **Class Identity:** If two sequences of calls involve the same classes but different objects then they can be considered similar according to this criterion.
- **Depth-limiting:** This criterion consists of comparing two sequences (which represent two subtrees of the call tree) up to a certain depth.
- **Repetition:** It is very common to have two different sequences of calls that differ only by the number of repetitions due to loops and recursion. If this number is ignored then these two sequences can be considered equivalent.
- **Polymorphism:** This criterion suggests considering two subclasses of the same base class as the same. However, this applies only if they invoke the same polymorphic operations.

2.3.4 Jinsight

Jinsight is a Java visualization tool that shows the execution behaviour of Java programs [De Pauw 02]. Jinsight provides several views that can be very helpful for detecting performance problems. These views can be summarized as follows:

- **The Histogram View:** It helps the analyst detect performance bottlenecks. It also shows object references, instantiation and garbage collection.
- **The Execution View:** This view displays the program execution sequence (Figure 2.3). It helps the analyst understand concurrent behaviour and thread interactions, as well as detect deadlocks.
- **The Reference Pattern View:** This is used to show the interconnections among objects. For this purpose, Jinsight implements pattern recognition algorithms to reduce the information overhead. In fact, this view is equivalent to the pattern execution view of Ovation introduced by the same authors and that was described in the previous subsection.

- **The Call Tree View:** This shows the sequence of method calls, including the number of calls and their contribution to the total execution time as shown in Figure 2.4.

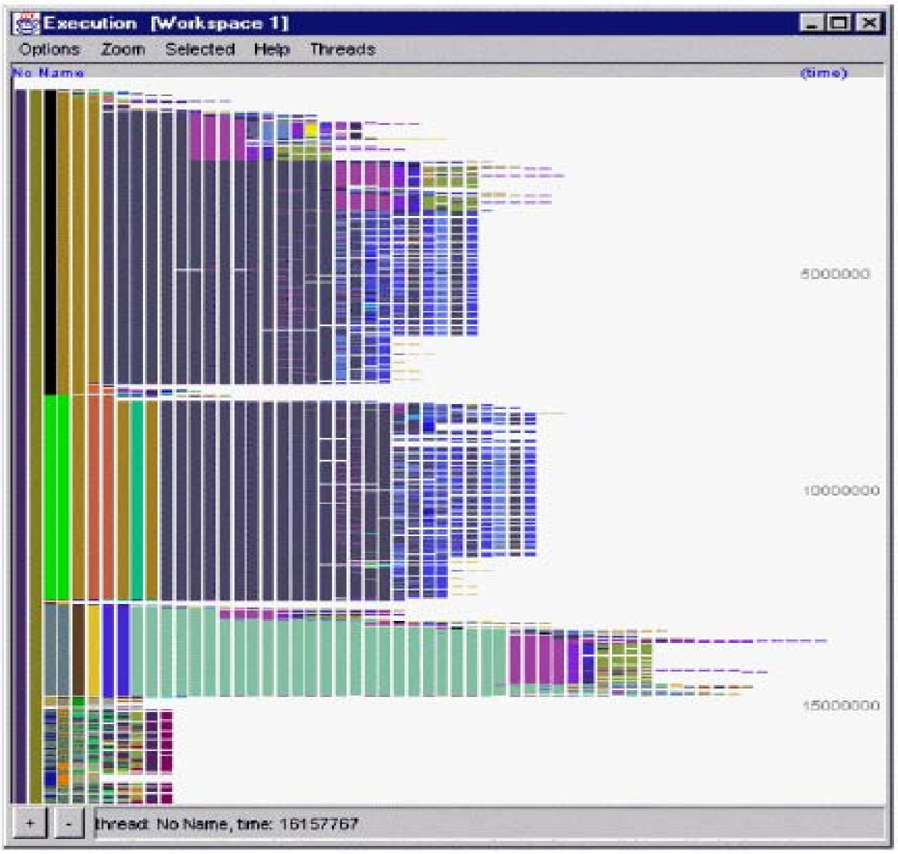
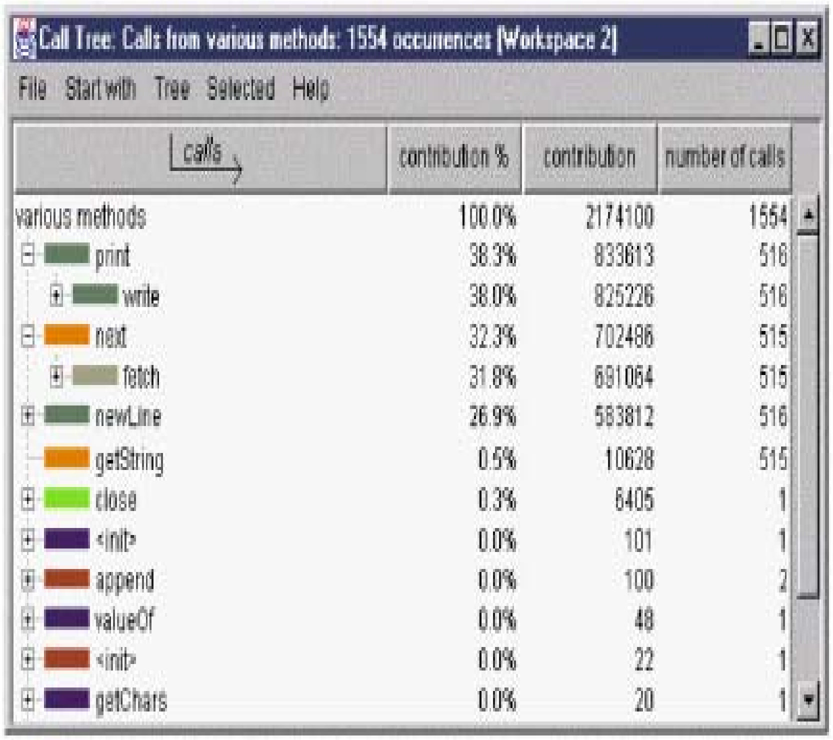


Figure 2.3. Jinsight Execution View

Jinsight is heavily tuned towards performance analysis rather than program comprehension. However, according to the authors, the reference pattern and the call tree views can be used for general understanding of the system execution.

Jinsight uses a model for representing the information about the execution of an object-oriented program introduced by De Pauw et al. in [De Pauw 94]. Events of interest in this model are object construction/destruction and method invocation and return. The authors organized these artefacts in a four-dimensional event space having axes for classes, instances, methods and time as shown in Figure 2.5. Each point corresponds to an event during program execution. Information is extracted by traversing or projecting one or more dimensions of the space in different combinations to produce subspaces.


Call Tree: Calls from various methods: 1554 occurences [Workspace 2]
File Start with Tree Selected Help

calls	contribution %	contribution	number of calls
various methods	100.0%	2174100	1554
⊟ 🟩 print	38.3%	833613	516
⊞ 🟩 write	38.0%	825226	516
⊟ 🟧 next	32.3%	702486	515
⊞ 🟫 fetch	31.8%	691064	515
⊞ 🟩 newLine	26.9%	583812	516
🟧 getString	0.5%	10628	515
⊞ 🟩 close	0.3%	6405	1
⊞ 🟪 <init>	0.0%	101	1
⊞ 🟥 append	0.0%	100	2
⊞ 🟪 valueOf	0.0%	48	1
⊞ 🟥 <init>	0.0%	22	1
⊞ 🟪 getChars	0.0%	20	1

Figure 2.4. Jinsight Call Tree View

However, the event space of even a small system might be very large. To overcome this problem, the authors introduce the concept of call frames. A call frame is a combination of events that depicts a communication pattern among a set of objects. For example, consider a method m1 of class c1 that calls a method m2 of class c2. This sequence typically involves an object o1 of c1 and an object o2 of c2 (this does not apply if static methods are used). The whole sequence is saved as one call frame instead of saving every single event of this sequence.

Statistical information can also be computed at the same time the system executes. For example, we can associate with the previous call frame the number of times the method m1 calls m2 or the number of times the class c1 calls c2. For this purpose, several data structures are used to represent the call frames.

Although this technique might result in a significant reduction of the number of events, it is more tuned towards performance analysis than program comprehension. Indeed. Most of the visualization views supported by Jinsight exhibit statistical information only and are similar in principle to the way profilers work.

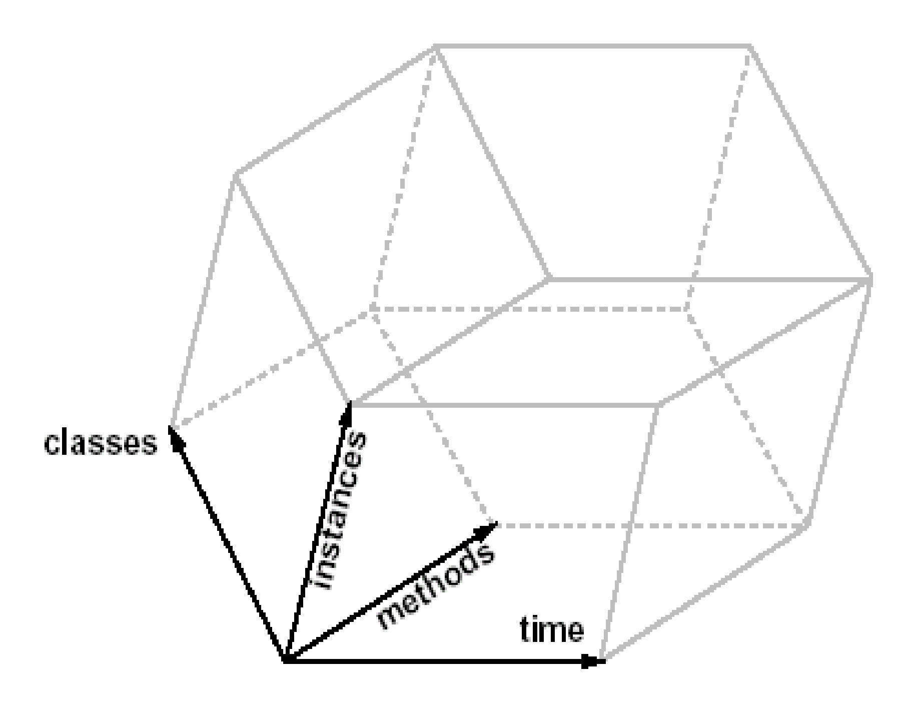


Figure 2.5. Four-dimensional event space used by Jinsight to represent trace events

2.3.5 Program Explorer

Program Explorer is a C++ exploration tool that focuses on analyzing interactions among objects and classes [Lange 97]. The authors start by introducing a common model and notation for OO program execution. Interactions among objects are modeled using a directed graph called the Interaction Graph (Figure 2.6). The nodes of the graph represent objects and the arcs represent method invocations. Arcs are labelled with the name of the method, the time at which the invocation of the method takes place and the time at which the execution returns to the caller.

To overcome the size explosion problem, Program Explorer uses several filtering techniques, which are:

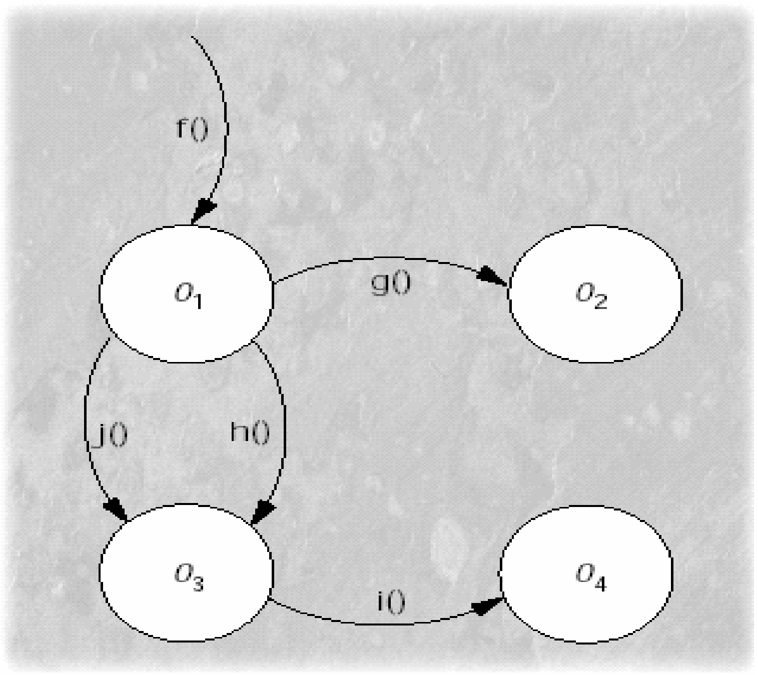


Figure 2.6. The interaction graph is used by Program Explorer to represent object interactions.

- **Merging:** Using Program Explorer, the analyst can merge arcs that represent identical methods among pairs of objects. The resulting graph is called the Object Graph and emphasizes how objects interact but hides the order and the multiplicity of invocations. Furthermore, the analyst can merge objects of the same classes into one node to reduce the number of nodes. The resulting graph is called the Class Graph and focuses on class interaction rather than object interaction. An example of the class graph is shown in Figure 2.7. This is similar in principle to the pattern matching criteria that are supported by Ovation.

- **Pruning:** Pruning is the process of removing information from the interaction graph in order to reduce its size. Program Explorer implements three kinds of pruning techniques: object pruning, method pruning and class pruning. Pruning an object consists of removing its corresponding node from the interaction graph. The incoming and outgoing arcs of this node are also removed. Method pruning consists of performing the same task on specific methods. The subsequent invocations that derive from them are also removed. Pruning can also apply to inheritance hierarchies and is called class pruning. Class pruning relies on the fact that pruning a superclass method will result in pruning this method at the subclasses level. The several pruning techniques are exactly similar to the many different browsing capabilities that exist in ISVis, Ovation and Jinsight.

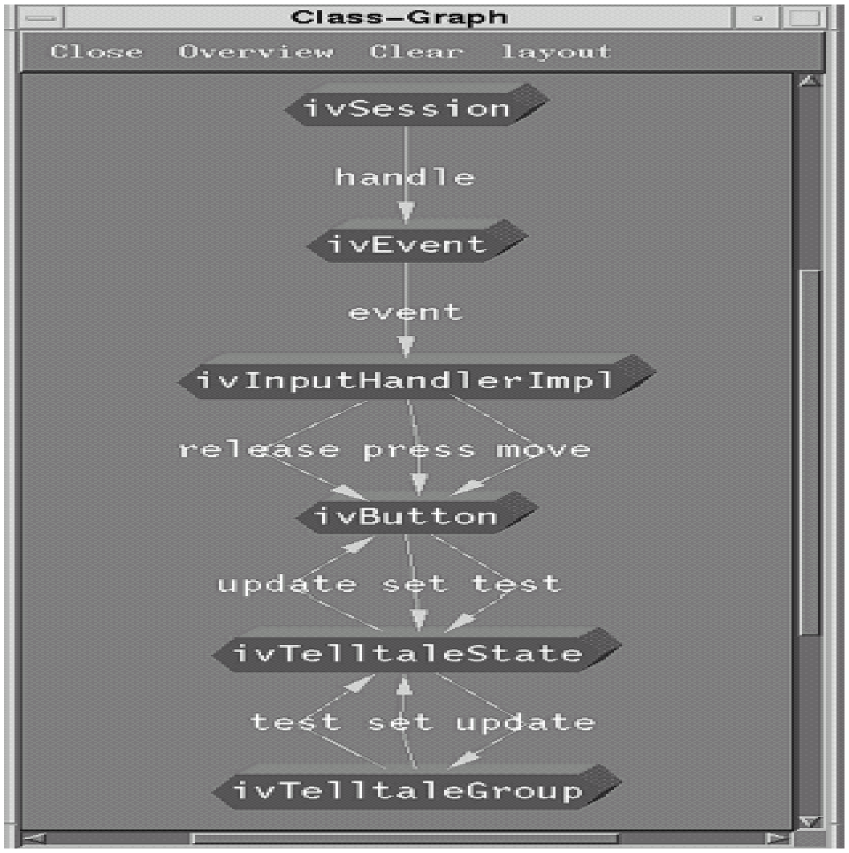


Figure 2.7. A class graph focuses on class interactions rather than object interactions

- **Slicing:** Object slicing is similar to dynamic slicing [Korel 97] and aims at keeping all the activation paths in which this object participates. That is, all the other paths are removed from the graph. Method slicing accomplishes the same task as object slicing except that it focuses on keeping specific methods of an object.

2.3.6 AVID

Walker et al. describe a tool, called AVID (Architecture Visualization of Dynamics in Java Systems), for visualizing dynamic behaviour at the architectural level [Walker 98]. AVID uses run-time information and a user-defined architecture of the system to create a dynamic view of the system components and the way they interact.

First, the analyst creates a trace describing the method calls and the instantiation and destruction of objects. Next, he or she needs to cluster classes into components called entities. In AVID, clusters are represented as boxes and the dynamic relationships extracted from the trace as directed arcs as shown in Figure 2.8. An arc between two entities A and B is labelled with the number of calls the methods of the classes in A make to the methods of the classes in B. Instantiation and destruction of objects are shown as bar-chart style of histograms associated with each box.

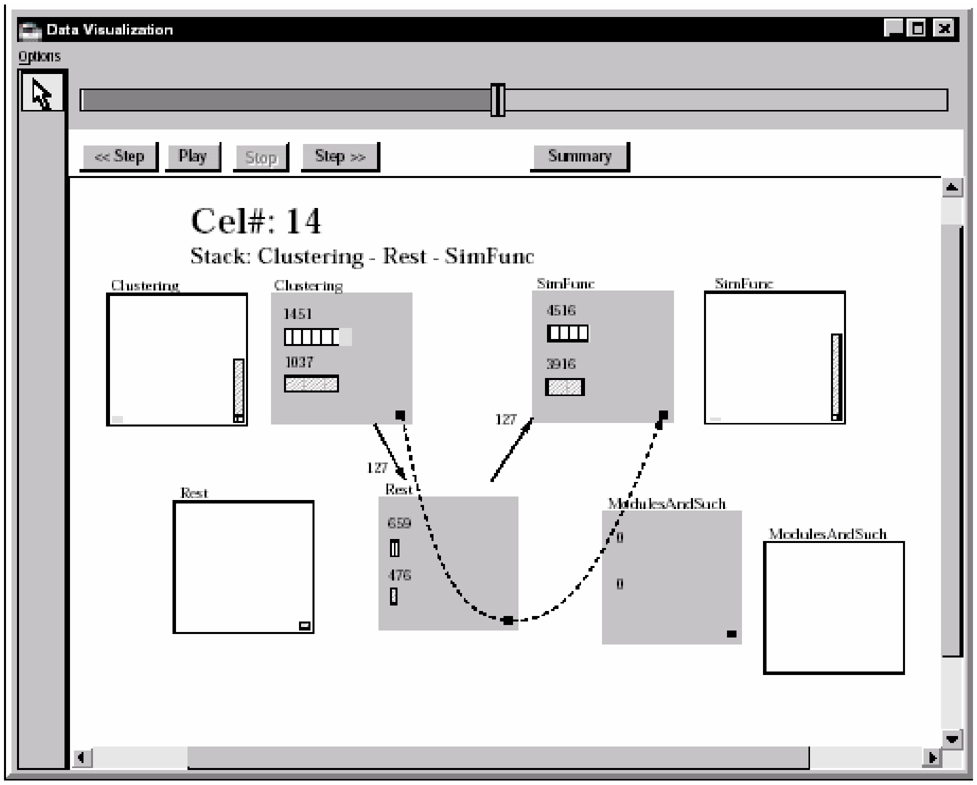


Figure 2.8. Interactions among the system clusters as represented by AVID. Here, the analyst has replayed the execution and stopped at Cel#14

In AVID, the analyst can control the sequence of events he or she wants to visualize. This is done by breaking the execution trace into a sequence of views called *cels*. Animation techniques allow the analyst to show the whole execution cel by cel (which is also called the play mode), stop the animation, as well as go forward and backward. These techniques aim to reduce the information overhead when dealing with large execution traces. Furthermore, AVID contains a summary view in which all the interactions are shown.

Although animation techniques can help reduce the information overhead, traces are very large and there is a need to investigate more techniques to reduce their size. In a recent paper [Chan 03], Chan et al. describe how AVID was improved to consider filtering techniques based on sampling. The authors describe a set of sampling parameters that can be used by the analyst to consider only a sample of the execution trace. For example, the analyst can choose the events that appear after a certain timestamp only, a snapshot of the call stack every x^{th} event, etc.

However, there is a lack of scientific evidence regarding which parameters are best to use. If they work for a given scenario they may not work for others; this is demonstrated in the results of the case studies conducted by the authors of AVID in which some parameter settings worked for one case study but did not work for the other case study.

2.3.7 Scene

Koskimies and Mössenböck present a tool called Scene (Scenario Environment) that is used to produce scenario diagrams from a dynamic event trace [Koskimies 96a]. The authors notice that horizontal scrolling makes the diagrams cumbersome and there is a need for techniques that center the information conveyed by scenario diagrams on the screen. They name this problem the focusing problem and suggest several visualization-oriented techniques to solve it. Among these techniques, we have:

- **Call compression:** This technique consists of collapsing the internal invocations that derive from a given call. A click on this call will result in making its internal invocations visible. Figure 2.9 shows an example of three calls that have been collapsed, which can be expanded by a click from the user.

- **Partitioning:** This feature divides the call tree invoked by a given method into parts. The user can choose to click on a specific part to expand the invocations it encapsulates without opening the other parts.

- **Projection and removal:** This operation enables the user to select an object and show only the interactions that involve its methods. The other interactions are then hidden.

- **Single-step mode:** The single-step mode allows the user to display the internal invocations of a given call one step at a time by clicking on the last visualized call.

Scene also provides a summary view which consists of a matrix that shows how the classes of the system interact among each other.

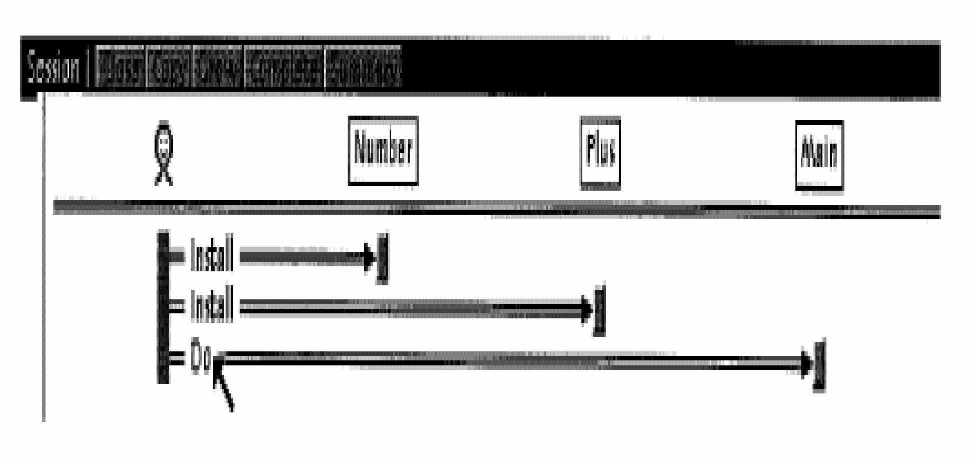


Figure 2.9. The calls Install and Do have been collapsed. The user can click on these calls to see the methods they invoke

2.3.8 The Collaboration Browser

Richner and Ducasse describe a tool called The Collaboration Browser that is used to recover object collaborations from execution traces [Richner 02].

The authors define a collaboration instance as a sequence of method invocations that starts from a given method and terminates at its return point. This includes a single method call that does not generate other calls. Similar collaboration instances define a collaboration pattern (which is similar to Shimba behavioural patterns and ISVis interaction patterns). Similarity is measured according to three kinds of matching criteria:

- **Criteria based on information about the event:** An event in the trace contains information about the sender, receiver and the invoked method. The analyst can choose to include or omit any of these attributes in the matching process. For example, the analyst may decide to ignore the invoked method and match two sequences of calls using the sender and receiver classes (or objects) only. These represent an extension to the criteria that are supported by Ovation.

- **Excluding events:** This category allows the analyst to exclude specific events in the matching scheme. For example, the analyst may decide to ignore events in which an object sends a message to itself, events that appear after a certain depth in the trace, etc.

- **Structure of the collaboration instance:** A collaboration instance is a tree of events. The authors notice that similar collaboration instances may differ in their structure and still represent the same behaviour. Therefore, one can consider two collaboration instances as instances of the same collaboration pattern if they contain the same set of events no matter in which order they occur or their nesting relationships.

Once the classes that constitute a given collaboration are determined, the user can query the trace or the collaboration pattern to extract the role of each of its classes. The role of a class is represented by its public methods. In addition to this, the tool enables the developer to filter out dynamic information by removing classes or methods that are not of interest. It can also display an instance of a collaboration pattern as a UML sequence diagram.

The authors conducted a case study with a framework for the creation of graphical editors called HotDraw. They were interested in understanding the implementation of one aspect of this framework, which is concerned with the tools that are responsible for creating and manipulating figures. First, they instrumented all the methods of the system. Next, they run a short scenario that involves the feature under analysis. The resulting trace contains 53735 method invocations.

To extract collaboration patterns, the authors arbitrarily picked several matching criteria. For example, they decided to ignore self-invocations, limit the depth of invocation to 20 and not consider the tree structure of collaboration instances during the matching process. 183 patterns were generated.

The next step was to query the patterns to extract only the collaboration patterns that describe the implementation of the feature under analysis. This process is iterative and assumes that the analyst has knowledge of the system so as to know what to look for.

2.4 Discussion

In this section, we discuss the pros and cons of the features supported by the above trace exploration tools. The focus is on the following points:

- The models used to represent traces
- The levels of granularity of the analysis permitted by the existing techniques
- The techniques used to cope with the large size of execution traces

2.4.1 Modeling Execution Traces

In order to analyze large program executions, an efficient representation of the event space is needed. Unfortunately, most of the literature about the above tools does not even discuss this aspect.

Among the tools whose literature does discuss modeling issues, ISVis seems to implement the most interesting approach. It uses a graph-theory concept that consists of transforming a rooted labelled tree into a directed acyclic graph by representing identical subtrees only once. This technique has been widely used in trace compression and encoding [Reiss 01, Larus 99] and was first introduced by Downey et al. [Downey 80] to enable efficient analysis of tree structures.

Another interesting approach for modeling large execution traces is implemented in Jinsight. As we showed earlier, Jinsight uses the call-frame principle to represent cumulative information about the traces such as the number of calls a method 'A' makes to 'B' and so on. However, Jinsight's approach is more useful for performance analysis than for program comprehension and therefore it will not be considered in this study.

Finally, Trace Explorer uses a graph to represent the execution traces, where the nodes represent the objects and the arcs represent the method calls. However, this technique requires extra data structures to keep track of the order of calls, which makes traversing the graph time consuming.

2.4.2 Levels of Granularity of the Analysis

A key aspect of reverse engineering is to extract different levels of abstraction of a software system. Depending on the tool used, one can view the content of an execution trace at the following levels of granularity:

- **Object level:** This level is concerned with visualizing method calls among objects and is supported by most existing tools. This can be useful for detecting memory leaks and other performance bottlenecks.

- **Class level:** In this level, objects of the same class are substituted with the name of their classes. This level suits best activities that require high-level understanding of the system's behaviour such as recovering the documentation, understanding which classes implement a particular feature, etc. This thesis focuses on this level of granularity.

- **Subsystem level:** This level consists of grouping classes into clusters and showing how the system's components interact with each other.

2.4.3 Dealing with the Large Size of Traces

A key element for a successful dynamic analysis tool consists of implementing efficient techniques for reducing the amount of information contained in traces. We classify the techniques used by these tools into two categories. The first category is concerned with the ability to browse the content of the trace easily, search for specific elements and so on. We call this category: Basic Trace Exploration. The second category is concerned with the ability to reduce the size of the trace by removing (or hiding) some of its components. We call this category of techniques: Trace Filtering.

Basic Trace Exploration techniques are tightly coupled with the visualization tools that implement them. Generally speaking, using these techniques an analyst can browse, animate, slice or search the traces. It seems that there is an agreement about the importance of such techniques in reducing the information overhead and most of the tools support these features. Basic trace exploration is also concerned with techniques that allow searching the trace content for specific components. ISVis, for example, enables the analysts to use wildcards to formulate sophisticated queries.

Trace filtering techniques operate on the execution traces independently of any visualization scheme. The goal is to hide some components of the trace in order to abstract out its content. Most tools process the traces in an off-line manner (i.e. the trace is first generated and then filtered). We found that the tools discussed earlier implement different techniques that we present below:

Data Collection Techniques:

The collection of trace data can be done either at the system level or at the level of selected components. These two approaches have their advantages and disadvantages. The advantage of the system-level data collection approach is that the analyst does not need to know which components implement the feature under study. However, the resulting execution traces are usually quite large and require advanced filtering techniques. The component-level data collection technique has the obvious advantage of resulting in smaller execution traces but requires from the analyst to know, in advance, which components need to be instrumented. Shimba, for example, involves the analyst in the process of detecting the components that implement the desired feature. We do not think that this is practical for complex features. There is a need for feature localization techniques such as the ones described by Wilde et al. [Wilde 95] and Eisenbarth et al. [Eisenbarth 01].

In this thesis, we do not make any assumption with respect to the level of expertise a software engineer has of the system. The techniques presented in this thesis can apply to any traces independently of the number of components involved in generating them.

Pattern Matching:

Most of the tools use pattern detection abilities to group similar sequences of events in the form of execution patterns so as the analyst does not need to look at the same sequence twice. The concept of execution patterns has been given various names including behavioural patterns in Shimba, interaction patterns in ISVis, and collaboration patterns in The Collaboration Browser.

Patterns are efficient at reducing the size of traces if they are generalized [De Pauw 98]. For this purpose, a variety of matching criteria are suggested such as the ones implemented in Ovation and The Collaboration Browser. Some matching criteria require the setting of parameters. For example, the depth-limiting criterion presented in Section 2.3.3 involves setting the depth at which two sequences of events need to be compared. The challenge is to find the appropriate settings to achieve an understanding of the feature under study. Furthermore, the different combinations of matching criteria will result in different filtering of the content of trace, which poses real challenges to using these criteria in practice. A more discussion about matching criteria is presented in Chapter 3.

Hiding Components:

Removing specific components from traces is another way to reduce the trace size. For example, the analyst may simply decide to hide all the invocations of a specific method. Most of the tools implement capabilities for removing information from the trace. The Collaboration Explorer and Program Explorer, for example, allow the analyst to remove methods, specific objects or even classes. Pruning and slicing are two concepts used in Program Explorer that achieve this. However, it is totally up to the maintainer to uncover the components that can be filtered out without affecting the comprehension of the trace. There is certainly a need for automatic assistance. In this thesis, we argue that not all routines invoked in a trace are important in order to understand its content. Some routines are mere implementation details and obscure the content of traces. One of the main contributions of this thesis is a detailed definition of the concept of utilities. We have also developed a technique for automatic detection of utilities based on fan-in analysis of the system components (see Section 4.4).

Sampling:

Sampling is an interesting way of reducing the size of the trace and was used in AVID. It is concerned with choosing only a sample of the trace for analysis instead of the whole trace. However, finding the right sampling parameters is not an easy task and even if some parameter settings work for understanding one feature, it is not evident that the same settings will work for another feature. In this thesis, we do not deal with sampling techniques.

Architectural-level filtering:

Another approach for reducing the size of traces is to show the dynamic interactions among the architectural components of the system rather than among single objects (or classes). For this purpose, the analyst first determines the system architecture (if it is not available) and then the execution trace is abstracted out to show the interactions among the components. However, this approach can help software engineers understand the system at the architectural level only. In addition to this, it requires the system architecture to be present. AVID, for example, assumes that the analyst is familiar enough with the system to cluster classes into components. However, this is often not true in practice. Often, there is a need for automatic clustering techniques such as the ones described by Anquetil et al. [Anquetil 03], Müller et al. [Müller 93], and Tzerpos et al. [Tzerpos 98].

2.5 Summary

Table 2.1 summarizes the trace analysis techniques discussed in the previous section. It also sets out the selected tools against these techniques. This way we can assess which part of the trace analysis techniques is covered by existing tools and which part is still left open. For example, except for AVID, most existing tools do not permit the analysis of the dynamics of the system at the architectural level.

From the table we can also infer several other facts concerning the tools studied in this chapter. For example, Program Explorer and Scene do not use any filtering technique. They rely mainly on the user exploration of the trace content.

In addition to this, we can see that the most used filtering techniques are the ones based on pattern detection capabilities (they are supported by four tools out of eight). The use of a static analysis tool for selecting the components that will be traced (i.e. data collection filtering technique) is supported by Shimba only. This is probably because most existing tools do not have access to the static representation of the system. Shimba uses Rigi to achieve this goal.

Table 2.1. Classification of trace analysis tools

Trace Analysis Techniques	**Shimba**	**ISVis**	**Ovation**	**Jinsight**	**Program Explorer**	**AVID**	**Scene**	**Collaboration Browser**
1. Modeling Traces								
Tree Structures	x		x				x	x
DAG		x						
Graph with labelled edges				x	x	x		
2. Dealing with the large size of traces								
Basic Trace Exploration		x	x	x	x	x	x	x
Trace Filtering								
Data collection	x							
Pattern matching	x	x	x					x
Architectural-level filtering						x		
Sampling						x		
3. Levels of granularity								
Object		x	x	x	x		x	x
Class	x	x			x		x	x
Package / Subsystem / Cluster						x		

Chapter 3. Trace Metrics

3.1 Introduction

Since the outset of our research, our goal has been to find ways to make it easier for software engineers to understand a program's behaviour by exploring traces. This necessitates simplifying views of traces in various ways – in other words, reducing their size and complexity while keeping as much of their *essence* as possible.

However, in order to characterize and quantify the techniques we develop, we need to be able to *measure* various aspects of traces. Having suitable metrics will allow us, for example, to compare the outputs of various filtering algorithms. Such metrics might also be built into tools: One application would be to give an algorithm a goal to reach such as to shorten a trace by a certain percentage. Another application would be to use colouring or other encoding to highlight to the user the parts of a trace that are more complex, or the quantity of nodes that are hidden.

Existing tools, such as those presented in Chapter 2, provide various filtering operations, but do not give feedback to the user in a meaningful way about how much complexity has been removed by the filtering. We looked, informally, at the behaviour embedded in most interesting traces and found that they can be considerably more complex than expected; yet due to a lack of quantitative guidance, it was not obvious how to combine the filtering techniques to reduce this complexity.

At first glance, one might imagine measuring a trace could be rather straightforward: A naïve approach might just be to report the file size or the number of lines in the trace. However, a myriad of subtleties arise, for example:

- The file size and number of lines depend on the schema used to represent the trace and the syntax used to convey the data specified by such a schema (See Chapter 6 for more on this).

- Not all elements of a trace are equally important, or contribute equally to complexity. For example, a long series of identical method calls would be rather simpler to understand than a highly varied and non-repetitive sequence of the same length.

- The notion of complexity is itself rather vague, suggesting that we need to be able to measure a wide variety of aspects that may contribute to complexity so we can later experiment with various approaches to complexity reduction.

In this chapter, we first present a catalog of metrics for measuring various aspects of traces. We then report the result of applying these metrics to thirty traces generated from three software systems.

The outcome of this work can be used in different ways:

- Software engineers can use the metrics to choose which traces to analyze, to generate traces that are neither too complex nor too simple, and to select parts of traces to analyze that have a suitable complexity level.

- The designers of trace analysis tools can incorporate the metrics into tool features. They can design facilities to reduce the amount of information being displayed to some threshold (by hiding sufficient detail). Tool designers could also 'color' each subtree of a trace to give software engineers a better sense of the complexity to be found in that subtree before the software engineer 'opens' it for exploration.

- Researchers in the field of dynamic analysis (with a focus on program comprehension) can use the material presented here to help characterize the techniques they develop for reducing the complexity of traces.

This chapter is organized as follows: In the next section, we introduce the concept of *comprehension units* that will help us determine some of the metrics used in this chapter. In Section 3.3, we describe the metrics and motivate why they are important for characterizing the effort required to understand an execution trace. In Section 3.4, we show the results of analyzing traces of three software systems. In Section 3.5, we discuss how these metrics can be supported by tools. Some of the metrics presented in this chapter rely on the fact that a tree structure can be represented in the form of an ordered directed acyclic graph (DAG) [Downey 80]. We therefore, include in this chapter an algorithm that performs this transformation and present it in Section 3.6.

Much of the material in this chapter is adapted and expanded from a paper published in the 10th IEEE International Conference on Engineering of Complex Computer Systems, 2005 [Hamou-Lhadj 05b].

Initial measurements we performed on traces have been published in the 2nd ICSM International Workshop on Visualizing Software for Understanding and Analysis, 2003 [Hamou-Lhadj 03a], and the 10th IEEE International Workshop on Program Comprehension (IWPC), 2002 [Hamou-Lhadj 02].

3.2 The Concept of Comprehension Units

We define a *comprehension unit* as a distinct subtree of the trace. We hypothesize that in order to fully understand the trace, without any prior knowledge of the system, the analyst would need to investigate all the comprehension units that constitute it. It is important to notice that in practice, full comprehension would rarely be needed because the analyst will achieve his or her comprehension goals prior to achieving a full comprehension. Also, he or she will likely not need to try to understand the differences among the many comprehension units that only have slight differences.

We deliberately choose to use the term ‘comprehension unit’. Both words in this choice of terminology have been criticised, but we intend to maintain our choices. The word ‘comprehension’ is used so as to emphasise the fact that we are indeed trying to consider the distinct parts of the trace that need to be *comprehended*. Also the term ‘unit’ used to indicate that these are items that can be counted; it is clear that the amount of comprehension required to understand the internals of two comprehension units will likely differ widely, but the same would be true of other ‘units’ in software engineering such as lines of code, methods, etc. In other words, it does not matter that the amount of understanding is different, and it does not matter that there will be other things (other than simply the subtrees) to understand. The key idea is that two equal comprehension units will not need understanding more than once.

An efficient technique for extracting comprehension units is based on transforming the trace into its compact form by representing similar subtrees only once. This transformation results in an ordered directed acyclic graph (DAG) [Downey 80, Flajolet 90].

Figure 3.1 shows a trace in the form of a tree structure and its corresponding ordered directed acyclic graph. The graph shows that the trace contains six comprehension units and that the comprehension unit rooted at ‘B’ is repeated twice in a non-contiguous way. We refer to comprehension units that are repeated non-contiguously as *trace patterns*.

Many concepts presented in this thesis consider the DAG form of a trace rather than a tree structure. For this purpose, we developed an algorithm for on the fly transformation of a tree into an ordered directed acyclic graph. Using this algorithm, a trace will ever need to be saved as a tree structure. The details of this algorithm are presented in Section 3.6.

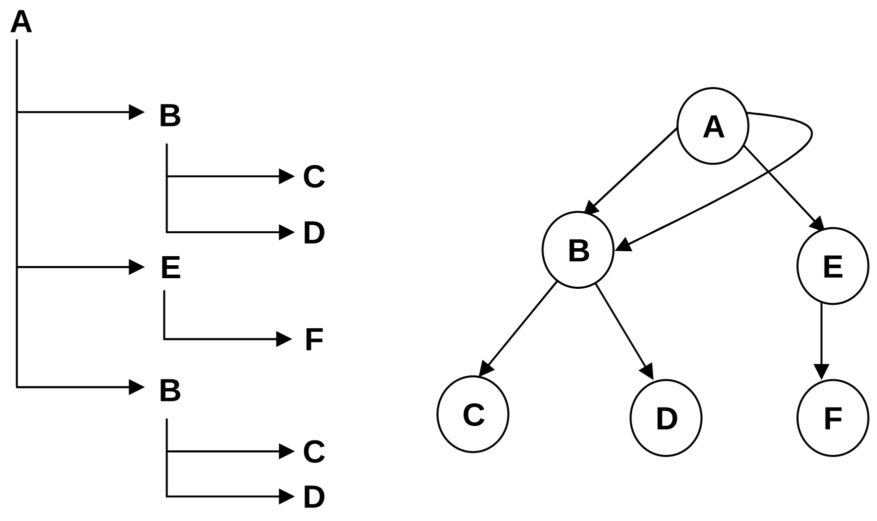


Figure 3.1. The graph representation of a trace is a better way to spot the number of distinct subtrees it contains

3.3 The Catalog of Metrics

In this subsection, we develop a set of metrics that can be used to measure potentially useful properties of execution traces. We motivate the design of these metrics using the Goals/Questions/Metrics (GQM) model [Basili 94] as a framework.

The top level goal can be stated as follows:

> *To enable software engineers to more quickly understand the behaviour of a running system.*

To achieve this goal, and in accordance with the GQM approach, we have designed questions that aim at characterizing the effort required for understanding an execution trace. We present these questions along with the metrics that address them.

Software engineers will only benefit from these metrics if they are incorporated into tools, in ways such as those suggested earlier.

The metrics we present are designed to explore the possible space of size and complexity metrics. Not all of them will necessarily be of equal value. Later in the chapter we will present case studies where we use some of them to actually measure some traces. We will also use the metrics in discussions later in the thesis.

3.3.1 Call Volume Metrics

This category of metrics aim to answer the following question: How many calls does a trace contain?

Knowing the number of calls in a trace is one factor that will help determine the work required to understand the trace. The following metrics make this more precise:

Full size [S]: The full size is the raw number of calls recorded in the trace; it is the number of nodes in the call tree without any of the manipulations described below, such as removing repetitions. This forms a baseline for subsequent computations and will always be numerically larger than any of the other size metrics described below.

Note that we say that it is the number of calls *recorded*. It is possible that some or many calls are not recorded. For example, a software engineer will often choose to not record invocations of private methods. He or she may also sample execution during a certain time period only, or record only invocations in a certain subsystem. S is therefore always relative to the instrumentation used to gather the trace, but nevertheless represents the size of everything the software engineer has to work with in a given trace.

Size after removing contiguous repetitions [Scr]: This is the number of lines in the trace after removing contiguous repetitions due to loops and recursion. In other words, many identical calls are mapped into a single line by processing the trace. We refer to this process as the repetition removal stage.

Note that by identical, we are referring to identity between subtrees. At the leaf level of the call tree, the sequences AAABBB and AABBBB would result in Scr=2 (i.e. the nodes AB), whereas AABBBAA would result in Scr=3 (nodes ABA). And one level higher, $A_{CCDD}A_{CD}B_D$[3] and $A_{CDDD}B_{DDD}$ would both result in Scr=5 five (nodes $A_{CD}B_D$), whereas $A_{CCD}B_{DD}A_{CD}$ would result in Scr=8 (nodes $A_{CD}B_DA_{CD}$).

According to our experience working with many execution traces, it seems that the number of lines after repetition removal is a much better indicator of the amount of work that will be required to fully understand the trace. This is due to the fact that the full size of a trace, S, is highly sensitive to the volume of input data used during the execution of the system. To understand the behaviour of an algorithm by analyzing its traces, it is often just as effective to study the trace after most of the repetitions are collapsed. And, in that case, studying a trace of execution over a very large data set would end up being similar to studying a trace of execution over a moderately sized data set.

Size, treating all called routines as a set [Sset]: This is the number of lines that remains after all repetitions and ordering are ignored. So for example $A_{CCDD}A_{CD}B_D$, $A_{CDDD}B_{DDD}$ and $A_{CCD}B_{DD}A_{CD}$ would all result in Sset=5 (nodes $A_{CD}B_D$).

Collapse ratio after removing contiguous repetitions [Rcr]: This is the ratio of the number of nodes after removing the contiguous repetitions from the full trace. Rcr = Scr/S.

[3] We use the notation A_C to represent 'A calls C'

Knowing that a program does a very high proportion of repetitive work (that Rcr is low) might lead program understanders to study possible optimizations, or to reduce input size. Knowing that Rcr is high would suggest that understanding the trace fully will be time consuming.

The Collapse Ratio is analogous to the notion of ‘compression ratio’ used in the context of data compression. However we have carefully avoided using the term ‘compression’ since it causes confusion: The purpose of compression algorithms is to make data as small as possible; a decompression process is required to reconstitute the data in order to use it for any purpose. On the other hand, the purpose of collapsing is to make the data somewhat smaller by eliminating unneeded data, with the intent being that the result will be intelligible and useful without the need for ‘uncollapsing’.

Collapse ratio treating calls as a set [Rset]: Analogously to the above, this is Sset/S.

3.3.2 Component Volume Metrics

This category of metrics is concerned with measuring the number of distinct components that are invoked during the execution of a particular scenario. The motivation behind the design of these metrics is that traces that cross-cut many different components of the system are likely to be harder to understand than traces involving fewer components. In addition to this, it is very common that software engineers exploring traces map the trace components to their implementation counterpart in order to better understand a particular functionality. Knowing that a trace involves a large number of the system components (e.g. classes, packages, etc) would suggest that tools ought to support techniques that would easily allow this mapping.

More specifically, these metrics aim to investigate the following question: How many system components are invoked in a given trace?

The term ‘component’ is very general. Separate metrics can be used to measure invocations of different types of components. In this chapter, since the target systems that are analyzed are all programmed in Java then it may be useful to measure the following:

Number of packages [Np]: This is the number of distinct packages invoked in the trace. By ‘invoking’ a package we mean that some method in the package is called.

Number of classes [Nc]: This is the number of distinct classes invoked in the trace.

Number of methods [Nm]: This is the number of distinct methods that appear in the trace (i.e. irrespective of how many times each method is called).

It may be useful to create similar metrics based on other types of components, e.g. the number of threads involved.

The following ratios enable one to determine the proportion of a system that is covered by the trace. The more of a system covered, the more time potentially required to understand the trace, but the more complete an understanding of the entire system may be gained. Thus we may measure the following:

Ratio of number of trace packages to the number of system packages [Rpsp]: This is the ratio of the number of packages invoked in a trace to the number of packages of the instrumented system. More formally, if we let:

- NSp = The number of packages of the instrumented system
- Np = The number of packages as defined above

Then: Rpsp = Np/NSp

Ratio of number of trace classes to the number of system classes [Rcsc]: This is the ratio of the number of classes invoked in a trace to the number of classes of the instrumented system. This is computed analogously to Rpsp.

Ratio of number of trace methods to the number of system methods [Rmsm]: This is the ratio of the number of methods invoked in a trace to the number of methods of the instrumented system.

3.3.3 Comprehension Unit Volume Metrics

The comprehension unit volume category of metrics is computed from the graph representation of the trace. The question that we are interested in investigating is: How many comprehension units exist in a trace?

We suggest that metrics based on comprehension units will give a more realistic indication than call volume of the complexity of a trace.

Number of comprehension units [Scu_{sim}]: This family of metrics represents the number of comprehension units (i.e. distinct subtrees) of a trace. The number may vary depending on the way similarity among the subtrees is computed. The subscript 'sim' is used to refer to the similarity function used.

Considering exact matches only will result in a maximum number of comprehension units, Scu_{exact}. For example, consider the tree of Figure 3.2; the two subtrees rooted at 'B' differ because the number of contiguous repetitions of their subtrees differs.

If the number of contiguous repetitions is ignored when comparing two subtrees we use the subscript cr, and the resulting metric is Scu_{cr}, which will always be equal to or less than Scu_{exact}. We will define Scu (unsubscripted) to mean Scu_{cr}, since in our experience, Scu_{cr} is a more useful basic measure of comprehension units than Scu_{exact}.

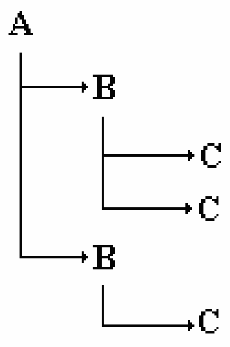


Figure 3.2. The two subtrees rooted at 'B' can be considered similar if the number of repetitions of 'C' is ignored

We can also compute Scu_{set} which compares the set of subtrees, ignoring all repetition (contiguous or not) and also ignoring order. It will be true that $Scu_{set} \leq Scu_{cr} \leq Scu_{exact}$. Further metrics (with other subscripts) can be computed by further varying the matching criteria.

Graph to tree ratio [Rgt_{sim}]: This family of metrics measures the ratio of the number of nodes of the ordered directed acyclic graph to the number of nodes of the trace. We expect to find a very low ratio since the acyclic graph factors out repetitions. More formally, let us consider:

- Scr = The size of a trace T after removing contiguous repetitions.
- Scu_{cr} = The size of the resulting graph after transforming T to a graph.

Then: $Rgt_{cr} = Scu_{cr}/Scr$.

Using another similarity function, we can also have $Rgt_{set} = Scu_{set}/Sset$.

If Rgt_{sim} is very low then this suggests that even a huge original trace might be relatively easy to understand.

3.3.4 Pattern Related Metrics

This category of metrics is concerned with measuring the number of patterns that exist in a trace. As a reminder, a pattern is a comprehension unit that is repeated in a trace non-contiguously. As we showed in Section 2.3, most trace analysis tools rely on pattern detection capabilities to help software engineers explore the content of large traces. In addition to this, Jerding et al. have shown that patterns play an important role in uncovering domain concepts or interesting pieces of computation that the software engineer would benefit from understanding [Jerding 97a, Jerding 97b]. In this category, we therefore investigate the following question: How many patterns exist in a trace?

We present the following metrics:

Number of trace patterns [$Nptt_{sim}$]: This metric simply computes the number of trace patterns that are contained in a trace, given 'sim' as the similarity function (as before, if it is omitted, we assume cr). If Nptt is small this suggests that the complexity of the trace will be high.

Ratio of the number of patterns to the number of comprehension units [Rpcu]: This metric computes the ratio of the number of patterns to the number of comprehension units. In other words, we want to assess the percentage of comprehension units that are also patterns.

3.3.5 Discussion

This section discusses the *necessity* and *sufficiency* of the four families of metrics presented in this chapter. In general, the metric families are necessary since they each tell a different story about a trace: Leaving one or other family out would leave gaps. On the other hand, sufficiency of a metric is normally evaluated on the basis of whether the metric captures an underlying concept (in cases where the metric is a surrogate for an underlying concept). The metrics described here are generally very direct measures, as opposed to surrogates, so sufficiency is less of an issue.

The call volume metrics can be used to indicate the complexity of a trace by simply looking at its length (i.e. the number of calls). These metrics also take into account the length of the trace after removing contiguous repetition (Scr) as well ignoring the order of calls (Sset). The advantage of these metrics is that they do not require extensive processing. Their limitation is that they do not consider the non-contiguous repetitions that occur in a trace. In other words, two identical subrees that occur in a non contiguous way will be counted twice. To address this issue, we presented the comprehension unit metrics (Scu, Rgt), which measure the content of a trace by factoring out all kinds of repetitions whether they are contiguous or not. We believe that these metrics are a better indicator of the work required to explore and understand the content of a trace since software engineers do not need to understand the same comprehension unit twice.

Trace patterns metrics are necessary in order to evaluate the number of patterns in a trace. As shown by Jerding et al. [Jerding 97a, Jerding 97b], software engineers often rely on exploring trace patterns so as to uncover the core behaviour embedded in a trace. These metrics can be used in combination with the comprehension unit metrics so as to compute the ratio of the number of patterns to the number of comprehension units. This ratio can be used to assess the extent to which a trace performs repetitive work.

Finally, we presented the component volume metrics to assess the number of the system's components invoked in a trace. These metrics are necessary given that software engineers will most likely need to map the content of a trace to the source code in order to get more information. Knowing that a trace, or part of trace, cross-cuts several system components is an indicator that this mapping might be complex to perform. These metrics can be used independently from the other metrics presented in this chapter.

3.4 Case Studies

We analyzed the execution traces of three Java software systems: Checkstyle [Checkstyle], Toad [Toad] and Weka [Weka, Witten 99]. This analysis has the following objectives:

- Compute a wide variety of the metrics presented in the previous section
- Perform further analysis to interpret the measurements using the metrics
- Draw inferences about the applicability of the metrics by comparing the results from the three systems

3.4.1 Target Systems

To begin with, we briefly describe the three target systems used in this study. Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard [Checkstyle]. This is very useful to projects that want to enforce a coding standard. The tool allows programmers to create XML-based files to represent almost any coding standard. Toad is an IBM tool that includes a large variety of static analysis tools for monitoring, analyzing and understanding Java programs [Toad]. Although these tools can be run as standalone tools, they can provide a much greater understanding of a Java application if they are used together. Weka is a collection of machine learning algorithms for data mining tasks [Weka, Witten 99]. Weka contains tools for data pre-processing, classification, regression, clustering and generating association rules.

Table 3.1. Characteristics of the target systems

	Packages	Classes	Non-private Methods	KLOC
Checkstyle	43	671	5827	258
Toad	68	885	5082	203
Weka	10	147	1642	95

Table 3.1 summarizes certain static characteristics of the target systems relevant to computing the trace metrics. For simplicity, we deliberately ignore the number of private methods (including private constructors) both in the table and in the traces: they are used to implement behaviour that will be localized within a class, so tracing them would provide minimal value in terms of comprehending the system, when compared to the added cost of a much larger trace. Abstract methods are also excluded since they have no presence at run time. Finally, the number of classes does not include abstract classes for the same reason.

3.4.2 Generating Traces

We used our own instrumentation tool based on BIT [Lee 97] to insert probes at the entry and exit points of each system's non-private methods. Constructors are treated in the same way as regular methods. Traces are generated as the system runs, and are saved in text files. Although all the target systems come with a GUI version, we can invoke their features using the command line. We favoured the command line approach over the GUI to avoid encumbering the traces with GUI components. A trace file contains the following information:

- Thread name
- Full class name (e.g. weka.core.Instance)
- Method name and
- A nesting level that maintains the order of calls

Table 3.2. Checkstyle Traces

Trace	Description
C-T1	Checks that each java file has a package.html
C-T2	Checks that there are no import statements that use the .* notation.
C-T3	Restricts the number of executable statements to a specified limit.
C-T4	Checks for the use of whitespace
C-T5	Checks that the order of modifiers conforms to the Java Language specification
C-T6	Checks for empty blocks
C-T7	Checks whether array initialization contains a trailing comma
C-T8	Checks visibility of class members
C-T9	Checks if the code contains duplicate portions of code
C-T10	Restrict the number of &&, || and ^ in an expression

Table 3.3. Toad Traces

Trace	Description
T-T1	Generates several statistics about the analyzed components
T-T2	Detects and provides reports on uses of Java features, like native code interaction and dynamic reflection invocations, etc.
T-T3	Generates statistics in html format about unreachable classes and methods, etc
T-T4	Specifies bytecode transformations that can be used to generate compressed version of the bytecode files
T-T5	Generates the inheritance hierarchy graph of the analyzed components
T-T6	Generates the call graph using rapid type analysis of the analyzed component
T-T7	Generates an html file that contains dependency relationships among class files

We noticed that all the tools use only one thread, so we ignored the thread information. We generated several traces from the execution of the target systems. The idea was to run the systems invoking several features of the system. We deliberately choose features that cover different aspects of the system and that are not slight variations of each other. This will allow us to better interpret the results. Table 3.2, 3.3 and 3.4 describe the features that have been traced for each system.

Table 3.4. Weka Traces

Trace	Description
W-T1	Cobweb Clustering algorithm
W-T2	EM Clustering algorithm
W-T3	IBk Classification algorithm
W-T4	OneRClassification algorithm
W-T5	Decision Table Classification algorithm
W-T6	J48 (C4.5) Classification algorithm
W-T7	SMO Classification algorithm
W-T8	Naïve Bayes Classification algorithm
W-T9	ZeroR Classification algorithm
W-T10	Decision Stump Classification algorithm
W-T11	Linear Regression Classification algorithm
W-T12	M5Prime Classification algorithm
W-T13	Apriori Association algorithm

3.4.3 Collecting Metrics

The collection of metrics resulted in a large set of data that we present in Table 3.5, Table 3.6, and Table 3.7. To help interpret the results, we added descriptive statistics such as the average, the maximum, and minimum.

The Call Volume Metrics

Table 3.5 shows the results of computing the call volume metrics for the Checkstyle system. The Full Size metric shows that traces are quite large even when they are triggered using a simple example as input data, which is the case in this study. As we can see in Table 3.5, the average size is around 74615 calls. The trace C-T9 is the only trace that does not follow this rule as it generates only 1013 calls. After an analysis of the content of this trace, we found that it is the only one that does not invoke methods of the "antlr" package, which is a package that seems to generate many calls in other traces. Future work should focus on analyzing how the metrics we have defined vary depending on the system components invoked in traces. The average size of the resulting traces after the repetition removal stage is 33293 calls, which is still too high for someone to completely understand.

The average size of Toad traces is 220409 calls as shown in Table 3.6, which is almost three times higher than the average size of Checkstyle traces. The collapse ratio after removing contiguous repetitions, Rcr, is around 5% which is much lower than Rcr for Checkstyle (46%). The average size of the resulting traces is around 10763 calls.

Table 3.5. CheckStyle Statistics

Checkstyle	**Call Volume Metrics**			**Component Volume Metrics**						**Comprehension Units and Patterns Metrics**			
Traces	**S**	**Scr**	**Rcr**	**Np**	**Rpsp**	**Nc**	**Rcsc**	**Nm**	**Rmsm**	**Scu_{cr}**	**Rgt_{cr}**	**Nptt**	**Rpcu**
C-T1	84040	37957	45%	14	33%	114	17%	590	10%	1261	3%	515	41%
C-T2	81052	35969	44%	13	30%	109	16%	540	9%	1106	3%	423	38%
C-T3	81639	36123	44%	13	30%	110	16%	561	10%	1153	3%	439	38%
C-T4	84299	37062	44%	14	33%	117	17%	590	10%	1191	3%	464	39%
C-T5	80393	35455	44%	13	30%	106	16%	547	9%	1098	3%	424	39%
C-T6	81550	36087	44%	14	33%	113	17%	562	10%	1125	3%	437	39%
C-T7	89085	41414	46%	14	33%	148	22%	700	12%	1455	4%	532	37%
C-T8	83106	37163	45%	14	33%	114	17%	586	10%	1234	3%	490	40%
C-T9	1013	618	61%	9	21%	70	10%	276	5%	306	50%	27	9%
C-T10	79969	35083	44%	13	30%	105	16%	521	9%	1071	3%	406	38%
Max	**89085**	**41414**	**61%**	**14**	**33%**	**148**	**22%**	**700**	**12%**	**1455**	**50%**	**532**	**41%**
Min	**1013**	**618**	**44%**	**9**	**21%**	**70**	**10%**	**276**	**5%**	**306**	**3%**	**27**	**9%**
Average	**74615**	**33293**	**46%**	**13**	**31%**	**111**	**16%**	**547**	**9%**	**1100**	**8%**	**416**	**36%**

Table 3.6. Toad Statistics

Toad	**Call Volume Metrics**			**Component Volume Metrics**						**Comprehension Units and Patterns Metrics**			
Traces	**S**	**Scr**	**Rcr**	**Np**	**Rpsp**	**Nc**	**Rcsc**	**Nm**	**Rmsm**	**Scu_{cr}**	**Rgt_{cr}**	**Nptt**	**Rpcu**
T-T1	219507	10409	5%	20	29%	172	19%	615	12%	827	8%	293	35%
T-T2	218867	10141	5%	20	29%	169	19%	592	12%	794	8%	282	36%
T-T3	226026	13132	6%	20	29%	191	22%	704	14%	971	7%	347	36%
T-T4	220438	10811	5%	20	29%	177	20%	626	12%	835	8%	299	36%
T-T5	218681	10002	5%	20	29%	164	19%	558	11%	754	8%	271	36%
T-T6	219171	10394	5%	20	29%	170	19%	605	12%	816	8%	296	36%
T-T7	220170	10450	5%	20	29%	165	19%	568	11%	782	7%	288	37%
Max	**226026**	**13132**	**6%**	**20**	**29%**	**191**	**22%**	**704**	**14%**	**971**	**8%**	**347**	**37%**
Min	**218681**	**10002**	**5%**	**20**	**29%**	**164**	**19%**	**558**	**11%**	**754**	**7%**	**271**	**35%**
Average	**220409**	**10763**	**5%**	**20**	**29%**	**173**	**20%**	**610**	**12%**	**826**	**8%**	**297**	**36%**

Although the full size (S) of most traces of the Toad system is considerably greater than the full size of Checkstyle traces, the removal of contiguous repetitions indicated by Scr suggest that the Checkstyle traces might require more time to explore than Toad traces.

The average size of Weka traces is around 145985 calls. Some Weka algorithms generate much smaller traces such as ZeroR (Trace W-T9). The differences in the size of traces as shown in Table 3.7 may be due to the complexity of the different data mining algorithms supported by Weka. The average size of the resulting traces after repetition removal is around 16147 calls.

Table 3.7. Weka Statistics

Weka	Call Volume Metrics			Component Volume Metrics						Comprehension Units and Patterns Metrics			
Trace	S	Scr	Rcr	Np	Rpsp	Nc	Rcsc	Nm	Rmsm	Scu_{cr}	Rgt_{cr}	Nptt	Rpcu
W-T1	193165	4081	2%	2	20%	10	7%	75	5%	89	2%	28	31%
W-T2	66645	6747	10%	3	30%	10	7%	64	4%	66	1%	15	23%
W-T3	39049	7760	20%	2	20%	12	8%	114	7%	177	2%	39	22%
W-T4	28139	4914	17%	2	20%	10	7%	116	7%	209	4%	49	23%
W-T5	157382	26714	17%	3	30%	19	13%	188	11%	309	1%	120	39%
W-T6	97413	25722	26%	3	30%	23	16%	181	11%	375	1%	137	37%
W-T7	283980	21524	8%	3	30%	15	10%	131	8%	168	1%	76	45%
W-T8	37095	6700	18%	3	30%	13	9%	114	7%	167	2%	41	25%
W-T9	12395	637	5%	2	20%	10	7%	93	6%	96	15%	29	30%
W-T10	43681	6427	15%	2	20%	10	7%	97	6%	131	2%	35	27%
W-T11	403704	34447	9%	4	40%	16	11%	147	9%	220	1%	78	35%
W-T12	378344	54871	15%	5	50%	26	18%	194	12%	637	1%	301	47%
W-T13	156814	9368	6%	2	20%	9	6%	72	4%	134	1%	51	38%
Max	**403704**	**54871**	**26%**	**5**	**50%**	**26**	**18%**	**194**	**12%**	**637**	**15%**	**301**	**47%**
Min	**12395**	**637**	**2%**	**2**	**20%**	**9**	**6%**	**64**	**4%**	**66**	**1%**	**15**	**22%**
Average	**145985**	**16147**	**13%**	**3**	**28%**	**14**	**10%**	**122**	**7%**	**214**	**3%**	**79**	**32%**

Figure 3.3 illustrates the average initial size and the average size after removing contiguous repetitions for traces of the three systems. Although, removal of contiguous repetitions can result in a considerable reduction of the number of calls, the resulting traces of all three systems continue to contain thousands of calls, which is still high for the users of trace analysis tools. Therefore, repetition removal is necessary but far from sufficient.

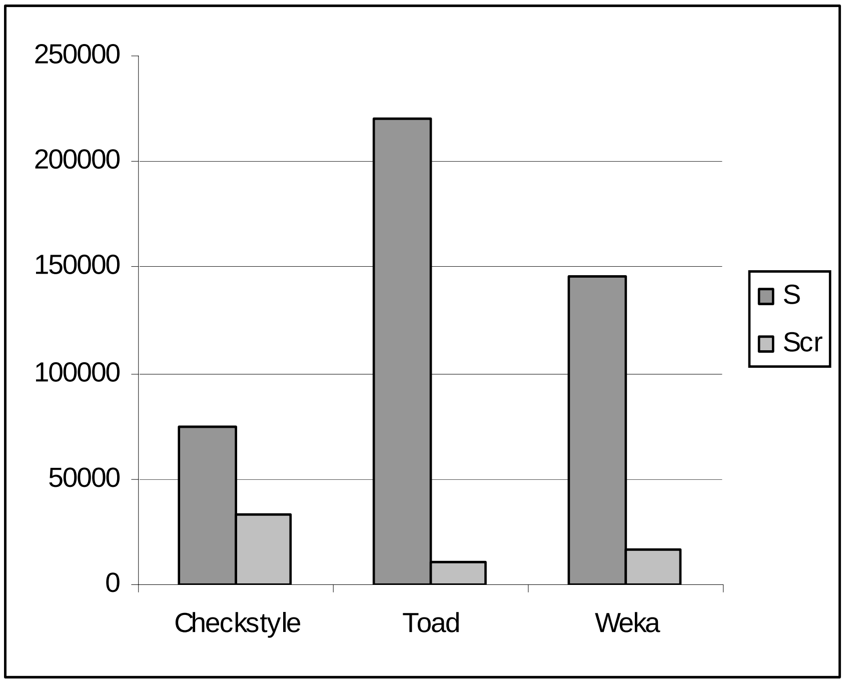


Figure 3.3. The initial size of traces and their size after the repetition removal stage for the three systems

The Component Volume Metrics

Table 3.5 shows that Checkstyle traces involve on average 31% of the system's packages; 16% of the system's classes and 9% of the system's methods.

Table 3.6 and Table 3.7 show that the other two systems have similar characteristics: The Toad traces involve on average 29% of the system's packages, 20% of the classes, and 12% of the methods; while Weka traces involve 28% of the packages, 10% of the classes and 7% of the methods. Figure 3.4 illustrates these results graphically.

However, the above results do not indicate the components of the system that contribute the most to the length of traces. In order to understand this relationship, we decided to analyse which packages constitute a high percentage of invocations in all traces of the above three systems. We did not attempt to replicate this analysis using classes and methods for simplicity reasons.

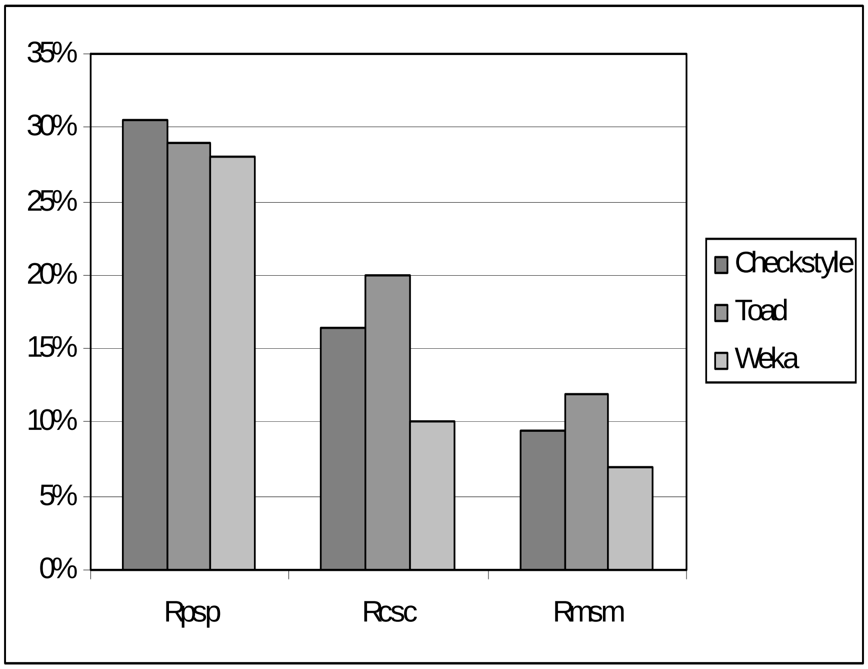


Figure 3.4. The number of components (packages, classes and methods, respectively) invoked in the traces of the three systems

Table 3.8 shows the relationship between the packages invoked in Checkstyle traces and the size of traces (we use the number of calls after removing contiguous repetitions, Scr). We notice that the package 'antlr' constitutes almost 84% of all total invocations. 'antlr' stands for ANother Tool for Language Recognition and is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions of various programming languages such as Java [ANTLR]. It is used by Checkstyle to build a representation of the Java systems that need to be processed. It is more a utility library than a package that performs actual checks for design compliance, which is the main functionality of Checkstyle.

Knowing that particular components are invoked significantly more than others can be used by tools to suggest *automatically* various strategies for rapid exploration of the content of a trace. For example, a tool can simply hide all internal invocations made to the 'antlr' package in order to reduce the amount of information displayed. Another strategy would be to use color coding to distinguish this package from the other components of the trace.

Table 3.8. The contribution of Checkstyle packages to the size of traces

Packages	**Number of invocations**	**Percentage**
com.puppycrawl.tools.checkstyle.checks.duplicates	11	0.00%
com.puppycrawl.tools.checkstyle.checks.metrics	32	0.01%
com.puppycrawl.tools.checkstyle.checks.javadoc	49	0.01%
com.puppycrawl.tools.checkstyle.checks.blocks	38	0.01%
com.puppycrawl.tools.checkstyle.checks.design	89	0.03%
com.puppycrawl.tools.checkstyle.checks.imports	90	0.03%
com.puppycrawl.tools.checkstyle.checks.whitespace	99	0.03%
com.puppycrawl.tools.checkstyle.checks.sizes	122	0.04%
com.puppycrawl.tools.checkstyle.checks	175	0.05%
org.apache.commons.logging	325	0.10%
com.puppycrawl.tools.checkstyle.checks.coding	549	0.16%
org.apache.commons.cli	640	0.19%
org.apache.commons.beanutils.converters	941	0.28%
org.apache.commons.logging.impl	1236	0.37%
org.apache.commons.collections	1779	0.53%
org.apache.regexp	3014	0.91%
org.apache.commons.beanutils	4420	1.33%
antlr.collections.impl	4536	1.36%
com.puppycrawl.tools.checkstyle	6408	1.92%
com.puppycrawl.tools.checkstyle.grammars	11430	3.43%
com.puppycrawl.tools.checkstyle.api	19087	5.73%
antlr	277861	83.46%
Total	332931	100%

Table 3.9 shows the result of performing the same analysis to traces of the Toad system. We notice that the packages ‘com.ibm.toad.cfparse’ and ‘com.ibm.toad.utils’ contribute to more than 63% of the size of the trace (Scr). The cfparse package stands for Class File Parser and is used by all Toad features to parse Java class files. The package com.ibm.toad.utils as indicated by its name groups system-scope utilities used by the various components of Toad. These two packages are clearly low-level implementation details that encumber the content of traces.

Finally, Table 3.10 shows that the package ‘weka.core’ generates almost 87% of total invocations. This package implements general purpose operations for manipulating datasets used in the implementation of the various data mining algorithms of Weka.

Table 3.9. The contribution of Toad packages to the size of traces

Packages	Number of invocations	Percentage
com.ibm.toad.jan.construction.builders.ehgbuilder	70	0.09%
com.ibm.toad.jan.lib	308	0.41%
com.ibm.toad.analyzer	536	0.71%
com.ibm.toad.jan.construction.builders.rgimpl	1061	1.41%
com.ibm.toad.cfparse.utils	1099	1.46%
com.ibm.toad.jan.lib.cgutils	1239	1.64%
com.ibm.toad.jan.lib.rgutils	1274	1.69%
com.ibm.toad.jan.jbc.utils	1309	1.74%
com.ibm.toad.jan.construction.builders.cgbuilder	1309	1.74%
com.ibm.toad.jan.construction.builders.cgbuilder.cgimpl	1333	1.77%
com.ibm.toad.jan.construction.builders.javainfoimpl	1386	1.84%
com.ibm.toad.jan.construction.builders.hgimpl	1383	1.84%
com.ibm.toad.jan.construction.builders	1589	2.11%
com.ibm.toad.jan.construction	1712	2.27%
com.ibm.toad.cfparse.attributes	2303	3.06%
com.ibm.toad.jan.lib.hgutils	2650	3.52%
com.ibm.toad.jan.coreapi	2915	3.87%
com.ibm.toad.utils	22816	30.28%
com.ibm.toad.cfparse	25501	33.85%
com.ibm.toad.jan.jbc	3546	4.71%
Total	75339	100%

Table 3.10. The contribution of Weka packages to the size of traces

Packages	Number of invocations	Percentage
weka.clusterers	463	0.22%
weka.estimators	1101	0.52%
weka.associations	1605	0.76%
weka.filters	3014	1.44%
weka.classifiers	6380	3.04%
weka.classifiers.j48	7256	3.46%
weka.classifiers.m5	8612	4.10%
weka.core	181481	86.46%
Total	209912	100%

<u>Comprehension Units Metrics</u>

Table 3.5 shows the number of comprehension units (Scu_{cr}) and the ratio achieved by transforming Checkstyle traces into acyclic graphs (Rgt_{cr}). All Checkstyle traces score a ratio of less than 4% except Trace C-T9 that scores 50%. This is due to the fact that initially this trace is significantly smaller than the other traces in terms of the number of calls but still contains a large number of methods, and that a lower bound on the number of comprehension units is the number of distinct methods in the trace. Toad traces exhibit a very low ratio as well, which is around 8%. Weka traces exhibit an even lower ratio of 3%. These results demonstrate the effectiveness of transforming the traces into ordered directed acyclic graphs. In fact, a scalable trace analysis tool should never represent traces as tree structures. In Chapter 6, we will use the graph representation of a trace to build CTF, the exchange format for representing traces.

Furthermore, a closer look at the above results reveals that the number of distinct methods (Nm) invoked in the traces of all three systems is significantly smaller than the number of calls generated, Scr, even after removal of contiguous repetitions. For example, the Checkstyle trace C-T1 invokes 590 methods but generates 37957 calls. This means that

1. Either traces contain several sequences of calls that are repeated in a non-contiguous way and/or

2. There exist several comprehension units that are similar but not exactly identical.

The first point emphasises the fact that a better estimation of the size of traces should not only rely on collapsing contiguous repetitions but also factoring out the non-contiguous ones justifying the importance of having metrics based on the number of comprehension units.

The second point reiterates the necessity to have tools implement various matching criteria (i.e. vary the 'sim' function) in order to reduce the number of comprehension units. In Section 3.6.4, we will discuss the implementation of matching criteria that can be applied while the trace is being generated. However, using matching criteria comes with a very challenging set of research questions. One of the issues is that it is difficult to predict how these criteria should be combined. Another problem is that it is not really clear how various combinations can help with specific maintenance tasks (e.g. debugging, adding a new feature, etc).

In this thesis, we do not attempt to address these issues. Our approach for reducing the amount of information contained in a trace (hence the number of comprehension units) is based on the fact that some components of the trace encumber the trace without adding any value to its main content. For example, the 'com.ibm.toad.utils' package is a clear example of such components. More details about our approach are presented in Chapter 4.

Finally we want to note that the number of comprehension units (Scu) and the number of methods (Nm) usually correlates as one can expect. Figure 3.5 illustrates the correlation between the number of methods (Nm) in the trace and the number of comprehension units (Scu) for the Checkstyle system. The graph shows, as expected, that the number of comprehension units (i.e distinct subtrees) of traces depends on the number of methods that are invoked. This also applies to Toad and Weka as illustrated in Figure 3.6 and Figure 3.7. Interestingly, the dependency in Weka is somewhat more variable than the other two systems, suggesting that some traces of Weka, and hence the corresponding pieces of functionality are particularly complex.

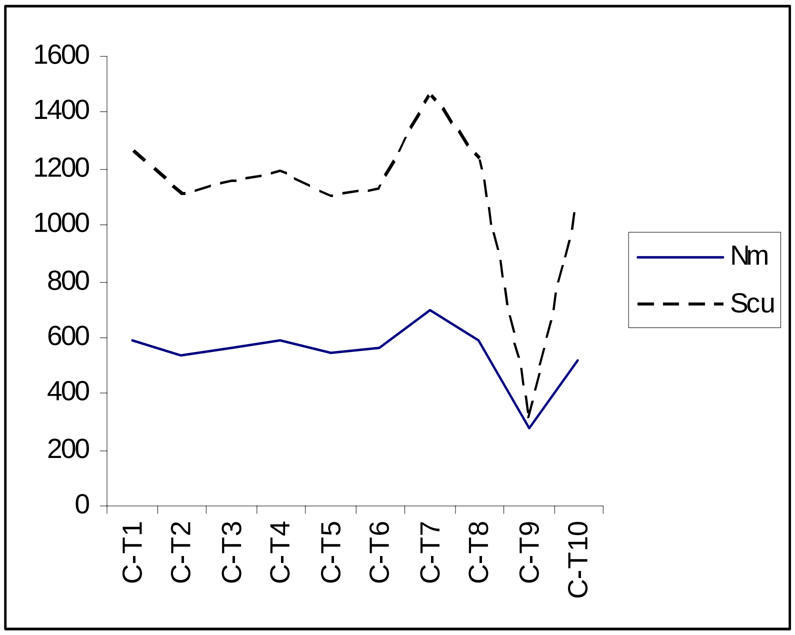


Figure 3.5. Relationship between the number of methods and the number of comprehension units for Checkstyle – Correlation coefficient = 0.99

Patterns Related Metrics

We notice that many of the comprehension units of the analyzed traces are repeated non-contiguously, which qualify them as trace patterns.

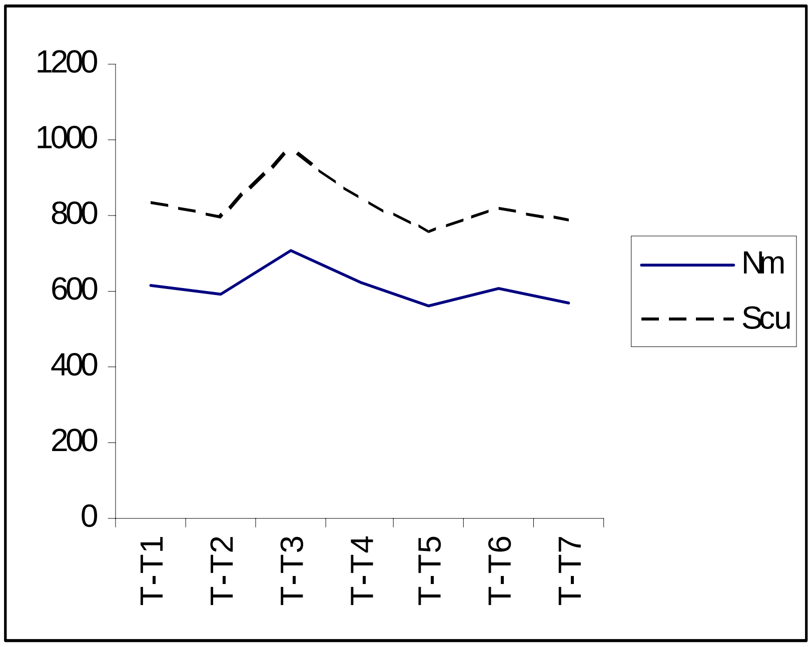


Figure 3.6. Relationship between the number of methods and the number of comprehension units for Toad – Correlation coefficient = 0.99

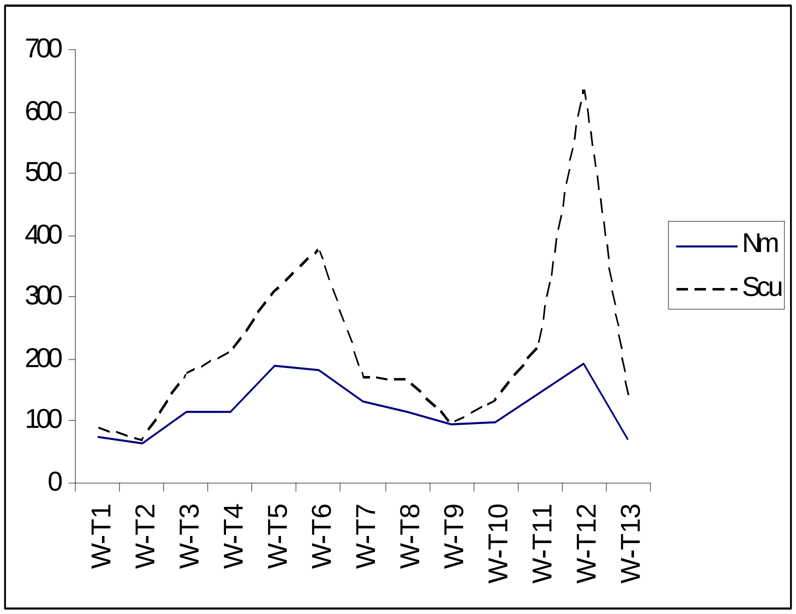


Figure 3.7. Relationship between the number of methods and the number of comprehension units for Weka – Correlation coefficient = 0.87

The average number of patterns that exist in Checkstyle traces is 416, which represents 36% of the number of comprehension units. Toad traces contain in average of 297 (36% of the number of comprehension units), and Weka traces contain on average 79 patterns (32% of the number of comprehension units).

We can see that the number of patterns can be quite high. In addition to this, it is important to note that the understanding of one pattern might also require prior understanding of other patterns.

3.5 Applying the Metrics in Tools

As discussed earlier, the goal of the metrics presented in this chapter has been to help software engineers understand traces. The metrics will, however, only be useful if actively supported by tool developers.

We suggested that tool developers could, in a rather straightforward way, simply use icons, coloring, or other graphic techniques to show the values of certain of the metrics in order to highlight information about parts of traces. However, there are a number of ways that tool developers could make deeper use of the metrics to do more sophisticated transformations of the traces.

A key objective would be to display the 'essence' of a trace – just enough to show the software engineer what happened during execution. If, in a visible portion of a trace, Scu (number of comprehension units) is much greater then Nm (number of methods), this suggests that a lot of redundant information is being displayed: Perhaps there are many somewhat similar comprehension units. In this case, we can apply techniques that change the similarity function discussed earlier so that similar comprehension units come to be treated the same, and may then also turn into patterns. Rather than seeing a lot of diversity, the software engineer might then see simply a small set of patterns. The metrics can also be used by tools to help prune the leaves and successively higher branches of traces to make what remains displayed somewhat simpler. However that will often not be enough. A tool could look at the values of ratios such as Rcr and Rgt_{sim}, for each subhierarchy, and based on the values of the ratios, contract the subtree to a certain level.

3.6 Converting a Tree Structure into a DAG

In this section, we present an algorithm for converting a tree structure into an ordered directed acyclic graph. One application of this algorithm is to help compute the family of metrics based on the concept of comprehension units (Scu, Nptt, etc).

Although the algorithm is presented in this chapter, it will also be used to generate traces in the form of CTF, the exchange format for representing traces discussed in Chapter 6.

Much of the material in this section is adapted and expanded from a paper published in the First ICSE International Workshop on Dynamic Analysis (WODA), 2003 [Hamou-Lhadj 03b].

3.6.1 Overview

The concept of transforming a rooted tree into an ordered directed acyclic graph was first introduced by Downey et al. [Downey 80]. The authors argued that the graph is a better representation since it saves storage space, and therefore allows tools that operate on trees to function efficiently. They named the problem of turning a tree into a DAG *'the common subexpression problem'* and proposed a rather complex solution for solving it.

A more practical solution was proposed by Flajolet et al. [Flajolet 90], where the authors presented a top-down recursive procedure that solves the problem in an expected linear time assuming that the degree of the tree is bounded by a constant. An iterative version of Flajolet et al's algorithm was proposed by Valiente in [Valiente 00] and was applied to solve the tree matching problem – Finding all occurrences of a subtree t in a tree T.

In this thesis, we improve Valiente's algorithm to consider two important points:

- The transformation of the tree (i.e. trace) is done while the events are generated (i.e. on the fly) in such a way that traces will never need to be saved as tree structures. The results of the Rgt metric presented in the previous sections clearly show that traces should be saved as DAGs.

- The second improvement takes into account the various matching criteria discussed in the definition of the Scu metric (i.e. the number of comprehension units). Valient's algorithm considers identical matches only.

The remainder of this section is organised as follows: the next subsection describes Valiente's solution to the common subexpression problem. In Section 3.6.3, we present our extension of Valient's algorithm. The matching criteria are discussed in Section 3.6.4.

3.6.2 Valiente's Algorithm

The algorithm presented by Valiente proceeds by traversing the tree in a bottom-up fashion (from the leaves to the root). Each node is assigned a *certificate* (a positive integer between 1 and n, where n represents the size of the tree). The certificates are assigned in such a way that two nodes *n1* and *n2* have the same certificate if and only if the subtrees rooted at *n1* and *n2* are isomorphic.

To compute the certificates, the algorithm uses a signature scheme that identifies each node. The signature of a node *n* consists of the concatenation of its label and the certificates of its direct children, if there are any. For example according to Figure 3.8, the signature of A should be "A 1 2". The signature of a leaf is simply its label (e.g. the signature of C is "C"). A global hash table is used to store the certificates and signatures and ensure that similar subtrees will always hash to the same element.

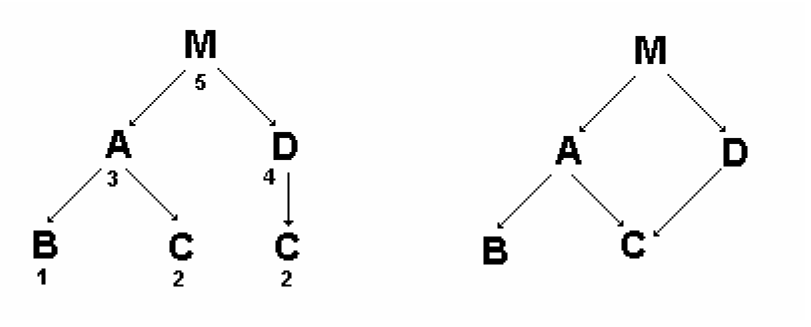


Figure 3.8. A tree (left) and its transformed DAG (right). The integers correspond to the certificates of the nodes

The global table that results from applying Valiente's algorithm to the tree of Figure 3.8 is shown in Table 3.11. From this table, we can easily construct the ordered directed acyclic graph that corresponds to the tree of Figure 3.8. The root of the graph corresponds to the table entry with the highest-numbered certificate.

Table 3.11. Result of applying Valiente's algorithm to the tree of Figure 3.8

Signature	Certificate
B	1
C	2
A 1 2	3
D 2	4
M 3 4	5

The complexity of the algorithm consists of the time it takes to traverse the tree, the time it takes to compare two subtrees, and the time it takes to compute the signatures.

3.6.3 Extension of Valiente's Algorithm to Trace Compaction

The idea behind the algorithm is to be able to detect when subtrees are constructed during the generation of the trace. Once this is done, we can compute their signatures, check if they exist in the global table, and update the global table. The final table will represent the DAG version of the trace.

To develop the algorithm, let us consider the following:

- The trace is generated as a sequence of events that have the following form: (Label, Nesting level). The label can be composed of the thread name, subsystem name, class name, object identifier, and the method name. The nesting level indicates the nesting level of the routine calls. The root of the dynamic call tree has a nesting level equal to zero. Other information can be added, such as timestamps, execution time of a routine, the number of statements, etc. For simplicity reasons, we do not include these in the algorithm.

- We will need to create a tree structure that will temporarily hold nodes corresponding to the trace events. These nodes will be destroyed once their signatures are computed.

- A node of the tree will contain the following attributes:

label	This is the label of the node.
parent	This represents the parent of the node.
nl	This represents the nesting level of the node (to avoid computing the nesting level every time we need to refer to it)
certificate	This represents the certificate that will be assigned to the node after checking the global table. Initially *certificate = 0*.

The steps of the algorithm are:

```
1. Read the first event e = (label, nl)
2. Create a node called root that represents the root of the temporary tree: root.label = e.label;
   root.parent = null; root.nl = 0; root.certificate = 0
3. prevNode = root   // Keeps track of the previous node
4. certificate = 1
5. globalTable = A Hash Table: The keys represent signatures and the values represent certificates.
                 We will use the put and get functions to insert and retrieve elements from
                 globalTable
6. For every event $e_1$ = ( $label_1$, $nl_1$ ) Do
   6.1 While ( $e_1.nl_1$ <= previousNode.nl ) Do

           /* If the current nesting level ($e_1.nl_1$) is less than or equal to the nesting level of
              the previous node which is here represented by parent.nl then the node
              prevNode must be a subtree. In fact, all the parents of the node prevNode that
              have a nesting level that is greater than or equal to $nl_1$ must also form
              subtrees. This explains why we need to use a loop to check for formed
              subtrees. */

           6.1.1 signature = Signature of  prevNode
           6.1.2 If  globalTable contains the key signature Then
                         a.  prevNode.certificate = globalTable.get (signature)
                     Else
                         b.  globalTable.put (signature, certificate)
                         c.  prevNode.certificate = certificate
                         d.  certificate++  // update the certificate
```

```
            End If
        6.1.3 prevNode = prevNode.parent
        End While
    6.2 If there are nodes for which certificates have been assigned then delete these nodes except
        the one that has the same nesting level as $nl_1$.
    6.3 Create a new node called node that represents $e_1$: node.label = $e_1$.label; node.parent =
        prevNode; node.nl = $e_1$.$nl_1$; node.certificate = 0
    6.4 prevNode = node
    End For
7.  Compute the signatures and check the table for all nodes from the last event to the root. Once
    this is done, delete all remaining nodes.
```

We illustrate the steps of the algorithm using the tree below:

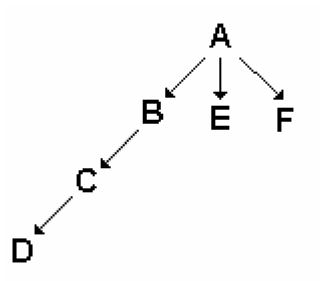
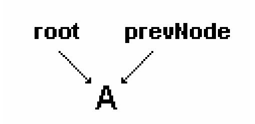
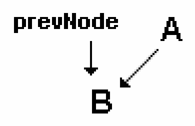
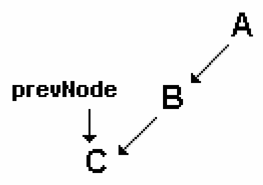
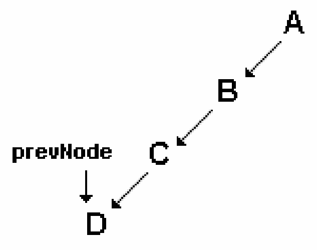


Figure 3.9. A sample tree that will be used to illustrate the algorithm

Steps of the algorithm	**Illustration**	**Global Table**
Step 1 consists of reading the first event of the trace (i.e. *e = (A, 0)*). Step 2 creates a root node that corresponds to *e = (A, 0).* Step 3 to 5 initialise variables to keep track of: - The previous node (*prevNode*) - The certificate value (*certificate*) - The global table (*globalTable*)		Signature | Certificate
In Step 6, the algorithm reads the subsequent events one by one. Step 6.1 checks if by encountering B, we		

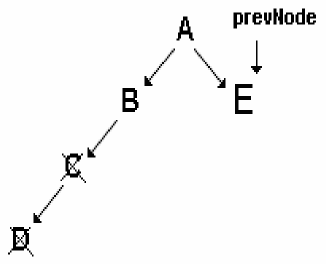
have discovered that A was the root of a complete subtree. We do this by comparing B's nesting level to that of A. Since B has a higher nesting level, the algorithm goes to Step 6.2: checks if there are nodes for which certificates have been assigned in order to clean up memory. This is not the case, 6.3 creates a node for B and links A to B. 6.4 updates prevNode.		Signature | Certificate
The algorithm continues in Step 6 to deal with the event (C, 2). It again checks whether a subtree has been formed at Step 6.1. Since this is not the case, the algorithm goes to 6.2, checks if there are nodes for which certificates have been assigned in order to clean up memory. This is not the case, so at step 6.3 it creates a node for C and links B to C. 6.4 updates prevNode.		Signature | Certificate
Same thing with the event (D, 3).		Signature | Certificate
The algorithm encounters the event (E, 1). In 6.1, the comparison reveals that the nesting level of E is less than the nesting level of D, which means than D is the root of a fully formed subtree. Therefore, the algorithm computes the signature of D		Signature | Certificat e D | 1 C 1 | 2 B 2 | 3

(which is equal to "D" in this case, since D is a leaf), checks if the signature exists in the table, if not, assigns to D the current certificate (i.e. certificate = 1), inserts the pair (signature of D, 1) into the global table, assigns the certificate to the node that holds D, and updates the value of *certificate* (steps b, c, and d).

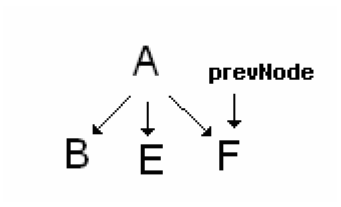
Step 6.1 causes the algorithm to repeat the process with the node C. C's signature is 'C 1' and its certificate is 2. It repeats the process again with B. B's signature is 'B 2' and its certificate is 3.

Step 6.2 deletes the nodes corresponding to the nodes D and C. However, it does not delete the node B since we will need it to compute the certificate of A.

In 6.3, the algorithm creates a node for E, links A to E and updates prevNode.



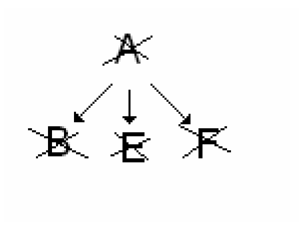
The event (F, 1) occurs causing the algorithm to compute the signature of the node E, to find that it doesn't exist in the table, to assign it a certificate, and to update the table (Steps b, c, d).



Signature	Certificate
D	1
C 1	2
B 2	3
E	4

No more events are read. The algorithm (Step 7) therefore computes the signatures of the last node all the way to the root (i.e. nodes F and A), checks whether the signatures already exist, if not assigns new certificates to these nodes, and update the table.

All remaining nodes are then deleted.



Signature	Certificate
D	1
C 1	2
B 2	3
E	4
F	5
A 3 4 5	6

The resulting table represent the ordered directed acyclic graph. The root of the graph is the entry that has the highest-numbered certificate, which is in this case the node labelled ‘A’. The links from one node to another are indicated in the signatures.

3.6.4 Matching Criteria

In this subsection, we discuss how some matching criteria can be added to the above algorithm.

A. Matching subtrees using the node labels:

One way of generalizing the algorithm is to group similar sequences of calls according to whether they invoke objects of the same class, same package, etc as proposed by Richner et al. [Richner 02] and De Pauw et al. [De Pauw 93]. To adapt the algorithm to such situations, we need to be able to select part of the node labels that will be used during the comparison process.

For example, let us consider a simple case of two leaf nodes called P1.C1.obj1.m1() and P1.C1.obj2.m1() respectively. These two nodes can be deemed similar if we decide to ignore the name of the objects (i.e. obj1 and obj2) during the computation of the signature of these nodes (i.e. Step 6.1.1 of the algorithm). The most restrictive case will be the one where all constituents of the label are compared.

B. Ignoring the number of repetitions:

Ignoring the number of repetitions of contiguous sequences of calls when looking for similar subtrees can be applied to avoid having two subtrees that differ only because some elements are repeated more than others as shown in Figure 3.10. This corresponds to the ‘cr’ subscript used to compute Scu (see Section 3.3.3).

Let us consider that the certificates of B and C are 1 and 2 respectively. In this case, the computation of the signature of A should ignore the number of times B and C are repeated in the subtree on the left side of Figure 3.10. This means that the signature of A should be equal to “A 1 2” in both cases. This implies changes in the way the signature is computed (i.e. Step 6.1.1 of the algorithm)

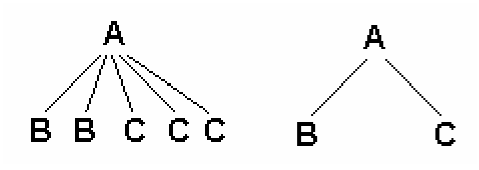


Figure 3.10. If contiguous repetitions are ignored then the above subtrees will be considered similar

C. Ignoring the order of calls:

If applied to the subtree of Figure 3.11, the ‘ignoring order of calls’ matching criterion will consider these subtrees as equivalent. The usage of this criterion corresponds to the ‘set’ subscript used to compute Scu (see Section 3.3.3).

To generalize the algorithm to unordered trees, we need to sort the certificates that appear in the signatures and then proceed with comparing the signatures. For example, if B’s certificate is 1 and that C’s certificate is 2 then A’s signature should be in both cases “A 1 2” (i.e. the certificates are sorted). Similar to the previous matching criterion, this involves changing the way the signature is computed (i.e. Step 6.1.1 of the algorithm)

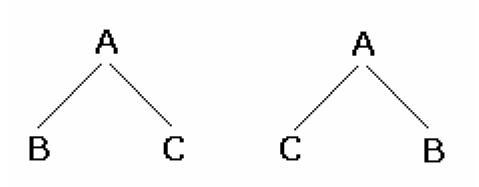


Figure 3.11. If order of calls is ignored then the above subtrees will be considered similar

D. Edit distance:

Another way of comparing two subtrees is to compute their edit distance [Tai 79]. Perhaps the easiest way to improve the algorithm to consider this metric is to modify the way it looks up for a given subtree in the global table (i.e. Step 6.1.2). This would imply that signatures can no longer be used and that subtrees need to be saved in the global table using references to these subtrees.

The application of the edit distance to two subtrees T1 and T2 of Figure 3.12 shows that these two subtrees require one edit operation (i.e. either adding the node F to T2 or removing F from T1) in order to be considered as similar. A user should have the flexibility to adjust the threshold according to his or her needs.

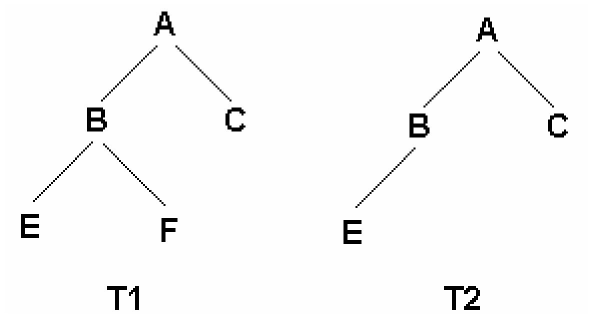


Figure 3.12. The subtrees T1 and T2 can be considered similar if an edit distance function is used

3.7 Summary

In this chapter, we developed metrics for the analysis of large execution traces of method calls. These metrics can be used by tool builders and software engineers who want to better understand traces. Tool builders can use these metrics to tune their techniques to better orient the analyst in his or her quest to understand the trace content. One possible help would be to distinguish parts of the trace that perform complex behaviour from parts that are relatively easy to grasp. Software maintainers can use these metrics to decide which traces to analyze and if there is a need to change the way the traces are collected (e.g. change the system's input) to generate less complex traces. Researchers can use these metrics to investigate further techniques for reducing the complexity of traces.

Using these metrics, we analyzed traces of three different Java systems to get a better understanding of their complexity. One of our metrics Rcr shows that when we remove contiguous repetitions, the size of the trace is reduced to between 5% and 46% of the original size. However, the resulting traces continue to have thousands of calls, which makes this basic type of collapsing necessary but not sufficient.

We found that traces might cross-cut up to 30% of the system's packages, 22% of the system's classes and 14% of the system's methods. We showed that in all systems there exist one or two packages that are the most responsible of the lengthy size of traces. Knowing this information will allow tools to adjust the amount of information displayed by either hiding the invocations made to these packages or using other visualization techniques.

In addition, we argued that the fact that the number of comprehension units (Scu) is higher than the number of distinct methods (Nm) leads us to the following observation: Tools which rely primarily on pattern detection will not allow the user to achieve an adequate level of abstraction. The problem is that there might be many subtrees in the trace that contain almost the same methods but structured in different ways. The hard part consists of selecting the appropriate matching criteria that will identify these subtrees as similar. Most tools leave this up to the users, but due to the size and complexity of traces, automated assistance is clearly needed. Tools need to suggest matching criteria that will collapse the trace to a manageable size by pre-computing the effect of each criterion. This computation can be based on the metrics described in this chapter.

Another finding regarding patterns is that traces might contain a very large number of them: over 400 in the case of the Checkstyle system. These patterns, in turn might have hundreds of occurrences, which can make the understanding of the whole trace using pattern detection a challenging task.

In the end of the chapter, we presented an algorithm for converting a trace into an ordered directed acyclic graph that can be used to compute the family of metrics related to the concept of comprehension units (Scu, Rgt, etc). Our algorithm is an extension to Valiente's algorithm by considering the on the fly generation of traces as well as various matching criteria.

Chapter 4. Trace Summarization

4.1 Introduction

Most existing trace analysis tools such as the ones presented in Section 2.3 rely on a set of fine-grained operations that software engineers can use to go from a raw sequence of events to a more understandable trace content. But due to the size and complexity of typical traces, this bottom-up approach can be difficult to perform and requires a considerable intervention of the users. In fact, when asked to describe their experience with using traces, many software engineers of the company that supports this research argued that they almost always want to be able to perform top-down analysis of a trace by having the ability to look at the *main content* first and then dig into the *details*. As previously discussed (see Section 2.2.2), many research studies in program comprehension have shown that an adequate understanding of the system artefacts necessitates both strategies (i.e. bottom-up and top-down).

In this chapter, we present the concept of trace summarization and define it as a semi-automatic process of summarizing the main content of a large trace. This is similar to text summarization where abstracts can be extracted from large documents. Abstracts are used to learn about the main facts of a document without reading entirely its content.

The primary objective of trace summarization is to extract a view of a trace that software engineers can readily work with when trying to understand the main information conveyed. We achieve this by removing from the trace the components that represent mere implementation details. However, depending on the expertise software engineers have of the system as well as the maintenance task performed, we anticipate that software engineers will need to have the flexibility to adjust the content of the summary if they find it too abstract or too detailed. When we present the trace summarization algorithm (Section 4.4.3), we will include a step where manual manipulation of the extracted summary is performed.

Trace summarization is also a powerful technique for extracting the behavioural models of a poorly documented system. Unlike the existing techniques such as the ones presented by Amyot et al. [Amyot 02] and Systä [Systä 01], our approach does not heavily rely on the intervention of users.

Indeed, Amyot et al. suggest tagging the source code at particular places in order to generate a trace that can later be represented using a use case map. This approach has the obvious drawback that it requires from the software engineers to know, in advance, where to insert the tags. It also necessitates the usage of static analysis tools. However, the authors recognised that visualizing the main flow of execution of the feature under study can help understand the system dynamics. We discussed Systä's approach in Section 2.3.1 and which involves using a static analysis tool called Rigi [Müller 88] to allow software engineers of her dynamic analysis tool to select the components of the system that need to be traced. Again, this approach requires a considerable amount of time and precludes prior knowledge of the system's static aspects.

The rest of this chapter is organized as follows: In the next section, we present the concept of trace summarization and our approach for achieving it that is based on filtering traces by removing implementation details such as utilities. In Section 4.3, we discuss in more detail the concept of utility components and how they differ from the other components of the system. The detection of utility components is the subject of Section 4.4. We defer the evaluation of our approach to Chapter 5, where we apply it to extract a summary from a trace generated from the execution of the Weka system [Weka, Witten 99].

An initial work on trace summarization has been published in the 9th IEEE European Conference on Software Maintenance and Reengineering (CSMR), 2005 [Hamou-Lhadj 05c].

4.2 What is a Trace Summary?

In general, a summary refers to an abstract representing the main points of a document while removing the details.

Jones defines a summary of a text as “a derivative of a source text condensed by selection and/or generalization on important content” [Jones 98]. Similarly, we define a summary of a trace as an abstract representation of the trace that results from removing unnecessary details by both selection and generalization.

This definition points towards several interesting questions that deserve further investigation. These are: what would be a suitable size for the summary? And how should the selection and generalization of important content be done?

4.2.1 Adequate Size of a Summary

While it is obvious that the size of a summary should be considerably smaller than the size of the source document, it seems unreasonable to declare that a summary should be of some particular fixed size. In fact, a suitable size will depend in part upon the knowledge the software engineer has of the functionality under study, the nature of the function being traced and the type of problem the trace is being used to solve (debugging, understanding, etc.). This suggests that tools should allow the summary to be dynamically expanded or contracted until it is right for the purpose at hand. We suggest that no matter how large the original trace, there will be situations when a summary of less than a page will be ideal, and there will be situations where a summary of several thousand lines may be better.

4.2.2 Content Selection

In text summarization, the selection of important content from a document is usually performed by ranking the document phrases according to their importance. Importance is measured using various techniques such as the ones based on word distribution [Edmunsdon 69, Lunh 58], cue phrases [Edmunsdon 69], and the location of phrases in the document [Baxendale 58].

When applied to execution traces, the key question is: what constitute the main content of an execution trace?

To help answer this question, we decided to investigate how execution traces are used in an industrial setting. We organized a two-hour brainstorming session with more that twenty software engineers of QNX Software Systems and asked them to discuss their experience with using execution traces (see Section 4.3 for more details). One of the results of this session is that most engineers agreed that when trying to understand the behaviour of the system, they will most likely want to hide low-level implementation details. They repeatedly referred to these low-level details as *utilities*[4]. The outcome of this study is presented in more detail in Section 4.3.

Our approach for summarizing the main content of traces is therefore based on the removal of low-level implementation details such as utilities.

Additionally, Zayour conducted a study in a company called Mitel Networks that consists of experimenting with traces of routine calls generated from one of the company's largest system. The author concluded that not all routines invoked in a trace have the same degree of importance [Zayour 02]. Some routines can be simple implementation details (e.g. sorting an array) so removing them would not affect the comprehension process very much. The author used the term 'utility routines' to refer to the least important routines. However, he only briefly discussed the concept of utilities and a way to detect them. He also did not validate whether removing utilities can indeed lead to a more comprehensible trace. One important aspect of this thesis is to validate[5] this concept. In other words, we need to validate the following hypothesis:

[4] In this thesis, however, we draw a distinction by considering utilities to be a subset of the implementation details.

[5] The validation of this hypothesis is deferred to Chapter 5 when we present the case study.

- H_1: Removing implementation details from a trace will in the opinion of software engineers reveal its main content

The null hypothesis would be:

- $H0_1$: Removing implementation details from a trace will, in the opinion of software engineers, not necessarily reveal its main content

4.2.3 Content Generalization

Content generalization consists of generalization of specific content; i.e. replacing it with more general abstract information [Jones 98]. When applied to execution traces, generalization can be performed in two ways, as follows

The first approach to generalization involves assigning a high-level description to selected sequences of events. For example, many trace analysis tools provide the users with the ability to manually select a sequence of calls and replace it with a description expressed in a natural language. However, this approach relies on user input so would not be practical to automate.

A second approach to generalization relies on treating similar sequences of events as if they were the same. This approach can be automated by varying a *similarity function* used to compare sequences of calls. In the previous chapter, we presented, for example, the ‘cr’ similarity function for treating varying numbers of contiguous nodes as the same, or the ‘set’ similarity function in which all sequences with the same elements, ignoring order, could be treated the same. Other possibilities include treating all subtrees that differ by only a certain *edit distance* as the same as shown in Section 3.6.4.

In the remainder of this thesis, we will focus on content selection, leaving content generalization for consideration as another line of research.

4.2.4 Our Approach to Trace Summarization

Figure 4.1 illustrates our approach for extracting summaries from large traces. The first three boxes in the figure show generation of a trace and the extraction of a summary by removal of implementation details. The remainder of the figure shows the iterative process of tuning the required level of detail by having software engineers assess the extracted high-level models generated from the summary.

The main steps are:

1. Generate the execution trace that corresponds to the software feature under study.
2. Filter the trace by removing low-level implementation details such as utilities. In Section 4.3, we discuss the concept of utilities in more detail. In Section 4.4, we show how fan-in analysis can be used to detect utilities.
3. Generate a summary as a result of the filtering process.
4. Convert the summary into a behavioural design model such as a UML sequence diagram to represent the extracted summary visually.
5. Validate the result with the original developers of the system.

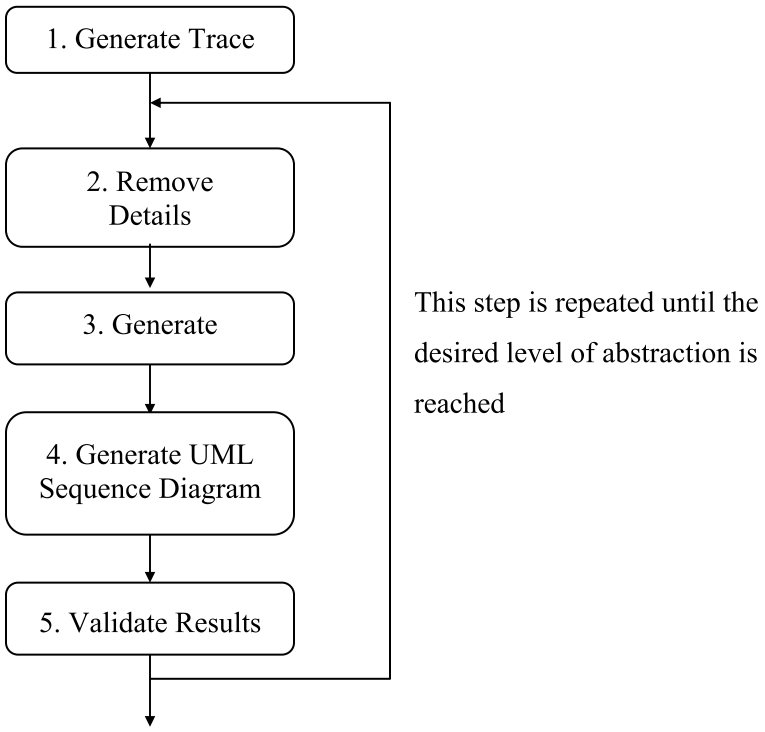

1. Generate Trace
2. Remove Details
3. Generate
4. Generate UML Sequence Diagram
5. Validate Results
This step is repeated until the desired level of abstraction is reached

Figure 4.1. Our approach to trace summarization

The validation step might lead to further filtering of the trace if the software engineers deem that the trace still contains too much detail.

It is important to note that Step 4, which involves the generation of UML sequence diagrams, is added here for visualization purposes only. Trace summarization is mainly concerned with the extraction of summaries and does not preclude any particular notation. We chose UML sequence diagrams due to the fact that a) they are widely accepted as a notation for representing the behavioural characteristics of a system, and b) the mapping of the components of traces of routine calls to UML sequence diagrams constructs is straightforward.

4.3 The Concept of Utilities

Despite the fact that the term "utility" is used frequently in practice, there has been little work in the literature that investigates this concept in more detail. Perhaps, the most interesting definition of a utility is the one cited in the UML 2.0 specification, which defines a utility as "a stereotype that groups global variables and procedures in the form of a class declaration. The utility attributes and operations become global variables and global procedures, respectively. A utility is not a fundamental modeling construct, but a programming convenience." [UML 2.0].

This definition is obviously too specific to be able to derive a complete understanding of what a utility is, since there are other kinds of utilities, such as utility methods or packages. However, it points towards several interesting aspects of utilities that deserve investigation: *global grouping, packaging, and role.* When we studied the comments made by QNX software engineers, we found that their definition of the concept of utilities also involves these aspects.

The remainder of this section starts by discussing the above aspects. A definition of the concept of utilities is presented in Section 4.3.4. However, it is important to note that we consider utilities to be a subset of a broader concept, which is the concept of implementation details. A definition of what we mean by an implementation detail as well as examples of implementation details are discussed in Section 4.3.5.

4.3.1 Global Grouping of a Utility

The global grouping of utilities as indicated in the above cited UML definition suggests that utilities are used by many other components of the system.

Along the same lines, Müller et al. [Müller 93] refer to components used by several other components of the system as *omnipresent nodes* and suggest that these components ought to be ignored in order to obtain a good understanding of the system architecture. According to the authors, omnipresent nodes tend to obscure the relationships among the system components. For this purpose, the authors compute fan-in and allow the users of their software engineering tools to filter out the components that exceed a certain 'omnipresent threshold'.

In the QNX brainstorming session, the participants also said that a key aspect of utilities is that their usage tends to be distributed around the system.

However, it is essential to realize that utilities do not always have a global scope. In a well-structured software system, the modularization constructs should form natural utility scopes. For example, in a language like Java, a package may often act as the scope for certain utilities; this means that there might be utilities in the package which are designed exclusively for use within this package. Individual classes might also act as utility scopes for methods intended only to be used in the class; these would normally be private utility methods. In a language like C, where there are no explicit packages, an individual file will often serve as a utility scope for utilities inside it; also the set of files including a certain header might be the utility scope for a utility declared in that header.

However, considering narrower scopes would mean that:

1. Either the system architecture is valid (or partly valid) or
2. Techniques for the recovery of the architecture are used.

The first point would put a strong assumption on the type of systems targeted by this research, which is unlikely to be valid in practice. The second point would necessitate combining the concepts presented in this chapter with architecture recovery techniques such as the ones discussed by Wiggert et al. [Wiggert 97]. Architecture recovery is out of the scope of this thesis. Our current mechanism for the detection of utilities does not consider the scope of utilities. We therefore leave this point for future research.

4.3.2 Packaging of Utilities

The QNX software engineers confirmed the intuitive idea that utilities tend to be grouped or packaged together in a class, a namespace, a library or some other construct.

Packaging of utilities is also raised in a number of other contexts: A UML utility class for example, is a module that groups together utilities that would otherwise have no "home". Tzerpos and Holt [Tzerpos 00] observed that software engineers tend to group components with large fan-in into one cluster that they call the *support library cluster*.

It is important to differentiate these utility packages from the individual utilities they contain. In fact, not all utilities exist in groupings that contain only other utilities; for example, accessing methods in most classes can be considered utilities although the classes that contain them are not necessarily utilities.

In our brainstorming session, another issue that was raised is that the utilities of a system are often designed or maintained by specific groups of people. This knowledge can help detect utilities in cases where there is little or no explicit packaging.

4.3.3 Role of a Utility

In the previous two points, we have discussed the scope characteristic in which utilities are defined and accessed, as well as how they are packaged. However, scoping and packaging are applied to many things, not just utilities. We must dig deeper in order to consider the role of utilities as distinct from other entities.

Perhaps the clearest suggestion of a utility's role can be seen in the term "programming convenience" used in the UML definition discussed earlier. QNX software engineers also said that a utility usually represents a detail that is needed to implement a general design element; it is at a lower level of abstraction than the design element it implements. Indeed, in many cases, utilities can be seen as logical extensions of programming languages, which all provide built-in facilities for manipulating data and interacting with operating systems. In fact, one can extend some programming languages to explicitly incorporate user-defined utilities; this is common practice in Smalltalk. For example, in that language you can readily add new control constructs to the system as a whole, new basic algorithms to many system classes, and even new syntactic elements to the language.

From our brainstorming session, it became clear that utilities are things that a programmer should not have to worry about when trying to see the 'big picture'. A programmer should understand the details of a programming language before attempting to program in it. Similarly, a skilled programmer will naturally understand 'at a glance' what a user-defined utility is doing without having to look at its details.

4.3.4 Our Working Definition of the Concept of Utilities

Given the above discussion, we define a utility as: *Any element of a program designed for the convenience of the designer and implementer and intended to be accessed from multiple places within a certain scope of the program.*

Many utilities will be designed to be reused in multiple programs; this definition does not preclude that, but does not require it. Also the definition allows a utility to be a method, class, package or some other element, and to be accessed from a scope that could be as narrow as a class or as wide as the entire system. A key to the definition is that a utility will be accessed from an unknown number of places, not just one. The definition does not require utilities to be grouped in any way, although it does not preclude that.

4.3.5 Definition of the Concept of Implementation Details

We want to note that according to the above definition of utilities, not all implementation details will be considered utilities. Many routines that implement details in algorithms might be designed to be called from one specific place. Later on, when we examine the compacting of traces by the removal of utilities, it may be necessary to consider the removal of other implementation details as well. We define an implementation detail as: *Any element of a program whose presence could be suppressed without reducing the overall comprehensibility of the design of a particular feature, component or algorithm*. This definition is, of course, dependent on the design component or algorithm being considered. Utilities are clearly one kind of implementation detail.

In the following, we discuss a set of criteria that can be used to evaluate the extent to which a routine can be deemed to be an implementation detail. These criteria will be validated in Chapter 5 when we present the case study.

Constructors and destructors:

Constructors and destructors are used simply to create and delete objects, rather than to implement the core system operations. Therefore, it may be best to ignore them while trying to understand the content of a trace.

Accessing methods:

Accessing methods are methods that return or modify directly the values of member variables. Therefore, they should be considered as implementation details.

Nested classes:

Most object-oriented languages such as Java and C++ provide the ability to create nested classes. Although the role of nested classes is not always clear, they can be used as utility classes that support the implementation of the classes that define them. Therefore, we can ignore the methods they invoke in order to reduce the complexity of a trace.

Methods of or derived from programming languages libraries:

Programming languages provide libraries that can be reused by software engineers. The methods that redefine these libraries should be considered as implementation details. Examples of such methods are the methods that implement or derive from the java.util package.

User specified implementation details:

We allow the trace summarization process to be flexible enough to allow users to specify manually components of the system that should be considered as utilities.

4.4 Fan-in Analysis

As the system undergoes several maintenance cycles, it becomes hard to distinguish the utility components from non-utilities. There is normally no programming language syntax to designate a utility, and they may or may not be given names that make it clear they are intended to be utilities. An effective tool should therefore support the automatic (or semi-automatic) detection of utilities. In this section, we discuss how fan-in analysis can be used for this purpose.

Fan-in analysis is performed on the call graph generated by static analysis of the system. We will use it to uncover the routines that have a large number of incoming edges (i.e., many dependents) and small number of outgoing edges (i.e. dependencies). These will be the candidate utilities. Our rationale is as follows: the more calls a method has from different places (the more incoming edges in the graph), then the more purposes it likely has, and hence the more likely it is to be a utility. Conversely, we would expect a utility routine to be relatively self-contained (i.e. have low coupling and high cohesion); if a routine has many calls (outgoing edges in the graph), this is evidence that it is less likely to be considered a utility. Also, routines that make many calls may be more needed in a trace summary to understand the system.

The rationale for using a static call graph is this: The static graph will normally give us a more complete graph. If we were to use the dynamic call graph, generated from the trace itself (or a set of traces), we may find many cases where a routine by chance has only one call to it (or just a few) and hence would not be considered a utility merely because the scenario that generated the graph did not result in calls from any other places. In this thesis, the graph is not weighted (i.e. all edges have the same weight). One possible consideration would be to use the number of calls that a routine makes to another routine at run time. This information can be extracted from the dynamic call graph. However, using this information as a weighting function is questionable for several reasons. First, there is no evidence that something that is called ten times due to a loop would be more or less important than a routine that is called once or twice just because it did not happen to be in an iterative construct. Second, the number of dynamic calls to a routine will differ depending on the input data provided to the system, yet the concept of a utility is related to the static perception of the software engineer, and will not vary from run to run.

4.4.1 Building a Call Graph

A static routine call graph consists of nodes and directed edges as shown in Figure 4.2, where the nodes consist of the system's routines and the edges depict a calling relationship from one routine to another. A node is created for each routine that can be reached starting from the entry points of the program. For example, in Java, entry points include the *main* method or if the program has threads then the call graph must also include all methods that can be reached starting at any *start* or *run* method.

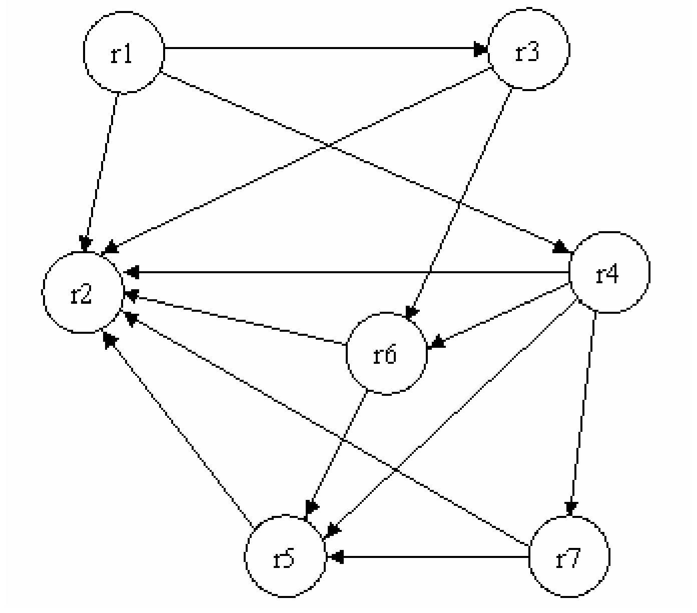


Figure 4.2. Example of a routine call graph

To build a (static) method call graph from an object-oriented system, we must first resolve polymorphic calls. There exist several techniques that accomplish this task including Class Hierarchy Analysis (CHA) [Bacon 96, Dean 95], Rapid Type Analysis (RTA) [Bacon 96], and Reaching-Type Analysis [Sundaresan 00], which differ mainly in the way they estimate the run-time types of the receiver objects.

CHA tends to be more conservative in the sense that the resulting call graph contains all possible call sites. For example, when applied to the Java example of Figure 4.3, CHA will create three edges because it will perceive all the subclasses of the 'Account' class as potential receivers. RTA is an improvement to the CHA technique by eliminating the invocations that will never occur at run-time. RTA uses information about object instantiation to remove unnecessary invocation edges from the graph produced by CHA. The analysis of the same Java example shows that the only possible invocation is the one between the 'Driver' class and the 'ChequingAccount' class. This eliminates the two unnecessary edges built by CHA.

In fact, most studies have shown that RTA is a significant improvement over CHA, often resolving more than 80% of the polymorphic invocations [Bacon 96]. Furthermore, RTA is implemented in the Jax [Jax] and Toad [Toad] tools from IBM Research, both available on the alphaWorks website.

Lately, Sundaresan et al. have developed a new technique for resolving polymorphic calls called reaching-type analysis [Sundaresan 00], which results in a better approximation of the call graph than RTA by considering chains of assignments between instantiations. The authors showed that their technique can result in 5% to 10% fewer edges than RTA. However, the authors also recognised that reaching-type analysis might be more expensive to compute. In this thesis, we use RTA for its simplicity, efficiency, and tool support.

```
class Driver {
   static void main (String[] args) {
      meth( new ChequingAccount() );
   }
   static void meth(Account a) {
        a.computInterest();
   }
}

class Account {
   abstract void computeInterest();
}
```

```
class ChequingAccount extends Account {
    void computeInterest() {
        System.out.println("Cheq account");
    }
}

class SavingAccount extends Account {
    void computeInterest() {
        System.out.println("Sav account");
    }
}

class CreditCardAccount extends Account {
    void computeInterest() {
        System.out.println("CC account");
    }
}
```

Figure 4.3. A Java program used to illustrate the resolution of polymorphic calls

4.4.2 The Utilityhood Metric

To measure the extent to which a particular routine can be considered a utility, we suggest the following utilityhood metric:

Given a routine *r* and the following:

- N = The number of routines in the routine call graph
- Fanin(r) = The number of routines in the graph, other than r, that call r.
- Fanout(r) = The number of routines in the graph, other than r, that r calls.

We define the utilityhood metric of the routine *r*, U(r)[6], as:

$$U(r) = \frac{Fanin(r)}{N} \frac{Log(\frac{N}{Fanout(r)+1})}{Log(N)}$$

[6] Note that this equation can be simplified to $U(r) = \frac{Fanin(r)}{N}(1 - \frac{Log(Fanout(r)+1)}{Log(N)})$. It is kept as indicated above to be able to better explain it.

U(r) has 0 (not a utility) as its minimum and approaches 1 (most likely to be a utility) as its maximum.

Explanation:

First, we want to note that Fanin(r) and Fanout(r) both vary from 0 to |N|-1 (i.e. self dependencies are ignored)

This formula can be split into two parts. The first part $\frac{Fanin(r)}{N}$ simply reflects the fact that the routines with large fan-in are the ones that are most likely to be utilities. For example, if the routine is called from all other routines of the system then its $\frac{Fanin(r)}{N}$ will be close to 1 (it will never reach 1 since self dependencies are ignored, i.e., Fanin(r) < N).

However, as discussed earlier, it is also important to consider the number of routines that are called *by* a particular routine. Therefore, we multiply the first part (i.e. $\frac{Fanin(r)}{N}$) by a coefficient that takes into account fan-out, although with lower emphasis. We achieve this using $Log(\frac{N}{Fanout(r)+1})$. We use $Fanout(r)+1$ for convenience to avoid situations where Fanout = 0.

If a routine *r* does not call any other routine of the system then Fanout (r) = 0, hence $Log(\frac{N}{Fanout(r)+1})$ = Log(N), which would be dependant on the size of the system under study. We divide this result by Log(N) to ensure that both this expression and the entire formula vary from 0 to 1. In the case of *r,* its utilityhood metric will be equal to $\frac{Fanin(r)}{N}$.

On the other hand, a routine that has very large fan-out will result in $Log(\frac{N}{Fanout(r)+1})$ that tends to zero cancelling the effect of fan-in. This is a routine that should not be considered as a utility.

The result of applying the utilityhood metric to the call graph of Figure 4.2 is shown in Table 4.1 that we refer to as the *Ranking Table* (in this table, the base of the logarithm is 2). This table is sorted according to the descending order of U. In this example, we can see that the routine r2 is a candidate utility routine since it is called by all other routines and does not call any routine (its U value is the highest).

Table 4.1. Ranking table computed from the call graph of Figure 4.2

Routines	**Fanin**	**Fanout**	**U**
r2	6	0	0.86
r5	3	1	0.27
r6	2	2	0.12
r3	1	2	0.06
r7	1	2	0.06
r4	1	4	0.02
r1	0	3	0.00

A tool using the utilityhood metric would need to pick a threshold, above which to consider routines as utilities, and therefore to suppress them from the trace. The exact threshold will vary depending on the context; for example, if there is a strong need to compact the trace further, then a higher threshold can be picked. Alternatively, if the trace has already been compacted too far, and the software engineer finds that he or she would like to see more detail in order to understand it, then the threshold can be reduced.

4.4.3 Trace Summarization Algorithm

The trace summarization algorithm takes a source trace as input, processes it by iteratively removing implementation details including utilities, and returns a summary of the trace as output. This process is done in a semi-automatic manner as shown below. The algorithm is deliberately underspecified, since further research is needed to determine the best settings of certain parameters (Step 1) and how to categorize and detect various other kinds of implementation details in addition to utilities (Step 2). The following are the steps of the algorithm:

- **Step 1:** Set the parameters for the summarization process. A key parameter is the Exit Condition (EC) that will be used to determine when to stop summarizing. More details about setting parameters are presented below.
- **Step 2:** Remove all known implementation details (i.e. the ones that do not necessarily have high fan-in but yet can be considered to be low-level details such as the ones presented in Section 4.3.5)
- **Step 3:** Compute U for the routines in the trace remaining after Step 2:
    - While EC is False, remove the routines invoked in the trace that have the highest remaining value of U.
- **Step 4:** Output the result of Step 3

Following Step 4, the maintainer can evaluate the result, adjust the parameters and run the algorithm again, or manually manipulate the output.

A. Some details of Step 1: Setting the parameters

Step 1 of the algorithm sets certain parameters that will guide the summarization process. The first of these is to determine an exit condition (EC) that will be used to stop the filtering process. There are several criteria that could be considered for this purpose. Perhaps the simplest one might be to compute the ratio of the size of the summary to the size of the initial trace. For example, we can stop the algorithm if the size of the summary reaches 10% of the initial size of the trace. However, due to the various types of repetitions that exist in a trace, using a simple size ratio will often not be useful. For example, simple elimination of the large number of repeated calls in one loop may cut the trace to 10% of its size, without improving our ability to comprehend it very much.

In Chapter 3, we introduced many metrics for measuring various properties of traces. One of these metrics is the number of comprehension units (Scu), which we defined as the number of distinct subtrees of the trace. We have therefore found it useful to base the exit condition, EC, on comprehension units. In particular we can define a ratio R that compares the number of comprehension units contained in the summary to the number of comprehension units of the initial trace.

More formally, let:

- Scu(T) = Number of comprehension units of a trace T
- Scu(S) = Number of comprehension units of the summary S extracted from the trace T

Then we define this ratio as R = Scu(S)/Scu(T)

Using this ratio, a suitable exit condition might be R = 10%. That is, we stop applying fan-in analysis when the ratio of the number of comprehension units of the summary to the number of comprehension units of the initial trace is equal to or less than 10%.

Another parameter to set in Step 1 of the algorithm is the matching criteria used to compute Scu; choosing appropriate criteria allows one to vary the degree of generalization of the trace content. In other words, two sequences of calls that are not necessarily identical can be grouped together as instances of the same comprehension unit if they exhibit certain similarities. In this thesis, the grouping we use is of the simplest kind: we ignore the *number of contiguous repetitions* when computing the number of comprehension units.

B. Explanation of the remaining steps

Step 2 of the algorithm proceeds by removing any known implementation details from the source trace. Examples of implementation details were presented in Section 4.3.5 and include accessing methods, methods that override the methods contained in the language library (e.g. Java.util), constructors, etc. The maintainers should also be able to manually specify a list of components that can be considered as implementation details.

Step 3 takes the traces resulting from Step 2 and applies to it fan-in analysis in order to further reduce its content. It proceeds by iteratively removing the routines with the highest remaining value of U (utilityhood) until the exit condition is satisfied.

Step 4 of the algorithm is a presentation step where the final summary is turned into a visual representation such as a UML sequence diagram and given to the users as output.

The application of the above steps is automatic. However, there is a need to account for the following situations:

- Situation 1: The resulting summary still contains too much information for the users.
- Situation 2: The resulting summary is too abstract for the users to develop a sufficient understanding of the system behaviour. For example, the removal of a certain widely-used utility might cause the number of comprehension units to drop considerably below the designated threshold.

If either of these situations occurs, the maintainer will find the summary to be uninformative, and will have to adjust the exit condition and re-run the algorithm. The maintainer might alternatively further process the result using a trace analysis tool.

4.5 Summary

In this chapter, we introduced the concept of trace summarization and how it used to summarize the content of large traces. One direct application of this concept is to recover the behavioural design models of the system as well as enable top-down analysis of traces.

Our approach for achieving trace summarization is based on the hypothesis that traces can be made easier to understand if we remove the components that are mere implementation details such as utilities. The motivation behind this hypothesis comes from the way QNX software engineers use traces as well as the study conducted by Zayour at Mitel Networks.

One of the main contributions of this chapter is the in-depth study of the concept of implementation details including utilities and a metric that can be used to measure the extent to which a routine can be considered a utility.

In addition, we presented a technique based on fan-in analysis that can be used to detect utilities. This technique is used as the main mechanism of the trace summarization algorithm.

The algorithm proceeds by reducing the size of the trace, using the number of comprehension units metric, to below some threshold.

Our approach also assumes that the users will adjust the algorithm's parameters and re-run the algorithm if they wish to try to improve the summary. Users are also expected to be able to use tools that would allow further manipulation of the results.

Chapter 5. Case Study

5.1 Introduction

In this chapter, we present a case study in order to evaluate the effectiveness of the trace summarization approach presented in the previous chapter by applying it to a trace generated from the execution of the Weka (ver. 3.0) system [Weka, Witten 99].

We do not only apply the algorithm, but also the manual steps involved in its use, such as manipulating the results (see Section 4.2.4). We then evaluate the overall approach by asking the developers of the system to provide feedback on the final results. During the evaluation process, we also validate the hypothesis, H_1, presented in the previous chapter.

This chapter is organized as follows: the next section presents the usage scenario chosen to generate the trace to summarize. In Section 5.3, we explain the evaluation process. We present the quantitative as well the qualitative results of this case study in Section 5.4 and Section 5.5 respectively. A summary of our findings is presented in Section 5.6. The content of this chapter is adapted and expanded from a paper submitted to the Journal of IEEE Transaction on Software Engineering (special issue on interaction and state-based modelling), 2005 [Hamou-Lhadj 05d].

5.2 Usage Scenario

The software feature we selected to analyze is the Weka implementation of the C4.5 classification algorithm, which is used for inducing classification models, also called decision trees, from datasets [Witten 99]. The algorithm proceeds by building a decision tree from a set of training data that will be used to classify future samples. It uses the concept of information gain to determine the best possible way of building the tree. The information gain can be described as the effective decrease in entropy resulting from making a choice as to which attribute to use and at what level of the tree. Another important step of the algorithm is pruning the decision tree. This is done by replacing a whole subtree by a leaf node to reduce the classification error rate. Weka supports various techniques that can be used to evaluate one decision tree algorithm over another algorithm using the same dataset. In this usage scenario, we chose to apply the cross-validation technique which is a procedure that involves splitting the training data into equally sized mutually exclusive subsets (called folds). Each one of the subsets is then used in turn as a testing set after all the other sets combined have been the training set on which a tree has been built.

5.3 Process Description

The process of using trace summarization consists of the following activities:

1. Instrumenting the Weka source code: We used our own instrumentation tool based on the BIT framework [Lee 97] to insert probes at the entry and exit points of each system's non-private methods. Constructors are treated in the same way as regular methods.

2. Generating a trace of method calls by exercising the target system according to the functionality under study (i.e. C4.5 algorithm): The trace was generated as the system was running, and was saved in a text file containing raw lines of events, where each line represents the full class name, method name, and an integer indicating the nesting level. For simplicity reasons, in the rest of this chapter, we refer to the generated trace as *The C45 Trace*.

3. Building the static method call graph using RTA: To apply the trace summarization algorithm, we first needed to build the method call graph by parsing the Weka system. For this purpose, we used the Toad tool [Toad], which supports the RTA technique for resolving polymorphic calls at compile time.

4. Computing the ranking table: We computed the ranking table by computing the utilityhood metric for the methods that appear in the trace.

5. Applying the trace summarization algorithm: Finally, we applied the trace summarization algorithm to the C45 trace. The results are discussed in the next sections.

5.4 Quantitative Results

In this section, we present the gain in terms of size achieved by filtering the C45 trace using the trace summarization algorithm. For a better representation of the results, we introduce the following notations:

- We identify every filtering operation using a distinct mnemonic. For example, the removal of accessing methods can be identified using the ‘ACC’ mnemonic.
- In addition, we define a transformation rule T_A = Transf(T, A) to represent a trace T_A that results from applying the filtering operation A to a source trace T.

The results of applying the trace summarization algorithms are as follows:

Step 1: Setting the parameters

The most important parameter to set is the exit condition, EC. From experience, we chose a threshold R = 10%. That is, we stop the algorithm when the ratio of the number of comprehension units of the resulting trace to the number of comprehension units of the initial trace drops just below 10%. As discussed previously, we also used simple elimination of repeated calls as our only approach to generalization. Other parameters we set included which implementation details to remove in the next step.

Step 2: Removing the implementation details

Step 2 of the algorithm deals with removing implementation details. In what follows, we present the various types of implementation details considered in this case study (a mnemonic is used to identify every single filtering operation).

- Removing methods of inner classes [INNER]: Methods of inner classes are removed from the trace. In Java, these are easy to detect using the Java reflection model.

- Removing accessing methods [ACC]: We noticed that all methods that start with ‘set’ and ‘get’ are accessing methods, which made removing such methods an easy task. However, we also found some methods that are named after specific instance variables (e.g. name(), dataset(), etc.) are also used as accessing methods. We did remove these methods. Perhaps, in the future, we will need to apply static analysis techniques such as data flow analysis to detect accessing methods although this might result in high computational overhead.

- Removing constructors and Java finalizers [CSTR]: Java constructors and static constructors are represented in the Java Virtual Machine (JVM) using the <init> and <clinit> mnemonics respectively. This is also the way they appear at run-time, which facilitate their removal. Finalizers are Java version of destructors.

- Removing methods found in the Java library [JLIB]: The methods of the Java library that were considered in this case study are the ones that belong to the java.lang.Object class (usually overridden by user-defined classes) and the classes of the java.util package (e.g. methods of the Enumeration interface, etc)

- Removing user-defined utility methods [UDEFUT]: We explored the Weka documentation to search for possible user-defined utility methods, classes or packages. We noticed that there is a class called weka.core.Utils that contains general purpose methods such as grOrEq, etc. We decided to remove its methods from the trace since they are an obvious case of utilities.

Table 5.1 summarizes the cumulative result of removing the above categories of methods. By cumulative, we mean that the table should be read using the following transformation rules:

- T_{INNER} = Transf(T, INNER)
- T_{ACC} = Transf(T_{INNER}, ACC)
- T_{CSTR} = Transf(T_{ACC}, CSTR)
- T_{JLIB} = Transf(T_{CSTR}, JLIB)
- T_{UDEFUT} = Transf(T_{JLIB}, UDEFUT)

Table 5.1. Results after removing implementation details from the C45 trace

	T	T_{INNER}		T_{ACC}		T_{CSTR}		T_{JLIB}		T_{UDEFUT}	
S	97413	87417	90%	48653	50%	40665	42%	33619	35%	31102	32%
Scu	275	271	99%	206	75%	164	60%	138	50%	120	44%
N_m	181	177	98%	160	88%	125	69%	104	57%	95	52%

Table 5.1 shows that the removal of methods of inner classes and accessing methods results in a reduction of (50%) of the size of the initial trace (S). However, the number of comprehension units (Scu) and the number of distinct methods (Nm) are still quite high (75% and 88% respectively).

Additional filtering is then necessary to reach the 10% threshold. The table shows that removing constructors/finalizers, java library methods, and user-defined utilities (i.e. methods the weka.core.Utils class) brings down the number of comprehension units to Scu = 120, which represents 44% of the number of comprehension units of the initial trace. The size of the resulting trace S is 31102 calls (i.e. 32% of the size of the initial trace), which is still rather high for humans to manage.

This first filtering phase confirms our intuition that the removal of implementation details is a crucial step for building summaries but is far from sufficient.

Step 3: Applying Fan-in Analysis

Step 3 aims to improve the results obtained in the previous step by applying fan-in analysis. For this purpose, the ranking table built from the Weka call graph was used. We proceeded by removing the routines that have high utilityhood value (the ones that are ranked first in the ranking table). After each iteration, we checked whether the exit condition, R = 10%, holds or not. This process continued until the algorithm hit a method called weka.j48.J48.buildClassifier. The removal of this method resulted in a trace that contains 156 calls (0.2% of the size of the initial trace), 20 comprehension units (7% of the number of comprehension units of the initial trace), and 20 routines (11% of the number of routines of the initial trace). Note that this trace contains considerably fewer comprehension units than the threshold (7% compared to 10%). Based on that, we decided to reverse the removal of this method and stop the fan-in analysis at a higher EC threshold.

The resulting trace is called $T_{faninut}$ and it contains S = 3219 calls (3%), Scu = 67 comprehension units (24%), and 51 methods (28%).

Step 4: Outputting the summary

At this stage of the analysis, we output the trace as a tree structure and proceeded to exploring its content.

Evaluation and processing of the content of the $T_{faninut}$:

Our initial objective was to have a summary that contains just below 10% of the total comprehension units of the initial trace. However, the algorithm in Step 3 overshot this, so as mentioned we backed up and stopped at 24% (i.e. we are in Situation 1 as described in Section 0). Therefore, we decided to further explore the content of the final trace, $T_{faninut}$, in order to make some further adjustments. This process was done using a trace exploration tool called SEAT (Software Exploration and Analysis Tool) [Hamou-Lhadj 05a, Hamou-Lhadj 04c] that we have developed to support most of the concepts presented in this thesis. SEAT implements various capabilities for the exploration of traces. Using SEAT, a software maintainer can explore the trace by searching for specific components, map the trace content to the other system artefacts using the facilities of the Eclipse environment, filter the trace content using several techniques such as pattern matching, sampling, and so on, detect patterns of execution using various matching criteria, allow the user to add specific trace components to a list of utilities that can be used during the processing of other traces generated from the same system, etc. SEAT uses CTF (see Chapter 6) to model the traces, which allows it to scale up to processing large traces. Finally, SEAT implements most of the metrics presented in Chapter 3.

Exploration using the tool showed that the method called *buildTree* generates three additional levels of the tree representation of the trace and most of the methods that appear in these levels have small fan-in (1 or 2) and small fan-out (1 or 2). The role of the *buildTree* method is to build the decision tree that is used by the C4.5 algorithm. At this point, we thought that the details of how the tree is built might be something that can be hidden and that it is sufficient for a summary to have an indication that a tree is being built. Therefore, we decided to remove the methods generated from the *buildTree* method from the summary. The whole process took no more than fifteen minutes and involved expanding and collapsing the tree along with displaying statistics about the content of the trace – these operations are efficiently supported by SEAT. Whether the content of the *buildTree* method should be kept in the summary or not is something that we will discuss in the next section in the context of evaluating the content of the summary.

The resulting trace is called T_{adjust} and contains 453 lines (0.5% of the initial size), 26 comprehension units (10% of the initial number of comprehension units), and 26 methods (14% of the initial total of methods).

Finally, the trace was converted manually into a UML sequence diagram (Figure 5.1a and Figure 5.1b) where the contiguous repetitions have been collapsed (some additional notations have been used to show repeated sequences such as the Loop and (*) constructs). The sequence diagram and the tree representation of the final trace were presented to the Weka software developers for evaluation.

5.5 Questionnaire Based Evaluation

One of the most difficult questions when evaluating a summary is to agree about what constitutes a good summary. In other words, what distinguishes good summaries from bad summaries (assuming that there can be a definitive answer to this that people will agree on reasonably well)?

In text summarization, there are two techniques for evaluating summaries: extrinsic and intrinsic evaluation. The extrinsic evaluation is based on evaluating the quality of the summary based on how it affects the completion of some other task [King 98]. The intrinsic evaluation consists of assessing the quality of the summary by analyzing its content [Paice 93]. Using the intrinsic approach, a summary is judged according to whether it conveys the main ideas of the text or not, how close it is to an ideal summary that would have been written by the author of the document, etc.

In this thesis, we adopt an intrinsic approach leaving the extrinsic evaluation for future research. The objective is to assess the trace summarization technique with respect to whether or not it extracts summaries that would convey the main content of a trace. We will also evaluate the summary with respect to its similarity to a behavioural model that a software engineer would design.

To perform intrinsic evaluation, we designed a questionnaire with thirteen questions that aim to evaluate various aspects of the extracted summary. The questionnaire was given to nine software engineers who have experience with using the Weka system: Either they were part of the Weka development team or they added new features to the system. The questionnaire consists of the following categories that we present here and discuss in more detail later:

- Background of the participants
- Order of calls of the trace summary
- Quality of the summary
- Usefulness of a summary in software maintenance tasks

5.5.1 Background of the Participants

We designed four questions to enable us classify our participants according to their expertise in the domain represented by Weka (i.e. machine learning algorithms) as well as their knowledge of the system structure.

For each question, the participants selected from fixed values ranging between *'Very poor'* (score of 1) and *'Excellent'* (score of 5). The questions are:

Q1. My knowledge of the Weka system (i.e. classes, methods, packages, etc.) is:

Q2. My knowledge of the domain represented by Weka is:

Q3. I read Chapter 8 of the book written by Witten and Frank that describes the Weka system and my understanding of it is (skip this question if your did not read this chapter):

Q4. My experience in software development is:

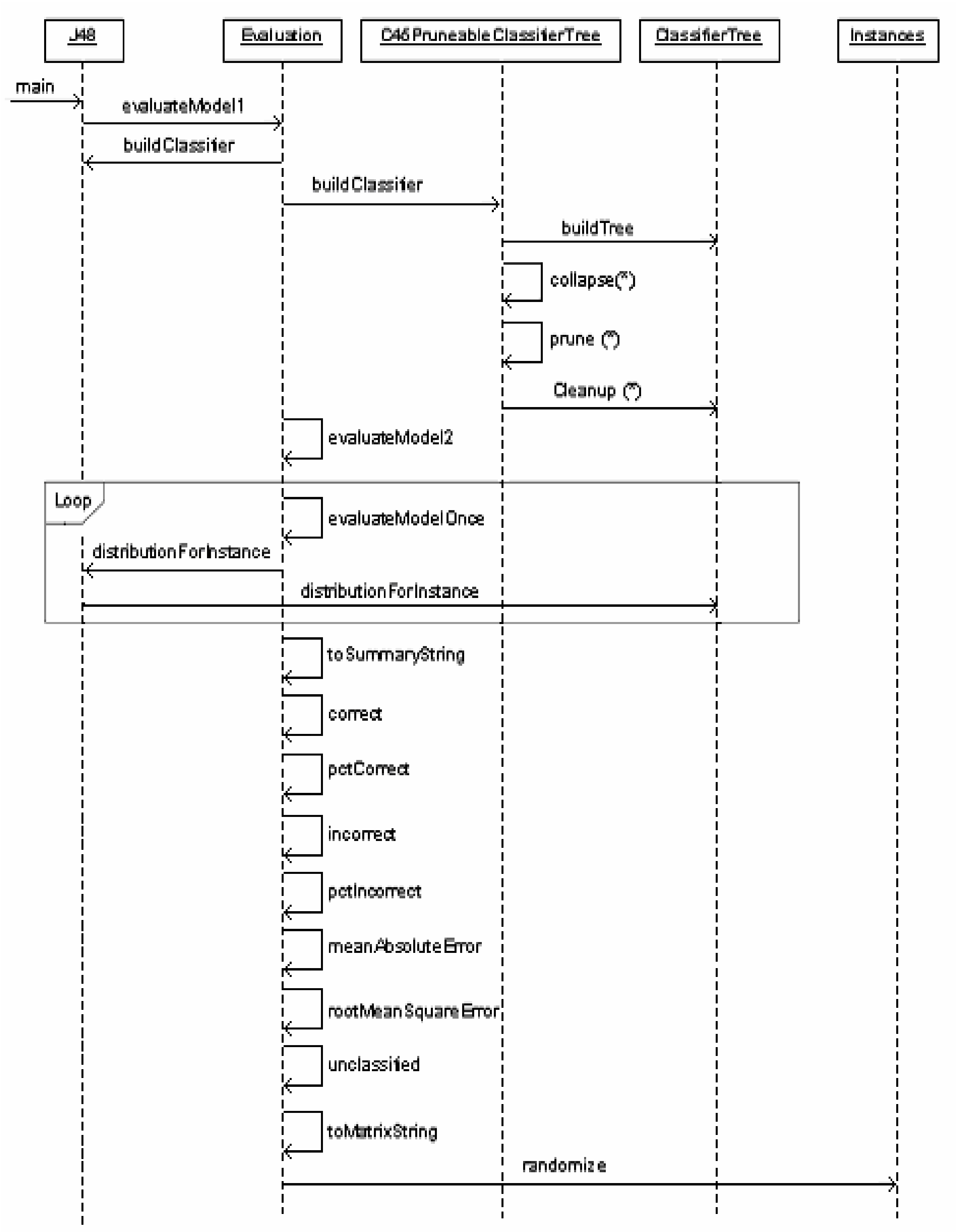


Figure 5.1a. The first part of the summary extracted from the C45 trace. This part deals with building classifiers and classifying the input the dataset

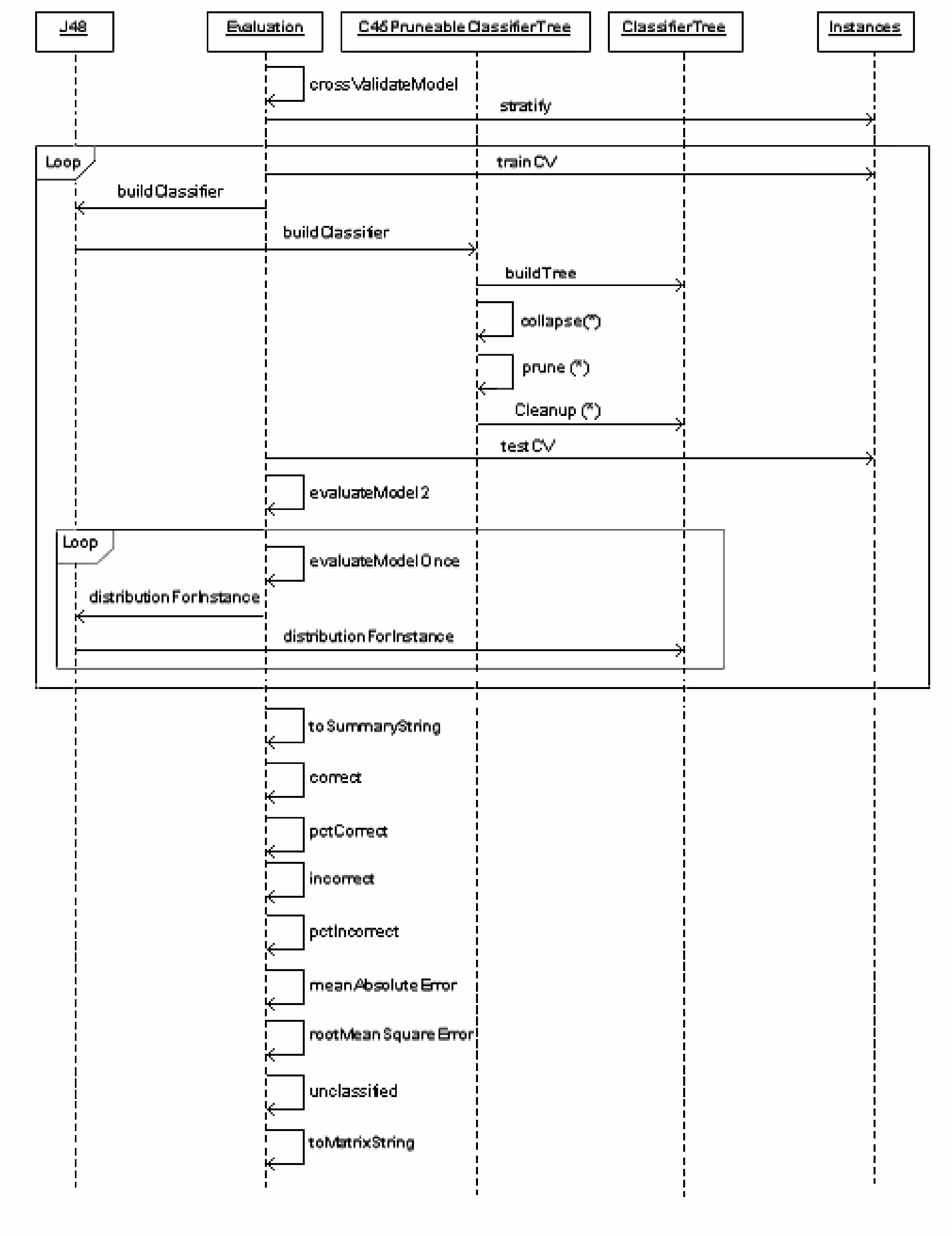


Figure 5.1b. The second part of the summary extracted from the C45 trace. This part deals with the cross validation process

The third question (Q3) refers to the book written by Witten and Frank [Witten 99] that devotes an entire chapter (Chapter 8) to the architecture of the Weka system, which can be used by software developers to add new features to Weka.

Table 5.2. The participants' background

Questions	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**Average**
Q1 (Structure)	3	3	4	4	4	5	5	5	5	4.2
Q2 (Domain)	4	4	4	4	4	5	5	5	5	4.4
Q3 (Chapter 8)	3	4	4	4	4	4	4	5	5	4.1
Q4 (SW Dev)	-	-	4	-	5	5	5	5	5	4.8

Table 5.2 shows the answers of the participants (P1 to P9), which can be divided into three groups according to the knowledge they have of the Weka structure (Q1) as well as the knowledge they have of the domain (Q2). The first group consists of participants P1 and P2 and can be qualified as intermediate users since they have an average knowledge of the Weka internal structure (score of 3) although they have good knowledge of the domain (score of 4). The second group consists of participants P3, P4, and P5 and we refer to them as experienced users (they all scored 4 out of 5 in both questions Q1 and Q2). Finally, the last group includes participants P6, P7, P8, and P9 and we call them experts since their knowledge of the internal structure of Weka as well as the domain is excellent (score of 5 for Q1 and Q2). These are also the users who contributed to the original development of Weka. Table 5.3 classifies the participants according to the category of users they belong to.

Table 5.3. The participants' background by categories

Group	**Participants**
Intermediate	P1 P2
Experienced	P3 P4 P5
Experts	P6 P7 P8 P9

In addition, all participants except P1 have good to excellent experience is software development. Presuming that they were also involved in maintaining software, their feedback will certainly help us evaluate the overall effectiveness of a trace summary in performing software maintenance tasks.

Among the participants who read Chapter 8 of the Weka book, P5 (an experienced user) and all four experts have an excellent understanding of its content. On the other hand P3 (an intermediate user) has a good understanding of the content of this chapter. It is important to mention that reading the chapter does not necessarily give the reader a deep insight into the Weka implementation: The book only describes the overall architecture of the tool.

5.5.2 Order of Calls of the Trace Summary

The goal of this category of questions is to cross-check the knowledge and attention of the participants. We achieve this by asking the participants to assess the extent to which the extracted summary maintains the order of calls of the initial trace. Since the trace summarization process merely cut methods, the order of calls will of course be kept. A participant who says the order is not correct would either lack knowledge of the system, or else would be holding an incorrect mental model.

However, it is important to note that if generalization techniques are used to extract summaries then checking the correct order of calls would be important. This is because generalization might involve treating sequence of calls as sets which can significantly alter the flow of execution of the initial trace.

In this category, we asked one question, which is:

Q5. In your opinion, to what extent does this summary preserve the correct order of calls? That is, there is no call that is made that contradicts the way the system is implemented.

The participants selected from fixed values ranging between *'The order of calls is completely inaccurate'* (score of 1) and *'The order of calls is well preserved'* (score of 5).

The first row of Table 5.4 summarises the participants' answers to Q5; it shows that the opinions of the participants vary from believing the order of calls is at least somewhat preserved (score of 4) to believing the order is well preserved (score of 5), which confirmed the knowledge of the system they claimed to have as well as the fact they had almost a correct mental model of the implementation of the traced feature.

When asked to elaborate on their answers, the participants pointed out the fact that the *evaluteModel1* method that is invoked in the beginning of the sequence diagram (Figure 5.1a) is called before the *buildClassifier* method. According to them, this is inconsistent with the way the C4.5 algorithm works and eventually represented in Weka. Indeed, conceptually the algorithm certainly builds the classifier and then starts the evaluation. However, after checking the source code, we realized that all Weka classifiers are executed through the *evaluateModel1* method although this method does not perform any evaluation. It only takes the name of the classification algorithm as a string, it builds the classifier (by creating an object of the corresponding classifier, e.g., C4.5) and then it calls a second method called *evaluateModel2* to perform the actual evaluation. The summary shows that *evaluateModel2* is invoked after building the classifier. Therefore, the trace as well as its summary did reflect the correct order of calls. It is just that the use of the same method name in Weka created confusion.

In addition, the discussion regarding the order of calls led to three interesting observations. Firstly, most of the participants said that they relied on their knowledge of the C4.5 algorithm and the method names to assess how the order of calls should be. This is interesting because without a good naming convention, one might never understand a summary effectively. By analogy, imagine having a summary of an English text where the words that are included are not proper English. Secondly, after discussing the content of the summary with the participants, it became obvious to us that the knowledge of the domain (or at least the traced feature) is a key for comprehending the content of a summary. In fact, without this knowledge, it will be hard to get any useful information out of a summary. This is like reading a summary of a document without having any idea about the topic of the document. Finally, the confusion raised by the *evaluteModel* method confirmed the idea that a tool that manipulates summaries must allow the user to map their content to the other system artefacts such as the complete source code. In fact, summaries themselves might include more than simple method names. For example, it might be better to add the list of parameters, source code comments, etc.

Table 5.4. Answers to questions about the content of the summary (Q5, Q6, Q7 and Q8)

	Intermediate		**Experienced**			**Experts**				
Questions	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**Average**
Q5 (Order)	4	5	4	5	5	4	4	5	4	4.3
Q6 (Constructors)	1	1	2	2	1	1	1	1	1	1.1
Q7 (Accessing)	1	1	2	1	2	1	1	1	1	1.1
Q8 (Inner)	-	-	1	1	2	1	1	1	5	1.7

5.5.3 Quality of the Summary

The objective of this category of questions is to assess whether the extracted summary captures the main interactions that implement the traced scenario. Several questions were asked due to the importance of this category. We divide these questions in to three subcategories: removing components; thorough analysis of the summary; and ranking the quality of the summary.

Removing components:

Questions Q6 to Q8 aim to evaluate the effect of removing constructors, accessing methods, and methods of inner classes on the quality of the summary. The effect of removing other types of implementation details such as the routines that implement (or derive from) Java library components and the ones that belong to the weka.core.Utils class has been discussed informally with the participants. As expected, all participants agreed that they constitute implementation details and indeed they should be removed. We did not feel the necessity to capture this feedback in the questionnaire so as to keep the users' focus on more important concepts.

The questions are:

Q6. *We deliberately removed the* **constructors** *from the summary. To what extent does this affect the quality of the summary?*

Q7. *We deliberately removed the* **accessing methods** *from the summary. To what extent does this affect the quality of the summary?*

Q8. *We deliberately removed* **methods of inner classes** *from the summary. To what extent does this affect the quality of the summary?*

The participants selected from fixed values ranging between *'Not at all'* (score of 1) and *'Very much'* (score of 5).

The results are also shown in Table 5.4. All intermediate and expert users said that the removal of constructors (Q6) does not affect the quality of the summary at all. However, two participants (P3 and P4) out of the three experienced participants said that this might affect it a just little. One comment made by P3 is that sometimes it is useful to retain the constructors if a class contains many constructors and that it might be important to know which one is called in order to have an idea of which path of the system is being executed. P4 explained that some constructors might be useful but certainly not all of them. When we asked which ones can be useful, the participant replied that these might be the constructors of the most important classes of the implementation of the C4.5 algorithm such as constructors of the J48 class (this is the Weka class that implement the core of the C4.5 algorithm).

In addition, we asked our participants to tell us why they think that constructors are not useful at a high level. Among the various answers, P5 (experienced user) commented that this is because they do not add any new information and that most of the time people already know what they do. This corresponds exactly to the comments made by the QNX software engineers when asked about what elements of the system should be considered as utilities. Our users also said that if they know what a piece of code does, that they will most likely want to hide its specifics and just skip it and look at something else.

All participants, except two experienced participants (P3 and P5), agree that the removal of accessing methods (Q7) does not negatively affect the content of the summary. The only comment made is that some accessing methods might return important information that will help understanding the rest of the trace. However, even the dissenters agreed that most accessing methods should indeed not appear in the summary.

Question Q8 asked the participants about the effect of removing methods of inner classes on the quality of the summary. Intermediate participants (P1 and P2) said that they are not familiar with the Weka inner classes. Therefore, they did not provide an answer. Participant P8 (an expert) said that they might affect the quality of the summary just a little assuming they work correctly (i.e. they are free of defects). According to him, one might need to explore their content to uncover a buggy behaviour. However, the participant added that if the purpose of the summary is other than fixing bugs then we might as well ignore inner classes due to the low-level granularity they represent.

One of the most surprising answers was provided by P9, one of the experts. According to this participant, methods of inner classes are important for the understanding of the internal structure of some characteristics of the C4.5 algorithm such as how splitting, collapsing and pruning decision trees work. After analyzing the source code and reading the Weka documentation, we found that, in this particular case, inner classes are used to hold data structures such as HashKey, NeighbourList, etc. When we asked the participant how these classes can be useful at a high level, he replied that although he likes to have the big picture, he prefers to go into the details when it comes to understanding the exact behaviour. We found this answer compatible with what we are trying to achieve. As we previously stated, a summary can only reflect the overall behaviour that can be used to build a global understanding of the trace. It is clear that depending on the maintenance task at hand, a software engineer will most likely need to dig deeper to understand the details.

The next two questions (Q9 and Q10) are concerned with a thorough analysis of the trace summarization process through the content of the C45 summary. We want to know whether the summary still contains methods that need to be further removed or whether, in contrast, it lacks important methods that were removed by the trace summarization algorithm.

Thorough analysis of the summary:

Question Q9 asks the participants to go through a list of the distinct methods that appear in the summary and put an X mark next to any method that *should not* appear in the summary. This would typically be a method that is a utility or implementation detail but was not detected by the trace summarization process.

The participants' answers were different, as one could expect. There are three types of answers that we distinguished. Participants P1, P4, P6 and P7 did not mark any method. According to them, the methods presented in the list should all be part of the summary. One of the comments made by P4 (experienced user) was that the summary is representative as it is, and even if there are one or two undesired methods, this will not affect its overall content.

The second pattern we noticed consists of the answers provided by the participants P2, P3, P5 and P9. The three of them argued that the methods related to displaying the results of the evaluation might not be needed at such a high-level representation of the trace. P9 suggested to further get rid of the methods *toSummaryString*, *toMatrixString*, and the calls they generate. According to him, the fact that the evaluation process took place (this is done using the *evaluateModel2* method) should be enough for someone who wants to understand the overall content of the trace and that the display of the results should be implicit. P3 added that *toSummaryString* only prints some results to the console which might not be needed in the summary. Both, participants P5 (experienced) and P2 (intermediate) made the same comment except that they still want to see the methods *toSummaryString* and *toMatrixString* but hide the methods they call such as *incorrect*, *pctIncorrect*, etc. According to them, this provides a level of detail that is not compatible with the level of detail provided for the other parts of the trace. For example, participant P2 said that the summary should be consistent in the level of abstraction it represents. When asked to elaborate on this particular comment, the participant replied that at some parts of the trace such as the part where the decision tree is built (see *buildTree* in Figure 5.1a and b), the summary does not say anything about how this is done. However, in other places such as the portions where the results of the evaluation are displayed, the summary provides considerably more detail.

The last pattern of answers we noticed was provided by participant P8 (expert), who said that there is no need to show the method *evaluateModelOnce* and the methods it generates. According to him, it is enough to know that the method *evaluateModel2* is executed and that it might be better to hide its internal structure. The role of the *evaluateModelOnce* method is to evaluate every instance of the dataset. The participant also added that the *cleanup* method seemed to be for implementation purposes. After analyzing the source code, we discovered that the role of the *cleanup* method is to free memory space. Therefore, it should not appear in the summary since it is typically an implementation detail.

The various answers of the participants lead to two observations: The first is that all participants confirmed that most methods that appear in the summary deserve to be in the summary. The second remark is that, as expected, it seems that no matter which methods are selected to be in the summary, there will always be some methods that are important to some users while less important to others. This supports the idea that a summary of a trace is not effective unless it allows enough flexibility to adjust the quantity and type of information it contains.

Question 10 asked the participants to go through a list of distinct methods that *were removed* from the trace and indicate with an X mark next to the ones that should appear in the summary. These are the methods that the process removed inappropriately.

Again, the answers vary from one participant to another. P7 and P9 (two experts) did not indicate any method. According to them, the methods that were removed are details that might be hidden at higher levels. They added that it all depends on the level of details that is wanted. For example, knowing that a decision tree is being built might be enough for people who have an extensive expertise of the system structure: They do not want to see what it is involved in building the tree.

However, most participants disagreed with that. In fact, they all pointed out to the fact that the summary does not tell much about how the decision tree is built. It only shows that the *buildTree* method is called. Most of them expected to see more about how the internals of the *buildTree* method. One expert (P6) mentioned that although the summary refers to the classes that implement the C4.5 algorithm such as the classes J48 and C45PruneableClassifierTree, it might be hard for someone who does not know the difference among the various classification algorithms supported by Weka to understand what distinguishes C4.5 from other algorithms. From the list of the removed methods, he marked the ones that deal with computing the information gain ratio (e.g. *infoGain*, *gainRatio*, *split*, *splitCritValue*, etc). The other participants marked similar methods, but with some noticeable differences. For example, some participants suggested having the methods that return the size of the decision tree after it is built (e.g. *numLeaves*, *numNodes*), others prefer focusing on the information gain ratio, and that the size of the decision tree might not be important.

Overall, the methods that most participants wished to see consist of the ones that we removed during the evaluation and processing of the content of the summary (after Step 4 of the trace summarization algorithm). It seems that the 10% threshold was lower than we expected. In fact, if R were set to 24 %, this would have resulted in including in the summary the methods generated by the *buildTree* method. On the other hand, this will also let many other undesirable methods that were removed find their way into the summary.

In addition, the total count of methods that the participants asked to include in the summary represent only 19% of the total number of methods that were listed as removed using fan-in analysis and using a trace analysis tool. This means that all participants agreed that the other 81% are indeed implementation details.

Ranking the quality and importance of the summary:

The next three questions aim to assess the overall quality of the summary. Question Q11 asked:

Q11. How would you rank the quality of the summary with respect to whether it captures the main interactions of the traced scenario?

The participants were asked to select from fixed values ranging between *'Very poor'* (score of 1) and *'Excellent'* (score of 5).

Table 5.5. Answers to questions about to the quality of the summary (Q11, Q12, Q13)

	Intermediate		**Experienced**			**Experts**				
Questions	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**Average**
Q11 (Quality)	4	4	4	4	4	4	5	5	3	4.1
Q12 (Diagram)	4	5	3	4	4	4	4	5	3	4
Q13 (Effectiveness)	4	4	5	5	5	4	4	5	4	4.4

Table 5.5 shows that intermediate and experienced participants all agree that the summary captures the most important interactions of the trace. Two experts added that it is actually an excellent representation of the main interactions. Participant P9 (an expert) commented that the summary lacks relevant information about how the decision tree is built and therefore ranked it as an average (score of 3) representation of the main events.

Question 12 asked:

Q12. If you designed or had to design a sequence diagram (or any other behavioural model) for the traced feature while you were designing the Weka system, how similar do you think that your sequence diagram would be to the extracted summary?

The participants were asked to select from fixed values ranging between *'Completely different'* (score of 1) and *'Very similar'* (score of 5)

Most participants including three experts answered that the sequence diagram they would have designed would most likely be similar (sometimes even very similar) to the summary extracted semi-automatically from the trace. However, participants P3 (experienced) and P9 (expert) commented that their design would have been slightly more concise than the summary. They mostly refereed to the fact that the summary lacks details about building the decision tree.

Question 13 asked:

Q13. In your opinion, how effective can a summary of a trace be in software maintenance?

The participants were asked to select from fixed values ranging between *'Very ineffective'* (score of 1) and *'Very effective'* (score of 5).

All participants agreed that a trace summary can be effective in software maintenance. Many of them added that this is a very good way to understand what the system is doing when the documentation is out of date or simply inexistent. We asked the participants to explain in which way summaries can help perform maintenance tasks. The answers were similar. The most common answer was that a summary can help understand the system behaviour, which will lead to performing maintenance tasks faster. Recovering the system behavioural models was the next most common comment made by most participants. Indeed, recovering the documentation has always been a challenging task, and when it is done it usually focuses on the system architecture. The techniques for recovering dynamic models are also needed just like in forward engineering where engineers focus on developing both static and dynamic views of the system.

5.6 Summary

We applied the trace summarization process to a trace generated from the Weka system. Initially, the trace contained 97413 method calls. The extracted summary, which contained 453 calls, was transformed into a UML sequence diagram and given to nine software developers of the Weka system for feedback.

Most participants agreed that removing constructors, accessing, methods, and methods of inner classes does not affect the quality of the summary in terms of the information it conveys.

Fan-in analysis has resulted in reducing the size of the trace significantly but did not allow the trace summarization to meet the exit condition. Additional filtering was done semi-automatically using a trace analysis tool.

Several aspects of the summary were evaluated. Most participants agreed that the extracted summary represents a useful abstraction of the traced scenario (Question 12). They also agreed that the summary did capture the main interactions invoked in the trace (Question Q11). When asked about the usefulness of trace summarization to perform maintenance tasks (Question Q13), the participants said that the concept can be helpful for maintaining systems with poor documentation.

In conclusion, the feedback received from the participants is in support of the hypothesis, H_1, stated in Chapter 4. We therefore confirm the fact that understanding the main content of a trace can be made easier if low-level implementation details are filtered out.

Chapter 6. The Compact Trace Format

6.1 Introduction

Existing trace analysis tools such as the ones presented in Chapter 2 use different formats for representing traces, which limits sharing and reuse of data. Although they have common features, each of them has its own advantages and specialized functions. Currently, the only way to take full advantage of these functions is to convert data from one format to another. Writing converters to and from all available formats would be impractical. To address this issue, we have developed an exchange format called CTF (Compact Trace Format) for representing traces of routine (method) calls.

CTF is built with the idea of scalability in mind. It takes advantage of the fact that dynamic call trees can be transformed into ordered directed acyclic graphs (DAG) by representing similar subtrees only once as shown in Chapter 3. The original trace can be reconstructed in the normal case where exact matching is used when comparing subtrees. In this default case, CTF is therefore a lossless representation of the trace.

Some of the advantages for having a standard exchange format can be summarised in what follows:

- It reduces the effort required when representing traces.
- It allows researchers to use different tools on the same input, which can help compare the techniques supported by each tool.
- It allows maintainers to combine the techniques from different tools to realize the given maintenance task.

An exchange format consists of two main components [Bowman 99]: A schema (i.e. metamodel) that represents the entities to exchange and their interconnections, and the syntactic form of the file that will contain the information to exchange (i.e. the instance data).

In this chapter, we present the CTF schema and discuss how it is used to represent the information needed in order to exchange traces of routine calls. We also discuss how CTF instance data can be 'carried' using existing syntactic formats such as GXL (Graph eXchange Language) [Holt 00] or TA (Tuple Attribute Language) [Holt 98].

The remainder of this chapter is organized as follows: In Section 6.2, we present an overview of existing metamodels used to describe dynamic information. In Section 6.3, we discuss the design of CTF with respect to well-studied requirements for the development of an exchange format. We present the CTF abstract syntax, semantics, and syntactic form in Section 6.4. Finally, CTF tool support and adoption is the subject of Section 6.5.

The content of this chapter is adapted and expanded from a paper appearing in the ICSM first International Workshop on Meta-models and Schemas for Reverse Engineering (ATEM), 2003 [Hamou-Lhadj 04d] and a paper submitted to the Journal of Software and Systems Modeling, 2005 [Hamou-Lhadj 05e].

6.2 Related Work

Several authors have discussed the benefits of representing a dynamic call tree in the form of a directed acyclic graph [Jerding 97b, Larus 99, Reiss 01] but none of them attempted to build a metamodel upon this concept.

Reiss and Renieris proposed a technique called string compaction that can be used to represent a dynamic call tree as a lengthy string [Reiss 01]. For example if function 'A' calls function 'B' which in turn calls function 'C', then the sequence could be represented as 'ABC'. Markers are added to indicate call returns. For example, a sequence 'A' calls 'B' and then calls 'C' will be represented as 'ABvC' (the marker 'v' will indicate a call return). However, we posit that this basic representation has a number of limitations: It does not have the flexibility to attach various attributes to routines, and cannot be easily adapted to support statement-level traces.

In her master's thesis [Leduc 04], Leduc presented a metamodel for representing traces of method calls, which supports also statement-level traces. Leduc used UML class diagrams to describe the components of the metamodel and the relationships among these components. However, one of the major drawbacks of her approach is the fact that the metamodel is an exact representation of the dynamic call tree. That means that if a trace, for example, contains one million calls then using Leduc's metamodel a trace analysis tool will need to create one million objects. Leduc's metamodel does not take into account any sort of compaction scheme.

There are other trace formats that exist in the literature. However, most of them do not necessarily deal with traces of routine calls. In [Brown 02], the authors presented STEP, a system designated for the efficient encoding of program trace data. One of the main components of the STEP system is STEP-DL, a definition language for representing STEP traces, which contain various types of events generated from the Java Virtual Machine including object allocation, variable declaration, etc. STEP is useful for applications that explore Java bytecode files. STEP-DL is a specialized metalanguage that describes the structure of the events supported by STEP. The authors argued that a specialized language is a better choice than using a language based on an existing mark-up approach such as XML. Their first argument is that, as noted by Chilimbi et al. in [Chilimbi 00], the wordiness of XML is incompatible with the key compactness requirement for traces. Second, the syntax for document type definitions (DTDs) in such languages tends to be complex for the task at hand.

Hyades is the Eclipse Test and Performance Project with the aim to "build a generic, extensible, standards-based tool platform for testing, tracing, profiling, tuning, logging, monitoring, and analysis" [Hyades]. Hyades integrates very sophisticated trace collection techniques using dedicated software agents. The 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. The CTF model is specifically designed to enable program understanding tools to exchange traces of routine calls, which form a natural hierarchy. The DAG-based CTF model is not directly supported by the trace model used in Hyades. However, a trace using the CTF model can be built upon information extracted from a trace using the Hyades model. Extra information found in the Hyades model, such as temporal information, correlation between threads, etc. can also supplement the 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. So these two models are complementary.

Other trace formats such as PDATS [Johnson 01], HATF [Chilimbi 00] and MaStA I/O [Scheuerl 95] have also been proposed. These formats focus on completely different type of traces. PDATS is a family of trace encoding techniques used to encode address and instruction traces, which are commonly used in the performance analysis of computer systems (e.g. simulation of cache memory, pipelined ALUs, load-store units, and other functional units). HATF is a trace format used to represent heap allocation traces (i.e. the events targeted are malloc, free, etc). MaStA I/O focuses on read/write traces that can be used to determine the cost of these statements when applied to databases. All these formats rely on encoding techniques such as the ones found in data compression.

6.3 Requirements for the Design of CTF

Requirements for a standard exchange format have been the subject of many studies [Bowman 99, Lethbridge 97, St-Denis 00, Woods 99]. The following subsections describe the requirements, which synthesise and extend the previous work that we used to guide the design of CTF.

6.3.1 Expressiveness

One important aspect of an exchange format is to support the various types of data that need to be shared. The study we conducted in Section 2.3 shows that most trace analysis tools would expect to manipulate traces of method calls at three levels of granularity at least: object, class, and subsystem level.

However, in order to allow non-OO systems to use CTF, we need to permit enough flexibility to represent the necessary constructs that are involved. For example, if the system is written in C then one possible scenario is to include the system files containing the invoked routines.

In addition to this, due to the multi-threaded nature of most existing software systems, the design of CTF needs to consider the threads of execution that are generated from the execution of a given system scenario.

Although this thesis is tuned towards program comprehension, we do feel the need to build in the flexibility to represent timestamps, and other statistical information such as the execution time of the routines. We believe that this will enable other types of applications such as profilers to use CTF.

Finally, the maintenance of a software system will typically involve the static as well as the dynamic aspects of the system. Having said this, we need to consider the fact that CTF can be used with existing metamodels that capture the static components of the system. In this chapter, we show how CTF can be used with the Dagstuhl Middle Metamodel (DMM) [Lethbridge 03] to satisfy this requirement.

6.3.2 Scalability

The adoption of any exchange format for representing execution traces will greatly depend on the capability to support large-sized information. In Chapter 3, we showed that an efficient way for representing traces is to turn them into ordered directed acyclic graphs. We also showed that such transformation could reach a graph-to-tree ratio of 3% (i.e. gain of 97%).

In addition to this, encoding techniques can be used to further reduce the physical storage space allocated to traces. For example, one can encode the routine names using special identifiers to avoid dealing with lengthy strings. In [Reiss 01], Reiss et al. describe several encoding techniques that aim to compress execution traces such as string compaction, interval compaction, etc.

6.3.3 Simplicity

The simplicity requirement refers to the simplicity of the CTF specification. What is needed is to have a design that is well documented and not too complex for tool builders to integrate into their tool suites. The detailed semantics of the CTF metamodel are presented in Section 6.4.2.

6.3.4 Transparency

This requirement ensures that the information is represented without any alteration. We need to have well-defined mechanisms for generating traces in the form of CTF. In Section 3.6, we presented an algorithm for the on the fly transformation of a tree structure into an ordered acyclic directed graph. This algorithm can easily be adapted to generate CTF traces by considering the syntactic form used to convey CTF instance data. A discussion on CTF syntactic form is presented in Section 6.4.3.

6.3.5 Neutrality

This requirement refers to an exchange format that is independent of any specific programming language, platform or technology. This requirement is satisfied due to the fact that the data conveyed in traces of routine calls contain almost always the same information independently of the software system from which they were generated. We support traces that contain the following information:

- Thread name
- Package or subsystem name
- Class or file name
- Object identifier
- Routine (method) name
- Timestamp information
- Method execution time

6.3.6 Extensibility

The design for extensibility is an important requirement for the design of an exchange format. We address this issue by extensive use of abstraction in the design of the CTF schema. We also show how CTF can be extended to consider the representation of the static components of the system (see Section 6.4.1.6).

6.3.7 Completeness

This requirement consists of having an exchange format that includes all the necessary information needed during the exchange: the data to exchange as well the schema in which the data needs to be interpreted. The rationale behind this is to enable tools to perform checks on the instance data to verify its validity with respect to the schema. We address this requirement by recommending a syntactic form that supports the exchange of the schema as well as the instance data (e.g. GXL and TA).

6.3.8 Solution Reuse

This requirement consists of reusing some existing technologies in the creation of the new exchange format. This will decrease the amount of time needed for testing the new format. In this thesis, we reuse existing syntactic forms such as GXL and TA to represent CTF instance data.

6.3.9 Popularity

The popularity requirement is concerned with the adoption of an exchange format by several users (e.g. tool builders). For this purpose, we have created an API for CTF as well as an eclipse plug-in that will allow different tools to directly generate traces in the CTF format. We have also presented CTF in various conferences. CTF is also supported by SEAT (the trace analysis tool introduced in Section 5.5) [Hamou-Lhadj 05a, Hamou-Lhadj 04c].

6.4 CTF Components

6.4.1 CTF Abstract Syntax

Figure 6.1 shows a UML class diagram that describes the CTF metamodel. The components of this metamodel are discussed in the subsequent sections.

6.4.1.1 Usage Scenario

The class Scenario is used to describe the usage scenario from which the execution trace is derived. We allow a scenario to be represented by many traces to support natural situations where many traces of the same scenario are gathered to detect anomalies caused by non-determinism.

6.4.1.2 Trace Types

The class Trace is an abstract class that describes common information that different types of traces are most likely to share such as timing information and any additional comments related to the generation of the trace. To create specialized types of traces, one can simply extend this class. We added the RoutineCallTrace class to represent traces of routine calls generated from procedural systems. As discussed at the beginning of the thesis, by *routine* we mean any function, whether or not it is a method of a class. Since programming languages such as C++ allow non-method routines (simple C functions) as well as methods, we decided to represent traces of pure method calls as a subclass of RoutineCallTrace. Using this hierarchy, an analyst can create traces of simple functions calls only, traces of method calls only, or traces that combine both, such as C++ execution traces. This design decision is intended to help keep the design simple and understandable.

6.4.1.3 Nodes and Edges

We use the term *comprehension unit initiator* to refer to the root of a comprehension unit. Comprehension unit initiators are represented in the class diagram using the ComprehensionUnitInitiator class. These are also the nodes in the graph; each can have many child nodes and many parent nodes as illustrated on the diagram using the parent and child

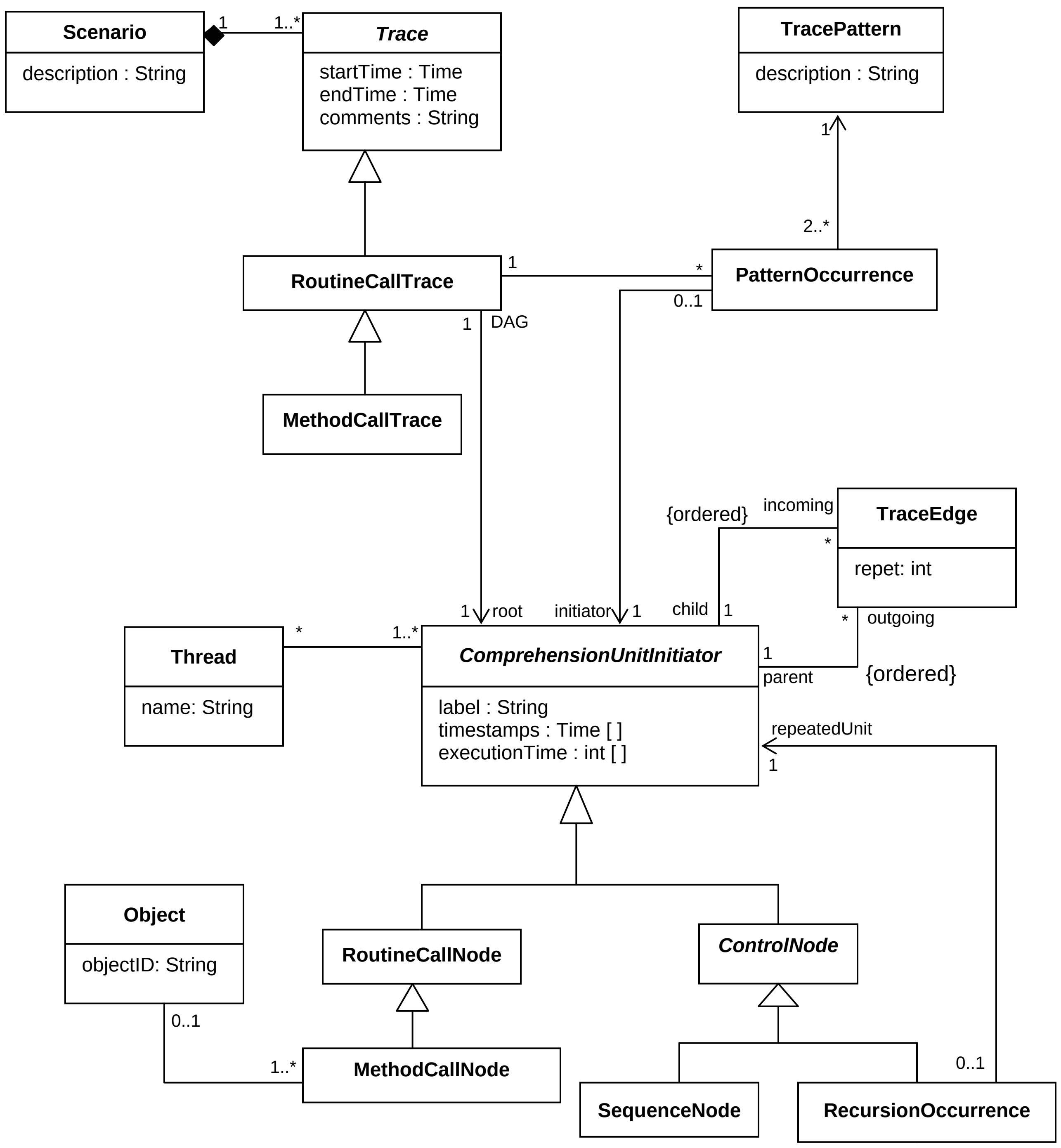


Figure 6.1. The CTF metamodel

roles. Each initiator maintains the timestamps of the routines calls it represents as well as their execution time.

Edges (i.e. calls) are represented using the TraceEdge class. An edge is labelled with the number of repetitions due to the existence of loops or recursion if there are any.

A node (ComprehensionUnitInitiator) can either be a routine call node (RoutineNodeCall), a method node (MethodCallNode) or a control node (ControlNode). Control nodes represent extra information that might be used by tools to customize parts of the traces. For example, a software engineer exploring traces might need to select a particular subtree and assign to it a description. This can easily be represented by adding a new node to the DAG that holds the description.

6.4.1.4 Dealing with Repetitions

One particular control node integrated into CTF is called SequenceNode and it is used to represent contiguous repetition of multiple comprehension units. Figure 6.2 shows how such control node can be used to represent the repetition of the sequence: BC and E.

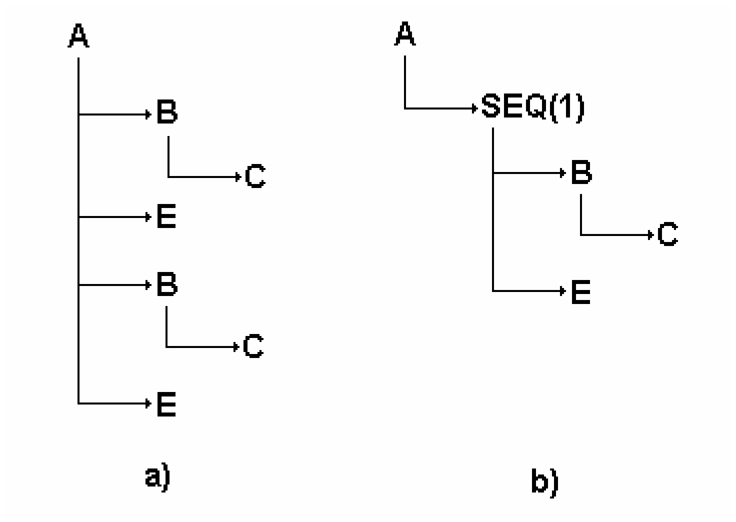


Figure 6.2. The control node SEQ is used to represent the contiguous repetitions of the comprehension units rooted at B and E

A recursive comprehension unit is represented using another control node called RecursionOccurrence, which in turn refers to the recursively repeated unit. Figure 6.3 shows a recursion occurrence node called REC that is used to collapse the recursive repetitions of the node B.

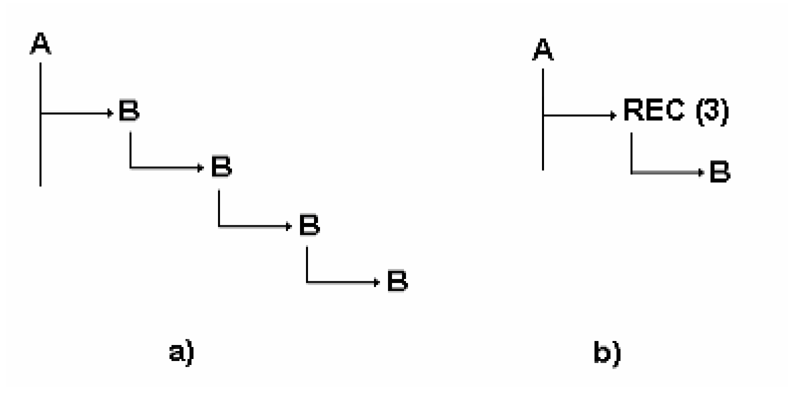


Figure 6.3. The control node REC is used to represent the recursive repetitions of the comprehension unit B

6.4.1.5 Threads

Although in other parts of this thesis we do not consider multi-threaded traces, we decided to consider them in CTF to allow the model to be more complete. Therefore, to represent thread information, we added a class called Thread. Each comprehension unit initiator has an object thread that is associated to it. Threads are identified using unique thread names since this is a common practice in languages such as Java and C++. Leduc made the same design decision [Leduc 04]. We do not distinguish between thread start/end routines from the other routines for simplicity reasons.

6.4.1.6 Static Components

Although the schema presented above focuses on describing run-time information, some of its components (e.g. RoutineCallNode) might need to refer to static components of the system.

The CTF metamodel as illustrated in Figure 6.1 relies on a string label to identify the trace events (i.e. the routines invoked). A possible extension to CTF is to consider references to actual objects representing each class, method, package, etc as shown in Figure 6.4.

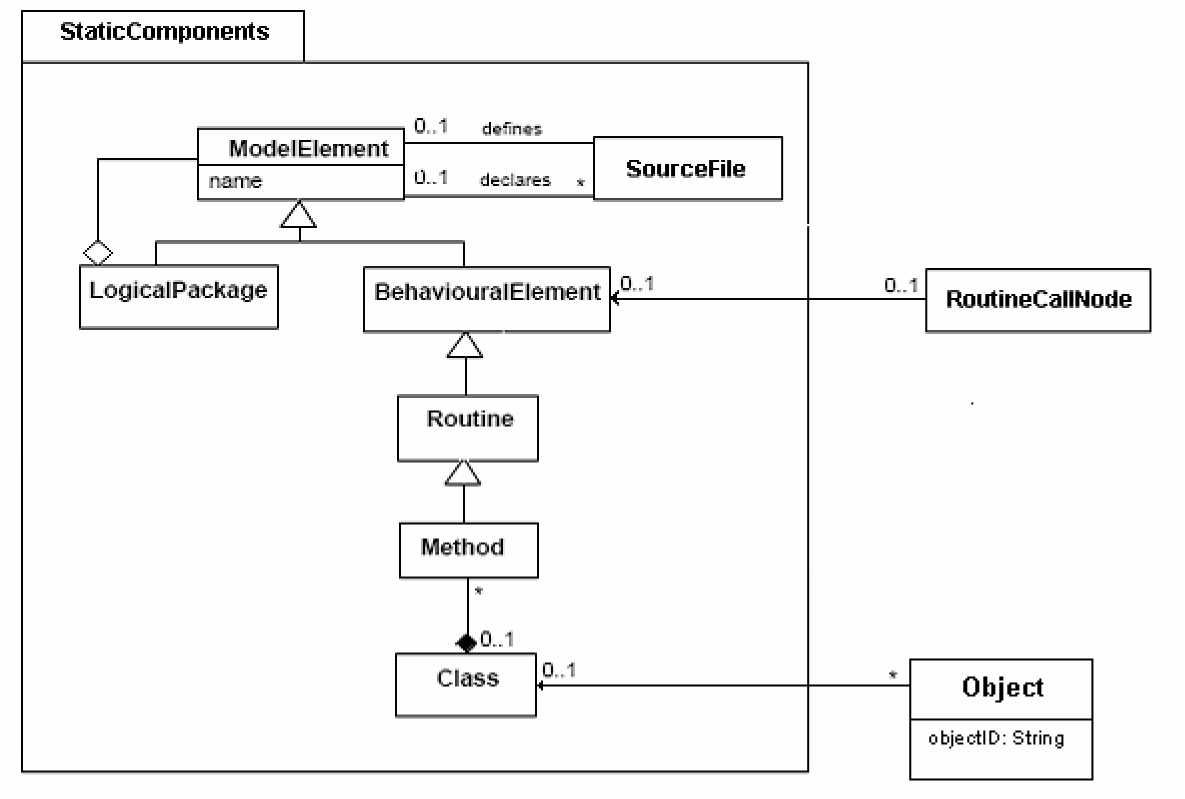


Figure 6.4. Extension of CTF to support static components

The diagram in Figure 6.4 is similar to the one described in the Dagstuhl Middle Metamodel (DMM) [Lethbridge 03], which is a model for representing the static relationships among the various components of a software system. DMM supports systems developed in most widely used procedural and object-oriented programming languages such as C, C++, and Java. The compatibility between CTF and DMM should enable these two metamodels to work together in the future. Currently, CTF neither requires nor precludes the usage of DMM.

6.4.1.7 Behavioural Patterns

Trace patterns (i.e. sequence of events repeated non-contiguously in the trace) are represented using the TracePattern class. This class contains an attribute that can be used to assign a high-level description to the pattern. The same trace pattern can occur in more than one trace. Indeed, a pattern that occurs in several traces might be more relevant than another pattern that appears in one or two traces only. Relevance, here, is defined with respect to how close the pattern is to a design concept. To capture this information, we use the PatternOccurrence class.

6.4.1.8 Illustration of CTF

To illustrate the use of CTF, let us consider an example: Suppose that the result of exercising a feature of a particular system generates the trace shown in Figure 6.5a. The trace involves three classes namely A, B and C and three objects, which are obj1, obj2 and obj3. There are four methods that have been invoked: m0, m1, m2, and m3. We notice that the call generated to obj2:B.m1 is repeated five times in the trace, probably due the existence of a loop in the source code. We also notice that this trace contains a pattern which consists of the call obj3:C.m2 as depicted clearly on the directed ordered acyclic graph of Figure 6.5b.

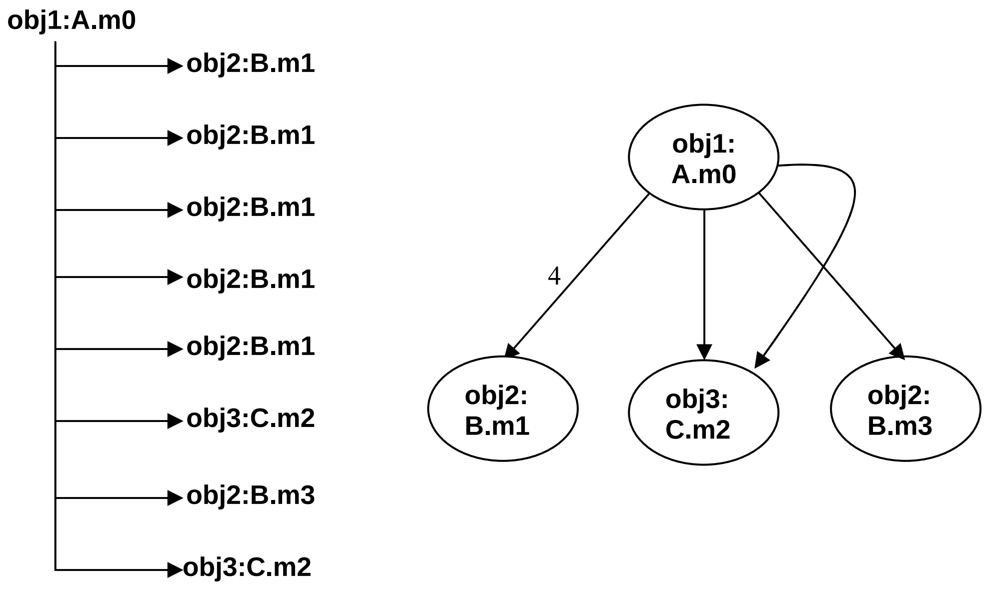


Figure 6.5. a) An example of a trace as a tree. b) The ordered directed acyclic graph corresponding to this trace

An instance diagram of the above trace using the CTF schema is shown in Figure 6.6. This diagram omits instances of the static model classes (e.g. Class, Method, etc) to avoid clutter. We imagine that the overall scenario is called “Draw Circle” (as in a drawing program); this scenario is represented with the object of class Scenario and the trace is depicted by the object of class MethodCallTrace. The nodes are represented with the objects obj1Am0, obj2Bm1, obj3Cm2, and obj2Bm3. Edges are represented using instances of the class TraceEdge. There are 4 edges. The node obj3Cm2 has two incoming edges. The software engineer using the tool can therefore mark this as a pattern, shown here as an instance of

PatternOccurrence. The user has indicated using the ‘description’ attribute that this pattern is concerned with the “Drag Mouse” operation. The instance of PatternOccurrence shows in which particular trace the pattern occurs as well as the node that initiates the pattern.

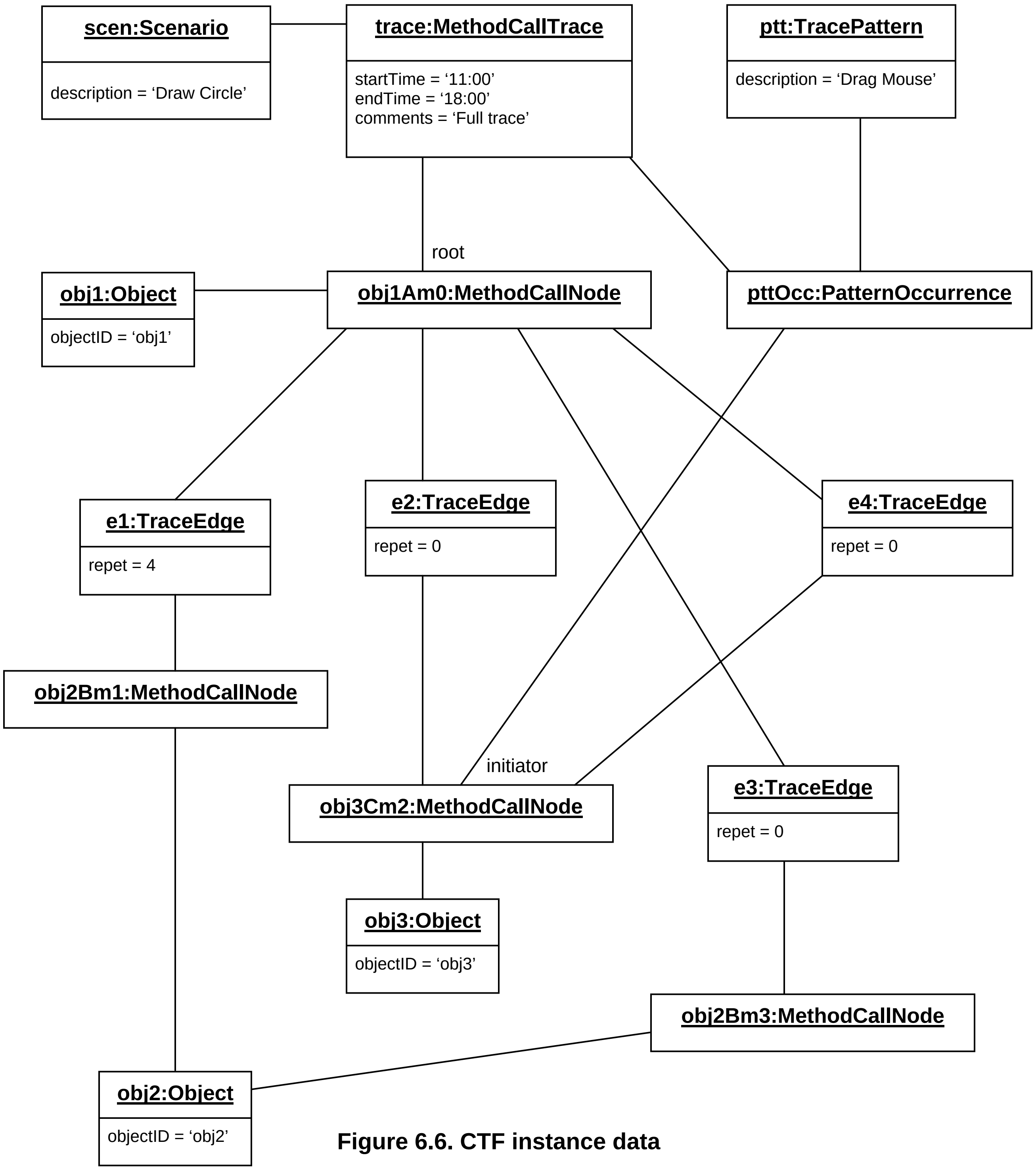


Figure 6.6. CTF instance data

6.4.2 CTF Class Description

In this section we present the semantics of the CTF metamodel. For this purpose, we use the description of the UML 2.0 metamodel as a template [UML 2.0]. The constraints on the model elements are described in OCL (Object Constraint Language) as well as in natural language.

6.4.2.1 Scenario

Semantics

Objects of the Scenario class represent the system scenario executed in order to generate the traces that need to be analysed.

Attributes

description: String	Specifies a description of the usage scenario (e.g. the name of the scenario, input data, etc).

Associations

Trace [1..*]	References the execution traces that are generated after the execution of the usage scenario. Note that one scenario can have more than one trace object that correspond to it.

Constraints

No additional constraints

6.4.2.2 Trace

Semantics

An abstract class representing common information about traces generated from the execution of the system.

Attributes

startTime: Time	Specifies the starting time of the generation of the trace.
endTime: Time	Specifies the ending time of the generation of the trace.
comments: String	Specifies comments that software engineers might need in order to describe the circumstances under which the trace is generated.

Associations

Scenario [1]	References the usage scenario that is exercised so as to generate the trace.

Constraints

[1] startTime and endTime should be different

self.endTime >= self.startTime

6.4.2.3 RoutineCallTrace (Subclass of Trace)

Semantics

An object of the RoutineCallTrace represents a trace of routine calls. A routine is defined as any function whether it is in a class or not.

Attributes

No additional attributes

Associations

PatternOccurrence [*]	References the occurrences of the execution patterns that are invoked in the trace.
root: ComprehenionUnitInitiator [1]	Specifies the root of the call tree.

Constraints

[1] The root of a trace must not have parent node

```
self.root.incoming ->isEmpty();
```

[2] The root node cannot be an object of ControlNode subclasses

```
not self.root.oclIsTypeOf(SequenceNode) and
not self.root.oclIsTypeOf(RecursionOccurrence)
```

[3] The graph needs to be an ordered directed acyclic graph.

6.4.2.4 MethodCallTrace (Subclass of RoutineCallTrace)

Semantics

An object of the MethodCallTrace represents a trace of method calls only. This class is added in case there will be a need in the future to distinguish between traces of routine (not methods) calls and traces of method calls.

Attributes

No additional attributes

Associations

No additional attributes

Constraints

No additional constraints

6.4.2.5 TracePattern

Semantics

An object of the class TracePattern represents a sequence of calls that is repeated in a non-contiguous manner in the trace.

Attributes

description: String	Specifies a textual description that a software engineer assigns to the execution pattern.

Associations

PatternOccurrence [2..*]	References the instances of the pattern in the trace.

Constraints

[1] The PatternOccurrence objects belong to the same trace (i.e. RoutineCallTrace object that contains the pattern occurrences).

6.4.2.6 PatternOccurrence

Semantics

This class represents the instances of an execution pattern.

Attributes

No additional constraints

Associations

TracePattern [1]	References the TracePattern object for which this object represents an occurrence of the pattern.
RoutineCallTrace [1]	References the Trace object where the pattern pointed to by the PatternOccurrence object appears.
initiator : ComprehensionUnitInitiator [1]	References the comprehension unit initiator (i.e. the node in the acyclic graph) that is the root of the pattern pointed to by the PatternOccurrence object.

Constraints

[1] The initiator of the PatternOccurrence object is reachable from the root of the RoutineCallTrace that contains the pattern occurrence.

6.4.2.7 ComprehensionUnitIntiator

Semantics

ComprehensionUnitInitiator is an abstract class that represent the nodes of the acyclic graph (compact form of the call tree), which are objects of the derived classes of ComprehensionUnitInitiator.

Attributes

label: String	If a static component is not specified, perhaps because it is not known, then the label can be used to indicate the node label. For example, a node label can simply represent the name of the routine represented by this comprehension unit. In Section 6.4.1.6, we discussed how CTF can be used to refer to objects that would represent classes, methods, source files, etc.
timestamps: Time []	Specifies the timestamps of the routines represented by this comprehension unit initiator.
executionTime: int [ ]	Specifies the execution time of the routines represented by this comprehension unit initiator.

Associations

DAG: RoutineCallTrace [1]	References the Trace for which this comprehension unit is the root.
PatternOccurrence [0..1]	References the pattern occurrence for which this comprehension unit is the initiator.

RecursionOccurrence [0..1]	Specifies a comprehension unit that is repeated recursively.
incoming: TraceEdge [*]	Specifies the TraceEdge objects that represent the incoming edges of this comprehension unit initiator.
outgoing: TraceEdge [*]	Specifies the TraceEdge objects that represent the outgoing edges of this comprehension unit initiator.
Thread [*]	References the Thread objects that represent the thread in which this comprehension unit is executed.

Constraints

[1] The timestamps of the routine calls represented by this comprehension unit initiator must be sorted in an ascending manner. This guarantees that the graph maintains the sequential execution of the routines.

```
self.timestamps is a sorted collection
```

[2] The parent nodes of this comprehension unit cannot be the same as its child nodes and vice-versa since the graph is acyclic.

```
self.incoming->excludesAll(self.outgoing) and

self.outgoing ->excludesAll(self.incoming)
```

6.4.2.8 TraceEdge

Semantics

Objects of the TraceEdge class represent the edges of the acyclic graph.

Attributes

repet: int	Specifies an edge label that will be used to represent the number of repetitions due to loops and recursion.

Associations

child: ComprehensionUnitInitiator [1]	References the comprehension unit initiator that represents the child node that is pointed to by the trace edge.
parent: ComprehensionUnitInitiator [1]	References the comprehension unit initiator that represents the parent node from which this edge is an outgoing edge.

Constraints

[1] The child and the parent must be different nodes. Recursion is represented using the RecursionOccurrence class (Section 6.4.1.4)

sefl.child <> self.parent

[2] The value of 'repet' must be greater than or equal to zero

self.repet >= 0

6.4.2.9 Thread

Semantics

Objects of the Thread class represent the thread of execution included in the trace.

Attributes

name: String	Specifies the name of the thread.

Associations

ComprehensionUnitInitiator [1..*]	References the ComprehensionUnitIntiator that are executed in this thread of execution.

Constraints

No additional constraints

6.4.2.10 RoutineCallNode (Subclass of ComprehensionUnitInitiator)

Semantics

Objects of the RoutinceCallNode represent the routine calls invoked in the trace.

Attributes

No additional attributes

Associations

BehaviouralElement [0..1]	References the static element that corresponds to this routine call (see Figure 6.4) if a model of the static components of the system is built. This association is an extension to CTF metamodel and can be ignored.

Constraints

No additional constraints

6.4.2.11 MethodCallNode (Subclass of ComprehensionUnitInitiator)

Semantics

Objects of the MethodCallNode represent the method calls invoked in the trace.

Attributes

No additional attributes

Associations

Object [0..1]	References the object on which the method is invoked.

Constraints

No additional constraints

6.4.2.12 Object

Semantics

This class represents the objects invoked in the trace. In some traces, information about objects may be present; in others such information (and hence instances of this class) may be absent.

Attributes

objectID: String	Specifies the object identifier.

Associations

Class [0..1]	Specifies the class that defines the object (see Figure 6.4). This association is an extension to CTF and should be used if the static components are also used.
MethodCallNode [1..*]	Specifies the methods invoked on this object.

Constraints

No additional constraints

6.4.2.13 ControlNode (Subclass of ComprehensionUnitInitiator)

Semantics

The ControlNode class is an abstract class that is used to specify additional information that can help better structure the trace.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] A control node cannot be the root of the entire trace

self.incoming ->notEmpty()

[2] A control node must have children

self.outgoing -> notEmpty()

6.4.2.14 RecursionOccurrence (Subclass of ControlNode)

Semantics

An object of the RecursionOccurrence is added to represent comprehension units that are repeated recursively. In Section 6.4.1.4, we showed how adding a new node called REC enables the removal of repetitions due to recursion.

Attributes

No additional attributes

Associations

repeatedUnit: ComprehensionUnitInitiator [1]	References the comprehension unit initiator that is repeated recursively.

Constraints:

No additional constraints

6.4.2.15 SequenceNode (Subclass of Control Node)

Semantics

An object of the SequenceNode class is added to represent multiple comprehension units that are repeated in a contiguous way (see Section 6.4.1.4).

Attributes

No additional attributes

Associations

No additional associations

Constraints:

No additional constraints

6.4.3 CTF Syntactic Form

CTF instance data can be conveyed using a syntactic form that supports the representation of graph structures. In this thesis, we do not attempt to discuss the advantages and disadvantages of all existing syntactic forms. We present how GXL and TA can be used with CTF. The choice of these languages is due to the fact that they are widely used in the reverse engineering research community. In addition, both languages support the exchange of the instance data as well as the metamodel; this is compliant with the completeness requirement discussed in Section 6.3.7.

GXL [Holt 00] is one candidate for the syntactic carrier for CTF. A GXL file consists of XML elements for describing nodes, edges, attributes, etc. It was designed to supersede a number of pre-existing graph formats such as GraX [Ebert 99], TA [Holt 98], and RSF [Müller 88]. GXL has been widely adopted as a standard exchange format for various types of graphs by both industry and academia.

However, a GXL representation of CTF would tend to be much larger than necessary due to the use of XML tags and the explicit need to express the data as GXL nodes and edges. The compactness benefits of CTF would therefore be partially cancelled out by representing it using GXL, as noted by Brown et al. [Brown 02] and Chilimbi et al. [Chilimbi 00]. Whereas the wordiness of GXL would not be a problem when expressing moderately sized graphs in other domains, the sheer hugeness of traces suggests an alternative might be appropriate.

Figure 6.7 shows an excerpt of a GXL file that represents the CTF example of Figure 6.6 (note that GXL and TA do not have a built-in data type for representing timing information, we used a string instead for simplicity reasons). The complete GXL file can be found in Appendix A.

```
<gxl>
 <graph>
   <node id = “scen”>
      <attr name = “description”>
            <string> Draw circle </string>
      </attr>
   </node>
   <node id = “trace”>
       <attr name = “startTime”>
            <string> 11:30 </string> </attr>
       <attr name = “endTime”>
            <string> 18:30 </string> </attr>
       <attr name = “comments”>
            <string> Full trace </string>
       </attr>
   </node>
   <node id = “obj1Am0”> </node>
   <node id = “obj1”>
       <attr name = “objectID”>
             <string> obj1 </string> </attr>
   </node>
   <node id = “e1”>
         <attr name = “repet”>
                <int> 4 </int>
          </attr>
       <node>

        REMIANING NODES.

       <edge from = “scen” to = “trace”>
       </edge>
       <edge id = “root”
             from = “trace” to = “obj1Am0”>
       </edge>
       <edge from = “obj1Am0” to = “obj1”>
        </edge>
             REMAINING EDGES
        </graph>
   <gxl>
```

Figure 6.7. GXL representation of the CTF instance data of Figure 6.6

One reasonable alternative to GXL is TA [Holt 98], which would substantially reduce the space required by a CTF trace. The TA language was originally developed to help visualize information about software systems. It has been used as a model interchange format in several contexts [Holt 97]. Based on RSF [Müller 88], TA retains the basic 3-tuple of space-separated text strings to record information about the static aspect of the system, called facts. However, it extends RSF by supporting the capability to add attributes to nodes and arcs. It also supports the exchange of the metamodel. TA files consist of two parts: A part that is used to specify the metamodel and it is called the *scheme* section, and a section that is used to specify the data to exchange referred to as the *fact* section. Figure 6.8 shows an excerpt of the Fact section that would correspond to the CTF example of Figure 6.6. The complete TA file representing can be found in Appendix A.

In order to illustrate the size requirement for a trace represented in GXL as opposed to TA, we computed the size of the GXL and TA files presented in Appendix A. The GXL file contains 1810 characters (excluding blanks) whereas the TA file contains 881 characters only, which represent almost half the size of the GXL file. This result shows that GXL is indeed a verbose language and that TA could be an appropriate alternative.

```
FACT TUPLE:
  $INSTANCE  scen Scenario
     $INSTANCE trace MethodCallTrace
     $INSTANCE obj1Am0 MethodCallNode
     $INSTANCE obj1 Object
     $INSTANCE e1 TraceEdge
     Remaining nodes
     link scen trace
     link trace obj1Am0
     link obj1Am0 obj1

     Remaining edges
FACT ATTRIBUTE:
     scen (description = “Draw Circle”)
     trace (startTime = “11:00”   endTime = “18:00”   comments =”Full trace”)

     The remaining attributes for nodes and edges should be entered here
```

Figure 6.8. TA representation of the CTF instance data of Figure 6.6

6.5 Adoption of CTF

CTF is the exchange format used by SEAT (Software Exploration and Analysis Tool), the trace analysis tool built in the University of Ottawa [Hamou-Lhadj 04c, Hamou-Lhadj 05a]. The tool manipulates traces in CTF and displays them using a tree widget. To help the user extract useful information from a trace, SEAT implements several trace filtering techniques such as the detection of utilities, application of matching criteria, detection of patterns, etc. Using SEAT, an analyst can reduce the size of a trace to the level where he or she can understand important aspects of its structure.

Moreover, a CTF API has been created and delivered to software engineers of QNX Software Systems. The API contains the main functions to create and manipulate CTF components

In order to motivate the use of CTF in academia, we have presented it in various conferences including ATEM 2003, ECOOP 2004, IWPC 2004, and IWPC 2005. So far, there has been an interest in using CTF from the following university research groups:

- Members of LORE (Lab On Reengineering) of the University of Antwerp, Belgium
- Members of the Software Composition Group of the University of Bern, Switzerland
- Members of the Knowledge-Based Reverse Engineering Research Group, University of Ottawa.

6.6 Summary

A common exchange format is important for allowing different tools to share data. In this chapter, we presented CTF (Compact Trace Format) a schema for representing traces of routine calls.

To deal with the vast size of typical traces, we designed CTF based on the idea that dynamic call trees can be turned into ordered directed acyclic graphs, where repeated subtrees are factored out.

CTF supports traces defined at different levels of abstraction including object, class and package level. It also supports the specification of threads of execution. Additional information such as timestamps and routine execution time are added to enable profilers to use CTF.

CTF, as described in this thesis, is a schema. Trace data conforming to CTF can be expressed using GXL, TA, or any other data 'carrier' language. However, we suggest using a compact representation since doing otherwise would somewhat defeat the compactness objective of CTF.

An algorithm for the on the fly generation of CTF-based traces is based on the algorithm presented in Section 3.6 and that aims to convert a tree into a DAG in an efficient way. This algorithm also supports various matching criteria that can be used to consider similar but not necessarily identical subtrees.

CTF is lossless, i.e. the original trace can be reconstructed, only when the simplest matching criterion is used: Two sequences of any length of calls to the same routine are considered identical.

Chapter 7. Conclusions

To be successful, software maintenance requires efficient program comprehension techniques. Similar to using static analysis for understanding the static aspects of the system, dynamic analysis focuses on helping software engineers understand the dynamics of a program.

Run-time information is often represented in the form of execution traces such as the ones based on routine calls. However, coping with the large size of typical traces is a challenging task, which has led to the creation of several trace analysis tools and techniques.

Our contributions consist of a set of simple yet efficient techniques that, if implemented in tools, should make the analysis of traces considerably easier.

The following section explains these contributions in more detail.

7.1 Research Contributions

Survey of trace analysis tools and techniques: We studied the techniques supported by eight trace tools in order to extract the most useful solutions to the problem of efficiently manipulating traces. The results of this study can help tool users to select the right tools that would fit their needs, and the tool builders to understand existing features in order to prevent reinventing them. A detailed qualitative discussion of the advantages and limitations of existing techniques is also provided.

Trace Metrics: We presented several metrics that aim to measure properties of traces. These metrics, once implemented in tools, are intended to help software engineers explore traces more easily; they can, for example be used to guide the user towards parts of traces that have greater complexity. Most existing tools do not offer any such guidance, making tool features such as filtering less useful. We experimented with the metrics by applying them to several execution traces of three different software systems. We believe that this can help researchers gain a good understanding of what makes trace complex and therefore build better trace analysis techniques.

Algorithm to convert the tree structure of a trace to a graph: We presented an algorithm for the on the fly generation of traces. Using this algorithm, the traces will not need to be saved as tree structures. We also showed how various matching criteria can be supported by this algorithm.

Trace Summarization: We presented a new concept for exploring traces based on summarizing their content. The objective is to enable top-down analysis of the trace as well as recovery of behavioural design models of systems. We discussed how trace summarization is related to text summarization. Our approach for trace summarization is based on filtering the content of traces by removing implementation details. For this purpose, we presented a definition of the concept of implementation details including utilities. We also presented a metric that aims to rank the system components according to their utilityhood. To detect utility components, we used fan-in analysis. A case study was presented where we generated a summary from a large trace generated from the Weka system. The summary was given for evaluation to the developers of the Weka system. Most of these developers agreed that the summary is a good high-level representation of the traced scenario. They also supported the idea of generating summaries to help understand the dynamics of a poorly documented system.

The Compact Trace Format: A lack of an exchange format for sharing traces hinders interoperability among tools. We addressed this issue by developing a common exchange format called CTF (Compact Trace Format). CTF is designed with the idea of scalability in mind and makes use of the fact that tree structures can be turned into directed acyclic graphs. The design of CTF is compliant with well-known requirements for a standard exchange format. The CTF metamodel was presented along with the semantics of its components. We argued that developing a metamodel should be independent from the syntactic form that needs to carry the instance data. However, we discussed how GXL or TA can be used with

CTF. CTF has been presented in various conferences and is now the official exchange format of a trace analysis tool called SEAT (Software Exploration and Analysis Too) developed at the University of Ottawa.

7.2 Opportunities for Further Research

In this section, we discuss areas in need of additional research effort:

7.2.1 Further Formalizing of the Concepts Presented in this Thesis

One direction of future work is to increase the level of formalization of certain concepts presented in this thesis. A formal language could be developed to better express the techniques used in various tools for analyzing trace content. Such a language could enable better evaluation of existing trace analysis tools, the subject of Chapter 2. The long-term objective is to develop a better theoretical ground for trace analysis.

7.2.2 Trace Metrics

One way to refine the metrics presented in Chapter 3 is to study the different matching criteria for the Scu_{sim} metric. The goal is to have tools that suggest combining different matching criteria automatically in order to further reduce the complexity of the trace, or part of the trace, under study.

Additional metrics can be developed as well. For example, we can combine trace metrics with static complexity metrics to better estimate the complexity of the content of a trace.

Metrics involving multiple traces are also an area of future research that can be very useful for understanding the dynamics of the system. For example, it might be interesting to measure to what extent multiple traces use the same set of routines, or have the same or similar comprehension units.

Additionally, the concept of entropy from information theory can be used to suggest areas of a trace that are more complex. Therefore, a useful avenue of investigation would be to develop trace metrics based on entropy.

Finally, it would be useful to investigate how trace metrics can be supported by tools. The objective is to allow software engineers exploring traces to easily spot parts of a trace that implement a complex behaviour. Color coding techniques are one possible visualization technique that can be used for this purpose. Tools should also be able to automatically suggest filtering techniques that can be used to reduce the complexity of traces by measuring ahead of time various properties of the trace.

7.2.3 The Concept of Implementation Details

Our definition of the concept of utilities relies mainly on the brainstorming session conducted at QNX Software System. There is definitely a need to further explore this concept and investigate how it is used in various contexts in order to have a more precise definition.

During the writing of the last chapters of this thesis, Zaidam et al [Zaidam 05] presented a technique that uses webmining principles for uncovering important classes in a system's architecture. Their approach is similar to the way search engines rank web pages according to their importance (i.e. using the concept of authorities and hubs). We think that this can also apply to traces in order to identify important components. There is definitely a need to further explore this concept and compare it to the techniques presented in this thesis (i.e. utilityhood metrics, etc).

We also need to identify categories of implementation details beyond those presented in Chapter 4 (e.g. accessing methods, constructor). Examples of such categories include components that implement data structures; mathematical functions, components that implement input/output operations, and so on.

The fan-in analysis technique for detecting utilities can be fine-tuned by considering the scope attribute of the system components when computing the utilityhood metric.

Finally, design conventions including naming conventions, comments, etc can also be used to detect utilities. For example, in many systems that we have studied, we found that they contain namespaces named using the word "Utils".

7.2.4 The Trace Summarization Process

One direction for future work would be to have the system automatically or semi-automatically suggest appropriate settings for the trace summarization algorithm based on the nature of the trace, as well as the current goals and experience of the maintainer. A key setting to investigate is the exit condition (i.e. when to stop the summarization process). Machine learning could be employed to help tune the settings by learning over time from the adjustments maintainers make.

There is also a need to investigate how better various artefacts of the system can be used to classify components as utilities or as important components. Our trace summarization algorithm uses the routine names only. It might be useful in the future to investigate other sources of information such as the method parameters, the source code comments, etc.

Another important aspect is related to the fact that fan-in analysis as presented in the thesis uses the static call graph whose edges have the same weight. One direction of future work would be to investigate appropriate weighting functions.

There is also a need to understand the matching criteria that can help with generalization, and therefore lead to more compact summaries.

Techniques for evaluating summaries are also needed. In Chapter 4, we presented a questionnaire based evaluation that addresses some aspects of the extracted summaries. It would be useful to investigate other aspects such as what would be an appropriate size of a summary for different maintenance tasks (e.g. fixing defects, adding features, etc), and different types of software engineers.

Finally, we need to investigate if and how text summarization techniques can be applied to trace summarization.

7.2.5 Extending The Compact Trace Format

While CTF covers a significant gap in terms of exchanging traces of routine calls, dynamic analysis is a highly versatile process that has a large number of needs including needs for dynamic information that is not necessarily supported by CTF. In Chapter 6, we mentioned how CTF can be used to support the dynamic presentation of multithreaded traces. However, CTF captures only the threads being executed and does not show how these threads communicate among each other. In addition to this, CTF is not designed to support statement-level traces although we expect that this is something that could be easily added in the future.

In Chapter 6, we showed how GXL and TA can be used to carry the data represented in CTF. Future work needs to investigate which one of the various syntactic form languages is the most efficient to use with CTF.

Finally, we need to work more towards the adoption of CTF by tool builders in industry as well as academia.

7.2.6 Exploring Trace Patterns

Trace patterns can play an important role in understanding a trace at higher levels of abstraction. They can also be useful to enable generalization during the trace summarization process. The idea is that one can try to understand an instance of a pattern as a whole functional behaviour instead of trying to understand every single routine invoked. Software engineers can also replace trace patterns with textual descriptions resulting in a more understandable call tree. Such a view will be of significant help to speed up the trace comprehension process. Future research needs to investigate the concept of trace patterns in more detail, to propose heuristics for classifying patterns that represent high-level concepts from the ones that are mere implementation details.

7.2.7 Program Comprehension Models

Program comprehension models such as top-down and bottom-up (see Section 2.2.2) have been built using a static representation of the system only. We need to investigate how software engineers would comprehend programs if they were given static and dynamic views of the system. The results of this study might suggest new program comprehension models, or enhance existing ones. However, we believe that using dynamic analysis views can only be practical if they are simplified, which reiterates the importance of trace summarization presented in this thesis.

7.2.8 A Tool Suite

The techniques presented in this thesis need to be integrated with trace analysis tools. We need to investigate how existing visualization schemes can be used to support these techniques.

7.3 Closing Remarks

Despite the many benefits that program comprehension can gain from applying dynamic analysis techniques, researchers have the tendency to turn into static analysis techniques paying little to no attention to the study of the behavioural aspects of a software system. This is mainly attributed to the fact that the large size of execution traces constitutes a serious obstacle to using traces in practice. In addition to this, understanding a software feature might require the analysis of several relates traces which makes techniques for simplifying traces very useful. We hope that the ones presented in this thesis will reduce the impact of this problem and enable dynamic analysis techniques to play an important role in software maintenance and program comprehension in particular.

Appendix A: GXL and TA Representations of CTF Traces

The following file represents the GXL representation of the CTF trace presented in Section 6.4.1.8.

```
<gxl>
 <graph>
  <node id = “scen” type = "Scenario">
   <attr name = “description”>
    <string> Draw circle </string>
   </attr>
  </node>
  <node id = “trace” type = "MethodCallTrace">
   <attr name = “startTime”>
    <string> 11:30 </string> </attr>
   <attr name = “endTime”>
    <string> 18:00 </string> </attr>
  <attr name = “comments”>
   <string> Full trace </string>
  </attr>
 </node>
 <node id = “ptt” type = "TracePattern">
  <attr name = “description”>
   <string> Drag Mouse </string>
  </attr>
 </node>
 <node id = “pttOcc” type = "PatternOccurrence"></node>
 <node id = “obj1Am0” type = "MethodCallNode"> </node>
 <node id = “obj2Bm1” type = "MethodCallNode"> </node>
 <node id = “obj2Bm4” type = "MethodCallNode"> </node>
 <node id = “obj3Cm2” type = "MethodCallNode"> </node>
 <node id = “obj1” type = "Object">
  <attr name = “objectID”>
   <string> obj1 </string> </attr>
 </node>
 <node id = “obj2” type = "Object">
  <attr name = “objectID”>
   <string> obj2 </string> </attr>
 </node>
 <node id = “obj3” type = "Object">
  <attr name = “objectID”>
   <string> obj3 </string> </attr>
 </node>
 <node id = “e1” type = "TraceEdge">
  <attr name = “repet”>
   <int> 4 </int>
  </attr>
 </node>
 <node id = “e2” type = "TraceEdge">
  <attr name = “repet”>
   <int> 0 </int>
  </attr>
 </node>
 <node id = “e3” type = "TraceEdge">
  <attr name = “repet”>
   <int> 0 </int>
  </attr>
 </node>
 <node id = “e4” type = "TraceEdge">
  <attr name = “repet”>
   <int> 0 </int>
  </attr>
 </node>
 <edge from = “scen” to = “trace”> </edge>
 <edge id = “root” from = “trace” to = “obj1Am0”>
 </edge>
 <edge from = “obj1Am0” to = “obj1”> </edge>
 <edge from = “obj2Bm1” to = “obj2”> </edge>
 <edge from = “obj2Bm4” to = “obj2”> </edge>
 <edge from = “obj3Cm2” to = “obj3”> </edge>
 <edge from = “obj1Am0” to = “e1”> </edge>
<edge from = “obj1Am0” to = “e2”> </edge>
 <edge from = “obj1Am0” to = “e3”> </edge>
 <edge from = “obj1Am0” to = “e4”> </edge>
 <edge from = “trace” to = “pttOcc”> </edge>
 <edge from = “ptt” to = “pttOcc”> </edge>
 <edge id = "initiator" from = “pttOcc” to =
  “obj3Cm2”> </edge>
 <edge from = “e1” to = “obj2Bm1”> </edge>
 <edge from = “e2” to = “obj3Cm2”> </edge>
 <edge from = “e3” to = “obj2Bm3”> </edge>
<edge from = “e4” to = “obj3Cm2”> </edge>

     </graph>
<gxl>
```

The following file represents the TA representation of the CTF trace presented in Section 6.4.1.8.

```
FACT TUPLE:

$INSTANCE  scen Scenario
$INSTANCE trace MethodCallTrace
$INSTANCE ptt TracePattern
$INSTANCE pttOcc PatternOccurrence
$INSTANCE obj1Am0 MethodCallNode
$INSTANCE obj2Bm1 MethodCallNode
$INSTANCE obj2Bm4 MethodCallNode
$INSTANCE obj3Cm2 MethodCallNode
$INSTANCE obj1 Object
$INSTANCE obj2 Object
$INSTANCE obj3 Object
$INSTANCE e1 TraceEdge
$INSTANCE e2 TraceEdge
$INSTANCE e3 TraceEdge
$INSTANCE e4 TraceEdge

link scen trace
link trace obj1Am0
link trace pttOcc
link obj1Am0 obj1
link obj3Cm2 obj3
link obj2Bm1 obj2
link obj2Bm4 obj2
link obj1Am0 e1
link obj1Am0 e2
link obj1Am0 e3
link obj1Am0 e4
link ptt pttOCc
link pttOcc obj3Cm2
link e1 obj2Bm1
link e2 obj3Cm2
link e3 obj2Bm3
link e4 obj3Cm2
```

```
FACT ATTRIBUTE:

scen (description = “Draw Circle”)
trace (startTime = “11:00”   endTime = “18:00”
comments =”Full trace”)
e1 (repet = 4)
e2 (repet = 0)
e3 (repet = 0)
e4 (repet = 0)

obj1 (objectId = "obj1")
obj2 (objectId = "obj2")
obj3 (objectId = "obj3")

(trace obj1Am0) (name = "root")
(pttOcc obj3Cm2) (name = "initiator")
```

Bibliography

Amyot 02	Amyot, D., Mussbacher, G., and Mansurov, N., "Understanding Existing Software with Use Case Map Scenarios", In *Proceedings of the 3rd SDL and MSC Workshop*, LNCS 2599, pages 124-140, 2002
Anquetil 03	Anquetil N. and Lethbridge T. C., "Comparative study of clustering algorithms and abstract representations for software remodularization", *IEE Proceedings - Software,* Volume 150, Number 3, pages 185-201, 2003
Anquetil 98	Anquetil N. and Lethbridge T. C., "Assessing the Relevance of Identifier Names in a Legacy Software System", In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, IBM Press, pages 213-222, 1998
ANSI/IEEE Std	ANSI/IEEE Standard 729-1983
ANTLR	Another Tool for Language Recognition (www.antlr.org)
Bacon 96	Bacon D. F. and Sweeney P. F., "Fast static analysis of C++ Virtual function calls", In *Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM Press, pages 324-341, 1996
Ball 99	Ball T., "The Concept of Dynamic Analysis", In *Proceedings of the 7th European Software Engineering Conference,* Springer-Verlag, pages 216-234, 1999
Basili 94	Basili V. R., Caldiera G., and Rombach H. D., "Goal Question Metric Paradigm", In *Encyclopaedia of Software Engineering,* John Wiley & Sons, pages 528-532, 1994
Baxendale 58	Baxendale P., "Machine-made index for technical literature – an experiment", *IBM Journal of Research and Development, Volume:2*, pages 354-361, 1958

Biggerstaff 89	Biggerstaff T. J., “Design Recovery for Maintenance and Reuse”, *IEEE Computer,* Volume 22, Issue 7, IEEE Computer Society, pages 36-49, 1989
Bowman 99	Bowman I. T., Godfrey M. W., and Holt R. C., “Connecting Architecture Reconstruction Frameworks”, *Journal of Information and Software Technology*, Volume 42, Number 2, Elsevier Science, pages 91-102, 2000
Boyer 97	Boyer R. S., and Moore J. S., “A Fast Searching Algorithm”, *Communications of the ACM*, Volume 20, Issue 10, pages 761-772, 1997
Brooks 83	Brooks R., “Towards a Theory of the Comprehension of Computer Programs”, *International Journal of Man-Machine Studies,* Volume 18, Number 6, 1983
Brown 02	Brown R., Driesen K., Eng D., Hendren L., Jorgensen J., Verbrugge C., and Wang Q., **“**STEP: a framework for the efficient encoding of general trace data”**,** In *Workshop on Program Analysis for Software Tools and Engineering*, ACM Press, pages 27 – 34, 2002
Chan 03	Chan A., Holmes R., Murphy G. C., and Ying A. T. T., “Scaling an Object-Oriented System Execution Visualizer through Sampling”, In *Proceedings of the 11th International Workshop on Program Comprehension*, IEEE Computer Society, pages 237-244, 2003
Chapin 88	Chapin N., “Software Maintenance Life Cycle”, In *Proceedings of the Conference on Software Maintenance*, IEEE Computer Society, pages 6-13, 1988
Checkstyle	Checkstyle System. http://checkstyle.sourceforge.net/
Chikofsky 90	Chikofsky E. J., Cross J. H., "Reverse Engineering and Design Recovery: A Taxonomy", *IEEE Software,* Volume 7, Issue 1, IEEE Computer Society, pages 13-17, 1990
Chilimbi 00	T. Chilimbi, R. Jones, and B. Zorn, “Designing a trace format for heap allocation events”, In *Proceedings of the International Symposium on Memory Management*, ACM Press, pages 35-49, 2000

De Pauw 02	De Pauw W., Jensen E., Mitchell N., Sevitsky G., and Vlissides J., Yang J., "Visualizing the Execution of Java programs", In *Proceedings of the International Seminar on Software Visualization,* LNCS 2269, Springer-Verlag, pages 151-162, 2002
De Pauw 93	De Pauw W., Helm R., Kimelman D., and Vlissides J., "Visualizing the Behaviour of Object-Oriented Systems", In *Proceedings of the 8th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM Press, pages 326-337, 1993
De Pauw 94	De Pauw W., Kimelman D., and Vlissides J., "Modelling Object-Oriented Program Execution", In *Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP)*, LNCS 821, Springer-Verlag, pages 163-182, 1994
De Pauw 98	De Pauw W., Lorenz D., Vlissides J., and Wegman M., "Execution Patterns in Object-Oriented Visualization", In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems*, pages 219-234, 1998
Dean 95	J. Dean, D. Grove, and Chambers, "Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis", In *Proceedings of the 9th European Conference on Object-Oriented Programming,* LNCS 952, Springer-Verlag, pages 77-101, 1995
Downey 80	Downey J.P., Sethi R., and Tarjan R.E., "Variations on the Common Subexpression Problem", *Journal of the ACM,* Volume 27, Issue 4, pages 758-771, 1980
Ebert 99	Ebert J., Kullbach B., and Winter A., "GraX – An Interchange Format for Reengineering Tools", In *Proceedings of the 6th Working Conference on Reverse Engineering*, IEEE computer Society, pages 89–98, 1999
Edmunsdon 69	Edmundson H., "New methods in automatic extracting", *Journal of the ACM*, Volume 6 Issue 2, pages 264-285, 1969
Eisenbarth 01	Eisenbarth T., Koschke R., and Simon D., "Feature-Driven Program Understanding using Concept Analysis of Execution Traces", In *Proceedings of the 9th International Workshop on Program Comprehension*, IEEE Computer Society, pages 300-309, 2001
Erdos 98	Erdos K. and Sneed .M., "Partial Comprehension of Complex Programs (enough to perform maintenance)", In *Proceedings of the 6th International Workshop on Program Comprehension*, IEEE

computer Society, pages 98-105, 1998

Flajolet 90 Flajolet P., Sipala P., and Steyaert J.–M., “Analytic Variations on the Common Subexpression Problem”, In *Proceedings of the 7th International Colloquium on Automata, Languages and Programming*, Springer-Verlag, pages 220-234, 1990

Hamou-Lhadj 02 Hamou-Lhadj A. and Lethbridge T., “Compression Techniques to Simplify the Analysis of Large Execution Traces”, In *Proceedings of the 10th International Workshop on Program Comprehension*, IEEE Computer Society, pages 159-168, 2002

Hamou-Lhadj 03a Hamou-Lhadj A. and Lethbridge T., “Techniques for Reducing the Complexity of Object-Oriented Execution Traces”, In *Proceedings of the 2nd ICSM International Workshop on Visualizing Software for Understanding and Analysis,* pages 35-40, 2003

Hamou-Lhadj 03b Hamou-Lhadj A. and Lethbridge T., “An Efficient Algorithm for Detecting Patterns in Traces of Procedure Calls”, In *Proceedings of the 1st ICSE International Workshop on Dynamic Analysis*, pages 33-36, 2003

Hamou-Lhadj 04a Hamou-Lhadj A. and Lethbridge T., “A Survey of Trace Exploration Tools and Techniques”, In *Proceedings of the 14th IBM Conference of the Centre for Advanced Studies on Collaborative Research*, IBM Press, pages 42-55, 2004

Hamou-Lhadj 04b Hamou-Lhadj A. and Lethbridge T., “Reasoning about the Concept of Utilities”, In *Proceedings of the 1st ECOOP International Workshop on Practical Problems of Programming in the Large*, LNCS 3344, Springer-Verlag, pages 10-22, 2005

Hamou-Lhadj 04c Hamou-Lhadj A., Lethbridge T., and Fu L., “Challenges and Requirements for an Effective Trace Exploration Tool”, In *Proceedings of the 12th International Workshop on Program Comprehension,* IEEE Computer Society, pages 70-58, 2004

Hamou-Lhadj 04d Hamou-Lhadj A. and Lethbridge T., “A Metamodel for Dynamic Information Generated from Object-Oriented Systems”, In *Proceedings of the First International Workshop on Meta-models and Schemas for Reverse Engineering,* Electronic Notes in Theoretical Computer Science Volume 94, pages 59-69, 2004

Hamou-Lhadj 05a Hamou-Lhadj A., Lethbridge T., and Fu L., "SEAT: A Usable Trace Analysis Tool", In *Proceedings of the 13th International Workshop on Program Comprehension*, IEEE Computer Society, pages 157-160, 2005

Hamou-Lhadj 05b	Hamou-Lhadj A. and Lethbridge T., “Measuring Various Properties of Execution Traces to Help Build Better Trace Analysis Tools”, In *Proceedings of the 10th International Conference on Engineering of Complex Computer Systems*, IEEE Computer Society, pages 559–568, 2005
Hamou-Lhadj 05c	Hamou-Lhadj A., Braun E., Amyot D., and Lethbridge T., “Recovering Behavioral Design Models from Execution Traces”, In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering,* IEEE Computer Society, pages 112-121, 2005
Hamou-Lhadj 05d	Hamou-Lhadj A. and Lethbridge T., “Trace Summarization: A Novel Technique for Recovering Design Models”, *Submitted to IEEE Transactions on Software Engineering, Special Issue: Interaction and State-Based Modelling*, 2005
Hamou-Lhadj 05e	Hamou-Lhadj A. and Lethbridge T., “A Metamodel for the Compact but Lossless Exchange of Execution Traces”, *Submitted to the Journal of Software and Systems Modeling*, 2005
Holt 00	Holt R. C., Winter A., and Schürr A., “GXL: Toward a Standard Exchange Format”, In *Proceedings of the 7th Working Conference on Reverse Engineering,* IEEE Computer Society, pages 162-171, 2000
Holt 97	Holt R. C., “Software Bookshelf: Overview and construction”, http://www-turing.cs.toronto.edu/pbs/papers/bsbuild.html, 1997
Holt 98	Holt R. C., “An Introduction to TA: The Tuple Attribute Language”, http://swag.uwaterloo.ca/pbs/papers/ta.html
Hyades	Hyades Project: http://www.eclipse.org/tptp/
Jax	Jax: http://www.alphaworks.ibm.com/formula/Jax
Jerding 97a	Jerding D. and Rugaber S., "Using Visualisation for Architecture Localization and Extraction", In *Proceedings of the 4th Working Conference on Reverse Engineering*, IEEE Computer Society, pages 56-65, 1997
Jerding 97b	Jerding D., Stasko J. and Ball T., “Visualizing Interactions in Program Executions”, In *Proceedings of the International Conference on Software Engineering*, ACM Press, pages 360-370, 1997

Jin 02	Jin D., Cordy J. R., Dean T. R., "Where is the Schema? A Taxonomy of Patterns for Software Exchange", In *Proceedings of the 10th International Workshop on Program Comprehension*, IEEE Computer Society, pages 65-74, 2002
Jing 98	Jing H. R., McKeown K., and Elhadad M., "Summarization evaluation methods: Experiments and analysis", In *Working Notes of the AAAI Spring Symposium on Intelligent Text Summarization*, pages 60-68, 1998
Johnson 01	Johnson E. E., Ha J., and Baqar Zaidi M., "Lossless Trace Compression", *IEEE Transactions on Computers*, Volume 50, Issue 2, pages 158-173, 2001
Jones 98	Jones K. S., "Automatic summarising: factors and directions", In *Advances in Automatic Text Summarization,* MIT Press, pages 1-14, 1998
Jorgensen 95	Jorgensen M., "An Empirical Study of Software Maintenance Tasks", *Journal of Software Maintenance*, Volume 7, Issue 1, pages 27-48, 1995
King 98	Jing H. R., McKeown K., and Elhadad M., "Summarization evaluation methods: Experiments and analysis", In *Working Notes of the AAAI Spring Symposium on Intelligent Text Summarization*, pages 60-68, 1998
Korel 97	Korel B., Rilling J., "Dynamic Program Slicing in Understanding of Program Execution", In *Proceedings of the 5th International Workshop on Program Comprehension*, IEEE Computer Society, pages 80-89, 1997
Koskimies 96a	Koskimies K. and Mössenböck H., "Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs", In *Proceedings of the 18th International Conference on Software Engineering*, ACM Press, pages 366-375, 1996
Koskimies 96b	Koskimies K., Männistö T., Systä T., and Tuomi J., "SCED: A Tool for Dynamic Modeling of Object Systems", *University of Tampere, Dept. of Computer Science, Report A-1996-4*, 1996
Lange 97	Lange D. B. and Nakamura Y., "Object-Oriented Program Tracing and Visualization", *IEEE Computer*, Volume 30, Issue 5, pages 63-70,

	1997
Larus 99	Larus J. R., “Whole program paths”, In *Proceedings of the Conference on Programming Language Design and Implementation*, ACM Press, pages 259-269, 1999
Leduc 04	J. Leduc, “Towards Reverse Engineering of UML Sequence Diagrams of Real-Time, Distributed Systems through Dynamic Analysis”, *Master’s Thesis of Applied Science*, Carleton University, 2004
Lee 97	Lee H. B., Zorn B. G., “BIT: A tool for Instrumenting Java Bytecodes”, In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 73-82, 1997
Lethbridge 01	Lethbridge C. T. and Laganière R., Object-Oriented Software Engineering: Practical Software Development using UML and Java, McGraw Hill, 2001
Lethbridge 03	Lethbridge C. T., Tichelaar S., and Ploedereder E. “The Dagstuhl Middle Metamodel: A Schema For Reverse Engineering”, In *Proceedings of the First International Workshop on Meta-models and Schemas for Reverse Engineering*, Electronic Notes in Theoretical Computer Science Volume 94, pages 7-8, 2004
Lethbridge 97	Lethbridge T. C. and Anquetil N., “Architecture of a Source Code Exploration Tool: A Software Engineering Case Study”, *Computer Science Technical Report TR-97-07,* University of Ottawa, 1997
Lunh 58	Lunh H., “The Automatic Creation of Literature Abstracts”, *IBM Journal of Research and Development,* Volume 2, Issue 2, pages 159-165, 1958
MOSAIC	National Center for Supercomputing Applications. “NCSA Mosaic Home Page.” http://www.ncsa.uiuc.edu/SDG/Software/ Mosaic/NCSAMosaicHome.html.
Müller 88	Müller H. A., Klashinsky K., “Rigi – A System for Programming in-the-large”, In *Proceedings of the 10th International Conference on Software Engineering*, ACM Press, pages80-86, 1988
Müller 93	Müller H. A., Orgun M. A., Tilley S. R., and Uhl J. S., “A Reverse Engineering Approach to Subsystem Structure Identification”, *Journal of Software Maintenance: Research and Practice*, Volume 5, Issue 4,

pages 181-204, 1993

Murphy 97 — Murphy G. C., and Notkin D., “Reengineering with reflexion models: A case study”, *IEEE Computer,* Volume 30, Issue 8, pages 29-36, 1997

Pacione 03 — Pacione M. J., Roper M., and Wood M., “A Comparative Evaluation of Dynamic Visualization Tools”, In *Proceedings of the 10th Working Conference on Reverse Engineering*, IEEE Computer Society, pages 80-89, 2003

Paice 93 — Paice C., and Jones P., “The identification of Important Concepts in Highly Structured Technical Papers”, In *Proceedings of the 16th Annual International Conference on Research and Development in Information Retrieval*, ACM Press, pages 69-78, 1993

Pennington 87 — Pennington N., “Comprehension Strategies in Programming” In *Second Workshop on Empirical Studies of Programmers*, Ablex Publishing Corporation, pages 100-113, 1987

Pfleeger 98 — Pfleeger S. L., Software Engineering: Theory and Practice, Prentice Hall, 1998

Reiss 01 — Reiss S. P. and Renieris M., “Encoding program executions”, In *Proceedings of the 23rd International Conference on Software Engineering*, ACM Press, pages 221-230, 2001

Richner 02 — Richner T. and Ducasse S., “Using Dynamic Information for the Iterative Recovery of Collaborations and Roles”, In *Proceedings of the 18th International Conference on Software Maintenance*, IEEE Computer Society, pages 34-43, 2002

Rockel 98 — Rockel I. And Heimes F.: FUJABA - Homepage, http://www.unipaderborn. de/fachbereich/AG/schaefer/ag_dt/PG/Fujaba/fujaba.html, 1999.

Rugaber 95 — Rugaber S., “Program Comprehension”, *TR-95 Georgia Institute of Technology*, 1995

Scheuerl 95 — Scheuerl S., Connor R., Morrison R., Moss J., and Munro D., “The MaStA I/O trace format”, *Technical Report CS/95/4, School of Mathematical and Computational Sciences, University of St Andrews*, 1995

Sneed 96	Sneed H. M., "Object-oriented Cobol Re-cycling", In *Proceedings of the 3rd Working Conference on Reverse Engineering*, IEEE Computer Society, pages 169-178, 1996
St-Denis 00	St-Denis G., Schauer R., and Keller R. K., "Selecting a Model Interchange Format: The SPOOL Case Study". In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences,* IEEE Computer Society, 2000
Storey 97	Storey. M. A., Wong K., and Muller H. A., "How do Program Understanding Tools Affect how Programmers Understand Programs?", In *Proceedings of the 4th Working Conference on Reverse Engineering*, IEEE Computer Society, pages 183-207, 1997
Strzalkowski 99	Strzalkowski T., Stein G., Wang J., Wise B., "Robust Practical Text Summarization", In *Proceedings of the AAAI Intelligent Text Summarization Workshop*, pages 26-30, 1998
Sundaresan 00	Sundaresan V., Hendren L., Razafimahefa C., Vallée-Rai R. Lam P., Gagnon E., and Godin C., "Practical virtual method call resolution for Java", In *Proceedings of the 15th Conference on Object-oriented Programming, Systems, Languages, and Applications,* ACM Press, pages 264-280, 2000
Systä 00a	Systä T., "Understanding the Behaviour of Java Programs", In *Proceedings of the 7th Working Conference on Reverse Engineering*, IEEE Computer Society, pages 214-223, 2000
Systä 00b	Systä T, "Incremental Construction of Dynamic Models for Object-Oriented Software Systems", *Journal of Object-Oriented Programming*, Volume 13, Issue 5, pages 18-27, 2000
Systä 01	Systä T., Koskimies K., Müller H., "Shimba – An Environment for Reverse Engineering Java Software Systems", *Software – Practice and Experience,* Volume 31, Issue 4, pages 371-394, 2001
Systä 99	Systä T., "Dynamic reverse engineering of Java software", *In Proceedings of the ECOOP Workshop on Experiences in Object-Oriented Re-Engineering*, A short version in LNCS 1743, Springer-Verlag, 1999
Tai 79	Tai K. C., "The tree-to-tree correction problem", *Journal of the ACM,* Volume 26, Issue 3, pages 422-433, 1979
Tilley 96	Tilley S. R., Paul S. and Smith D. B., "Towards a Framework for

	Program Comprehension", In *Proceedings of the 4th Workshop on Program Comprehension*, IEEE Computer Society, pages 19- 29, 1996
Toad	http://alphaworks.ibm.com/tech/toad
Tzerpos 00	Tzerpos V. and Holt R. C., "ACDC: An Algorithm for Comprehension-Driven Clustering", In *Proceedings of the Working Conference on Reverse Engineering,* IEEE Computer Society, pages 258-267, 2000
Tzerpos 98	Tzerpos V. and Holt R. C., "Software Botryology, Automatic Clustering of Software Systems", In *Proceedings of the International Workshop on Large-Scale Software Composition*, IEEE Computer Society, pages 811-818, 1998
UML 2.0	UML 2.0 Specification: www.omg.org/uml
Valiente 00	Valiente G., "Simple and Efficient Tree Pattern Matching", *Research Report E-08034*, *Technical University of Catalonia,* 2000
Von Mayrhauser 94	Von Mayrhauser A. and Vans A. M., "Comprehension Processes During Large Scale Maintenance", In *Proceedings of the 16th International Conference on Software Engineering*, IEEE Computer Society, pages 39-48, 1994
Von Mayrhauser 95	Von Mayrhauser A. and Vans A. M., "Program Comprehension During Software Maintenance and Evolution", *IEEE Computer,* Volume 28, Number 8, 1995
Von Mayrhauser 96	Von Mayrhauser A. and Vans A. M., "On the Role of Program Understanding in Re-engineering Tasks" In *Proceedings of the IEEE Aerospace Applications Conference,* pages 253-267, 1996
Von Mayrhauser 97	Von Mayrhauser A. and Vans A. M., "Hypothesis-Driven Understanding Processes During Corrective Maintenance of Large Scale Software", In *Proceedings of the 12th International Conference on Software Maintenance*, IEEE Computer Society, pages 12-20, 1997
Von Mayrhauser 98	Von Mayrhauser A., Vans A. M., "Program Understanding Behaviour During Adaptation of Large Scale Software", In *Proceedings of the 6th International Workshop on Program Comprehension*, IEEE Computer Society, pages 164-173, 1998

Walker 98	Walker R. J., Murphy G. C., Freeman-Benson B., Swanson D., and Isaak J., "Visualizing Dynamic Software System Information through High-level Models", In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM Press, pages 271-283, 1998
Weka	Weka: http://www.cs.waikato.ac.nz/ml/weka/
Wiggert 97	Wiggerts T. A., "Using Clustering Algorithms in Legacy Systems Remodularization", In *Proceedings of the 4th Working Conference on Reverse Engineering*, IEEE Computer Society, pages 33-43, 1997
Wilde 92	Wilde N., and Huit R., "Maintenance support for object-oriented Programs", *IEEE Transactions on Software Engineering,* Volume 18, Number 12, pages1038–1044, 1992
Wilde 95	Wilde N. and Scully M., "Software Reconnaissance: Mapping Program Features to Code," *Journal of Software Maintenance: Research and Practice,* Volume 7, Number 1, 1995
Witten 99	Witten I. H., E. Frank, Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations, Morgan Kaufmann, 1999
Woods 99	Woods S., Carrière S. J., and Kazman R., "A semantic foundation for architectural reengineering and interchange", In *Proceedings of International Conference on Software Maintenance*, IEEE Computer Society, pages 391–398, 1999,
Zaidam 05	Zaidam A., Calders T., Demeyer S., and Paredaens J., "Applying Webmining Techniques to Execution Traces to Support the Program Comprehension Process", In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering,* IEEE Computer Society, pages 134-142, 2005
Zayour 02	Zayour I. Reverse Engineering: A Cognitive Approach, a Case Study and a Tool. Ph.D. dissertation, University of Ottawa, http://www.site.uottawa.ca/~tcl/gradtheses/, 2002