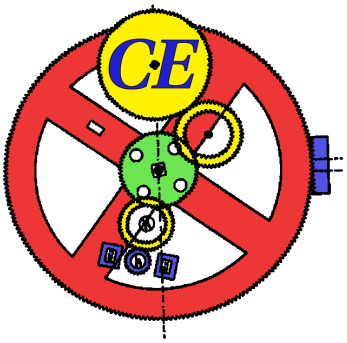

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2013

MSc THESIS

Modelling and Analysis of Execution Traces for Real-Time Applications

Preethi Ramamurthy

CE-MS-2013-07

Abstract

The need for increased computing capability and more diverse hardware with its evermore complex topologies continues to grow. The use of multicore processors, which is quite established in the desktop computers, is gaining importance in the embedded systems for industrial applications. Because of the complexity of the environment and the special requirements for these applications, the embedded multicore processors require optimised software architectures and the corresponding design methods. Performance optimisation is often based on the detailed knowledge of program behaviour. One such option to obtain information of program behaviour is software tracing, which forms the core of this thesis work.

Software tracing is being used for a while now to identify faults, anomalies and bottlenecks in sequential as well as parallel applications. Tracing is not only limited to collecting and recording the event data from applications, but also to utilise the trace data in a meaningful way for further analysis and visualization. It is also not trivial to classify or analyse the trace data, especially when the amount of trace data exceed a million of events per second. For this reason, there is a need to create a high level model of trace data.

In this thesis, we look at the theoretical model which describes the classification of trace events and also the relation between the events in a trace. The context for performing analysis on trace events is also discussed in detail. Based on the model, we have also created a prototype tool using Eclipse Modelling Framework (EMF) which performs the analysis on traces from real-time applications.

TU Delft

Faculty of EEMCS

Modelling and Analysis of Execution Traces for Real-Time Applications

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Preethi Ramamurthy
born in Coimbatore, India

Computer Engineering
Department of Electrical Engineering
Faculty of EEMCS
Delft University of Technology

Fraunhofer ESK
Munich, Germany

Modelling and Analysis of Execution Traces for Real-Time Applications

by Preethi Ramamurthy

Abstract

The need for increased computing capability and more diverse hardware with its evermore complex topologies continues to grow. The use of multicore processors, which is quite established in the desktop computers, is gaining importance in the embedded systems for industrial applications. Because of the complexity of the environment and the special requirements for these applications, the embedded multicore processors require optimised software architectures and the corresponding design methods. Performance optimisation is often based on the detailed knowledge of program behaviour. One such option to obtain information of program behaviour is software tracing, which forms the core of this thesis work.

Software tracing is being used for a while now to identify faults, anomalies and bottlenecks in sequential as well as parallel applications. Tracing is not only limited to collecting and recording the event data from applications, but also to utilise the trace data in a meaningful way for further analysis and visualisation. It is also not trivial to classify or analyse the trace data, especially when the amount of trace data exceed a million of events per second. For this reason, there is a need to create a high level model of trace data. In this thesis, we look at the theoretical model which describes the classification of trace events and also the relation between the events in a trace. The context for performing analysis on trace events is also discussed in detail. Based on the model, we have also created a prototype tool using Eclipse Modelling Framework (EMF) which performs the analysis on traces from real-time applications.

Laboratory : Computer Engineering
Codenumber : CE-MS-2013-07

Committee Members :

Advisor:	Dr.ir.Stephan Wong, CE, TU Delft
Chairperson:	Prof.dr.ir.Koen Bertels, CE, TU Delft
Member:	Ir.Salman Rafiq, Research Fellow, Fraunhofer ESK
Member:	Dr.Andy Zaidman, Assistant Professor, SERG, TU Delft

To my family and friends

Contents

List of Figures

List of Tables

Acknowledgements

I am grateful to my supervisor at Fraunhofer ESK, Mr.Salman Rafiq for providing me this opportunity to carry out the research work and for his constant support, motivation and valuable feedback at every phase of the thesis work. I would like to extend my gratitude to Mr.Ronald Strebelow at Fraunhofer ESK for his immeasurable support and guidance throughout the course of the work. I also thank Mr.Adriaan Schmidt for helping me with the technical issues whenever I had a doubt.

I would like to thank my supervisor at TU Delft, Dr.Stephan Wong for responding to all my queries patiently and guiding me during the span of this thesis work.

Last but not least, I would like to thank my family and friends for constantly supporting me through the difficult times and motivating me for the last two years.

Preethi Ramamurthy
Munich,Germany
July 15, 2013

1 Introduction

Embedded systems have always spanned a very broad spectrum of applications in the industry and is popular because of the low cost, size and technology used in it. There has been a dynamic change in the embedded systems as well as general purpose processors (GPP) from the single core processors to the multicore processors, due to the limitations in increasing the clock frequency in single core processors. Hardware designers are adding additional cores to chips to increase the throughput performance. These multicore systems share resources at multiple levels of the system and it becomes a difficulty when multiple applications are run on the same chip requiring them to interact and share the resources. On the other hand, even today many applications written are sequential in nature and porting them to multicore systems remains a challenge. When such applications are ported to multicore systems, maximum performance is not obtained because only a fraction of the available cores are used effectively. Diagnosis of performance bottlenecks on multicore based systems is quite difficult and laborious, as are multicore targeted optimisations.

Performance evaluation is key to the optimisation of computer applications on multicore systems. Performance optimisation most often is based on the detailed knowledge of program behaviour. The traditional methods used for obtaining program behaviour were not very efficient as argued in Section 1.1. Software tracing is one of the efficient and useful methods to collect information about the program behaviour and it forms the core of this thesis work.

In Section 1.2, a clear definition about the process of software tracing and the data obtained through software tracing, i.e., execution traces is provided. Section 1.3 highlights the research incentives behind conducting this research work. While Section 1.4 discusses the problem statement in detail, Section 1.5 describes the methodology used to solve the problem and the main contributions produced. Section 1.6 provides an outline of the entire thesis work by specifying the contents of each chapter in detail.

1.1 Traditional Methods to understand Program Behaviour

Traditionally, developers applied different methodologies to understand the program behaviour. The most common methods include reading the code, reading any associated documentation written by the software's author to understand the code or making use of a debugger to step through the code to understand its functionality. Sometimes the software is manually annotated with print statements to know the output at

specific points in the code during execution. All of the mentioned methods have some limitations, and now researchers have resorted to investigating by using automatically extracted execution traces as a viable data source [3].

Difficulties in understanding Program Behaviour

The traditional methods of understanding the program behaviour has some difficulties. Each developer might adopt a different coding style making it difficult to read or some parts of the original code may not be available due to legal reasons, or use of control flow jumps from one code block to another which cannot be predicted prior to actual execution of the software program [3].

In most cases, the associated documentation becomes outdated or does not exist for the particular software that the developer is interested in. In such cases, use of debuggers does not help much since the developer does not know where to insert the breakpoints to learn the program's behaviour.

Finally, manually inserting print statements requires a lot of effort from the developer to understand the program flow to be able to insert the code at points where useful information might be obtained [3].

The solution to the above problems is provided by software tracing, which employs instrumented code to collect information about the program behaviour.

1.2 Software Tracing

Software tracing is a technique used to understand what is going on in the system in order to debug or monitor it. Given the complexity of program structure of applications used in embedded systems in industries, use of tracing can help to debug a wide range of bugs that are otherwise extremely challenging.

Tracing can be compared to logging, as it basically consists of recording events that happen in a system at any given point of time. Unlike logging, tracing usually records information at a much lower level which occur frequently. Due to this reason, traces typically contain millions of events per second and the size of the trace files varies from megabytes to tens or hundreds of gigabytes [4].

Tracing is the process of consistent logging of a system and it can include events from the operating system kernel (system call entry/exit, network activity, scheduling activity etc) or it might consist of events from the user level (from applications).

Execution Traces

A broad definition of the term trace is given by Clements et. al.,[5] is as follows:

> "Traces are sequences of activities or interactions that describe the systems response to a specific stimulus when the system is in a specific state. These sequences document the trace of activities through a system described in terms of its structural elements and their interactions."

Execution traces or program traces are created in the runtime environment when the software is executing. Execution traces are a description of events that occur during the execution of a software application. An execution trace typically records events such as object creation, method calling, method returns, field access, field modifications, exception throwing and handling, class loading, and threading [3].

Execution traces are essentially static representation of the program which contains labels of dynamic profile information. It provides direct access to all the relevant profile information associated with every execution instance of every statement in the program. Any instance in the execution trace can correspond to statements at the source level, intermediate level or machine level.

In order to clearly represent every instance of the execution statements, it is necessary to distinguish between the execution instances of a program. For this purpose, the execution traces distinguishes the various instances by assigning unique *timestamps*. Timestamps are assigned by using a time counter, which is initialized according to the basic clock in the system. The notion of timestamps plays a vital role in representing and accessing the dynamic information in the execution traces [6].

1.3 Research Incentives

Multicore processors are becoming increasingly important in the application of embedded systems because of their high computing performance with low power consumption. Development of parallel software presents new challenges for many sectors since the existing software and tools are not designed for concurrency. The efficient development, optimization and testing of multicore software, especially for embedded systems with high reliability and time requirements, are open research problems.

This thesis work forms a part of the multicore system analysis done at Fraunhofer ESK, in particular the WEMUCS project. WEMUCS is a German acronym that stands for "methods and tools for the iterative development and optimization of software for embedded multicore systems". It is a project supported by the research and development programme "Information and Communication Technique Bavaria". The aim of the project is to provide efficient and iterative development, optimisation, and testing of multicore software with new tools and methods. For this purpose, several innovative technologies and tools for modelling, simulation, visualization, tracing and testing are

developed and integrated into a novel tool chain. These tools are evaluated using several case studies in the automotive industry, telecommunications and automation technology and developed iteratively.

1.4 Problem Statement

Software tracing has been used for a while now to identify faults, anomalies and bottlenecks in sequential as well as parallel applications. The recent trend of increase in the use of multicore in embedded industry has made it quite challenging to trace and optimise real-time software on such systems. In particular, the currently available software tracing/visualisation tools in the market focus mainly on the parallel programs for the High Performance Computing (HPC) domain and tools for programs in embedded domain are needed.

The uses of tracing are not just limited to collecting and recording event data from applications. The information from trace data can be utilised in a meaningful way to perform further analysis and visualisation. Meaningful data in a trace refers to any information in the trace streams that can help in understanding program behaviour or make debugging easier. This leads us to the first research question:

How to utilise the trace data in a meaningful way?

After identifying the useful information, it is also important to classify or analyse the trace data, especially when the amount of trace data exceeds to millions of events per second. This classification or analysis can then be used for understanding program behaviour in detail, debugging purposes and performance optimisation. The suggested classification has the need to satisfy the most common behaviour that can be accounted for in all the traces from embedded multicore applications. This leads us to the second research question:

How to classify and analyse the vast amount of information collected as execution traces?

Based on the classification, a general high level model can be drawn up, which supports the most common data format used in the trace files. The last research question being:

Will a generalised model help to analyse the trace data?

Considering the research questions, the main focus of the thesis is on the analysis of software traces as seen in Figure 1.1.

1.5 Methodology and Contributions

In order to answer the research questions through this thesis work, we use the following methodology:

1. To utilise trace data in a meaningful way, we need to first identify which information is considered meaningful/useful when looking at execution traces from real-time

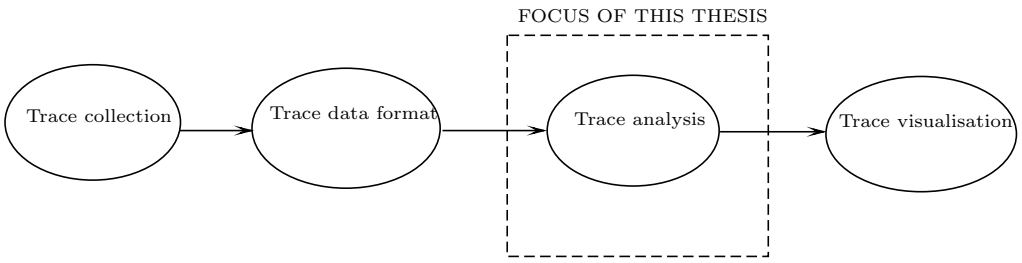

Figure 1.1: Thesis Outline

applications. Subsequently, we have to identify the useful information within the trace streams.

2. Describe a suitable and efficient model to classify and analyse the meaningful information found in the execution traces.

3. Choose a suitable platform and modelling environment to implement a prototype tool for analysing trace data according to the model described earlier. The prototype implementation can then be generalised.

The main contributions which have resulted as a consequence of this thesis research can be summarised as follows:

1. A model and hence a classification method which obtains the useful information from traces for analysis purposes is defined.

2. A generalised prototype tool which can be applied to traces from parallel applications is implemented and its use in debugging in illustrated.

The model developed in this thesis work is unique because of the fact that it provides a classification of event traces which makes use of the trace data in a meaningful manner. All the related work in this field was focussed on developing an annotation or assertion language to specify the expected or faulty behaviour and then check for it.

1.6 Thesis Outline

In this chapter, we have highlighted the research incentives and the problem statement. The rest of the thesis is structured as follows:

Chapter 2 gives the background information about the various steps involved in software tracing like extracting the information, storage format, analysing the trace and visualising the trace information. The related work in the field of modelling execution traces is also briefly discussed.

Chapter 3 discusses the development of a model which can be used to extract meaningful or useful information from the traces. It provides an overview of the general model

which describes events, its attributes and relations between events in a trace. A possible classification of trace events is also suggested, which then becomes the basis for implementing the prototype tool.

Chapter 4 elaborates how the model described in Chapter 3 is materialised using Eclipse Modelling Framework (EMF), so that it can be applied to a generalised set of data storage formats to test the program behaviour using the classification of trace streams as specified in Chapter 3.

In Chapter 5, we discuss the implementation of the EMF model with the most commonly used storage formats for traces in the embedded multicore field, i.e Open Trace Format (OTF), Common Trace Format (CTF) and Paraver trace file format. The trace format used in these storage types is provided in detail, followed by the detail of its implementation in the EMF model.

In Chapter 6, the results obtained when applying OTF trace files to the EMF model are discussed in detail for the various use cases and scenarios. These use cases and scenarios are based on the classification of trace streams as suggested in Chapter 3.

In Chapter 7, we end the thesis with conclusions and recommendations for future work.

2 Background

This chapter describes the various steps involved in the process of software tracing in detail which is essential to understand the thesis work. While, Section 2.2 describes how the trace data is obtained during the program execution, Section 2.3 provides details about the various storage formats used to store the trace data. Section 2.4 describes in detail, the context in which the analysis of trace data becomes essential and how the analysis can prove to be of help. Section 2.5 stresses the need for a visualisation of the trace data and gives a description of the most commonly used tools. Section 2.6 discusses the related work in the field of modelling or analysing event traces and also describes how the proposed work differs from the already existing work.

2.1 Trends in Multicore Tracing and Visualisation

To gather detailed insight into the activities and complex interactions of a multicore system managed by an operating system running multiple user applications simultaneously, a tracing structure and a tracing tool are required. Traditional tools and approaches fall short either because they are not scalable, not performing enough, or because they have been designed for a completely different purpose, but have been misused in lack of a better or adequate solution [7]. The various steps involved in the process of software tracing are as follows:

- Extracting the useful information or trace collection
- Storage format of traces
- Analysis of traces
- Visualising the extracted information

The use of dynamic analysis tools for program comprehension has been a popular research activity in the recent years. A recent survey [8] has shown that there have been a total of 172 articles on this topic between 1972 and 2008. Out of these 172 articles, more than 30 concern *execution trace analysis*, which has been shown to be beneficial for several activities like feature extraction, behavioural analysis, etc. [8]

Due to the large size of the execution traces, it is often difficult to understand a program through its execution traces. It was seen from an experiment performed by Reiss and Renieris[9] that one gigabyte of trace data was generated for every 2 seconds of executed C/C++ code or every ten seconds of Java code. Owing to this fact, many tools which perform automatic reduction of traces have been introduced. For E.g., [9, 10]

2.2 Extracting useful Information/ Trace Collection

There are basically two approaches to extract useful information from a program: instrumentation and sampling.

2.2.1 Instrumentation

With instrumentation, code is inserted in key places of the system (such as the top of particular method calls). This code then records the events at runtime for later offline analysis. The instrumentation can be done statically using trace-points or it can be done dynamically using software breakpoint-like constructs. The static instrumentation is widely used in Linux systems and is generally faster compared to the dynamic instrumentation. The static instrumentation provides low intrusiveness, low overhead and low performance impact to the target system's runtime behaviour. Dynamic instrumentation can be performed dynamically at run time and does not require intrinsic trace marker support in the operating systems kernel [11].

Other possible types of instrumenting the code includes manual instrumentation and auto instrumentation. Manual instrumentation involves inserting pieces of code manually at different points in the code to learn the program behaviour while in auto instrumentation the compiler inserts flags at random locations, which can then be used to study the program behaviour.

2.2.2 Sampling

With sampling the system remains unmodified and is instead analysed periodically by a profiler during runtime, allowing inspection of metrics such as the amount of CPU time used by processes and/or functions. The sampling utilities are capable of recording the contents of the program counter register whenever a certain number of performance counter events occur. The output from sampling can be of statistical nature. E.g., a list of execution time percentages for the most execution time consuming functions.

2.3 Trace Formats

In order to reduce the traces and express the useful information a common data format is essential. There are a variety of trace data formats being used for software trace data collection, analysis, and visualisation. In general, trace data format refers to the logical format that defines how the trace data records are designed and arranged. The logical format is of greater importance than the physical format as it determines the granularity and detail levels that can be represented and in succession the precision with which trace data can be correlated with the target system runtime behaviour. The most commonly used trace data format in multicore domain includes Open Trace Format (OTF), Common Trace Format (CTF), Paraver, Epilog, etc. [11]

All of the above mentioned formats share the basic principle of storing event records that are sorted with respect to time. Event types usually supported by all formats are: function call events, point to point message events, collective communication events, performance counter samples etc. So far the High Performance Computing (HPC) industry has been the key player in using these formats but the trend is also moving towards embedded industry to debug applications running on multicore systems.

The specification of a powerful trace file format has to fulfil a large number of requirements. The two main requirements include:

1. It must allow for an efficient collection of the event based performance data in a parallel program environment.
2. It has to provide a fast and comprehensive access to the large amount of performance data and the corresponding event definitions by the analysis software tools.

The following sections describe the most commonly used trace data formats, the Open Trace Format (OTF) and Common Trace Format (CTF) and Paraver trace file format. Only these three formats are used in this thesis work.

2.3.1 Open Trace Format (OTF)

The Open Trace Format (OTF) is an API specification and library implementation of a scalable trace file format, developed at TU Dresden and is a successor to the Vampir Trace Format (VTF3). OTF provides an open source means of efficiently reading and writing up to gigabytes of trace data from thousands of processes.

The Open Trace Format makes use of a portable ASCII encoding. It distributes single traces to multiple so called streams with one or more files each. The major advantage of using OTF is the fact that merging of records from multiple files is done transparently by the OTF library and the number of possible streams is not limited by the number of available file handles [12].

The read/write library acts as a portable interface for third party software and supports efficient parallel and distributed access to trace data and offers selective reading access regarding arbitrary time intervals, process selection and record types. Selective access becomes easier with optional auxiliary information.

2.3.2 Common Trace Format (CTF)

The common trace format specifies a trace format based on the needs of the embedded, telecom, high-performance and kernel communities. It is a collaborative effort of Multicore Association (MCA) and Linux community. It is designed to allow traces to be natively generated by Linux kernel, Linux user-space applications written in C/C++, and hardware components. One major element of CTF is the Trace Stream Description Language (TSDL) whose flexibility enables description of various binary trace stream layouts [13].

2.3.3 Paraver Trace File Format

PARAVER(PARAllel Visualisation and Events Representation) is a tool which is used to visualise and analyse a parallel events trace file. A trace file in Paraver can represent one or more applications running concurrently. The trace file format is based on the record stored. There are three types of records which can be stored using Paraver. They are state record, user event record and communication record [14].

2.4 Trace Analysis

The trace data collected according to a particular format can be utilised to the maximum extent when some analysis is performed on the information. The fundamental concept of software maintenance, reverse engineering, and data mining, which forms the context for performing analysis on events in a trace and visualising them is explained in the following sections, followed by the explanation of how software tracing helps in the application or usage of these concepts is given in Section 2.4.4.

2.4.1 Reverse Engineering

In forward engineering, higher-level concepts are transformed into lower-level artefacts such as source code [15]. Reverse Engineering refers to the opposite process of forward engineering, where the developer uses the lower-level artefacts to understand the higher-level concepts. An illustration of the concept behind forward and reverse engineering is shown in Figure 2.1. The specification for a specific functionality is converted into a design which is then materialised by coding in any specific programming language to provide the expected behaviour. The reverse engineering process then involves understanding the behaviour of the system and the code in order to bring out the exact specification. Tracing helps in this aspect by analysing the lower software artefacts, i.e., trace data, in order to obtain information about the actual expected behaviour of the program.

There are various tools and systems that support the reverse engineering tasks and these tools are specifically designed to help in the program comprehension process. The reverse engineering process involves the identification of software artefacts in a particular software system, and exploring how the artefacts interact with one another and their aggregation to form a comprehensive model of system representation that facilitates program understanding [16].

Dynamic versus Static Analysis Techniques

Reverse engineering techniques are classified into static, dynamic or a combination of both. Static analysis techniques involve examining the source code and building an abstraction of the runtime state, thereby allowing the programmers to account for all possible behaviours. As Diehl [17] states "*static analysis computes properties of a program which hold for all executions of the program.*" Dynamic analysis techniques, on

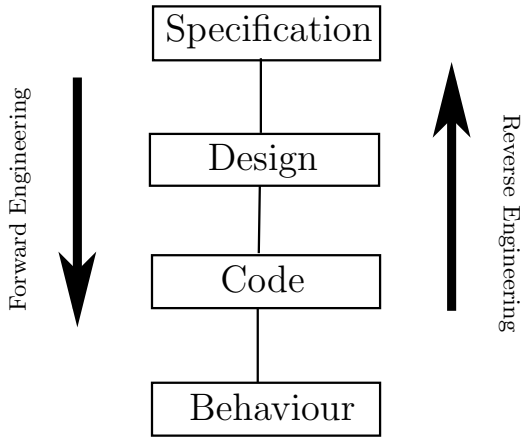

Figure 2.1: Concept behind Forward and Reverse Engineering

the other hand, executes the software system over certain input types and observes the execution over a period of time.

Program comprehension is not complete with just static analysis. Static program information captures the program structure, but information on runtime behaviour is not obtained from static analysis.

2.4.2 Software Maintenance

The implementation of software systems evolve over time and can become vast, consisting of thousands or even millions of lines of code. The software development is done over a period of time and mostly in uncontrolled environments. In the industry, when the product is not built to the proper quality standards, and without proper maintenance documentation, then maintenance effort becomes abnormally large and difficult. Another reason for the increasing complexity in maintaining the software systems is that their implementation is highly likely to contain design anomalies in the long run. The developers who develop the software systems in later stages, do not take the original developer's concept in mind, causing the system's implementation to degrade. Because of this consistent change, the implemented concepts tend to drift away from the initially documented ones and it becomes difficult to maintain the system or make changes to it [18].

Software maintenance is often perceived to be the fixing of bugs in the program, i.e., merely relative to errors and omissions. However, studies have shown that there is much more to software maintenance than just fixing bugs. Software maintenance as defined by IEEE standard [19] is as follows:

> "Modification of a software product to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment."

In general, maintenance is any change made to a software system after it is operational. These maintenance tasks may be required to eliminate bugs (corrective maintenance), tasks to adapt the system to changes in the environment (adaptive maintenance) and tasks to improve the performance or maintainability (perfective maintenance). Any software is constantly modified to adapt to the changing business or technology environment. The changing context requires better functionality in the software systems which is made possible by additions or modifications to the existing software. Continuous modifications becomes complex to maintain over a period of time [19].

Adaptive Maintenance accounts for most of the maintenance tasks performed in software industry these days. The process of adaptive maintenance requires the programmer to understand the program which is generally referred to as program comprehension. It has been estimated that the amount of time spent in program comprehension for maintenance exceeds 50 % of the overall time [20].

Information from software tracing can help in understanding the program behaviour better by providing details about the state of the system at the user level as well as kernel level.

Program Comprehension

In order to perform adaptive maintenance on the already available program, the programmer has to understand the structure and functionality of the program, i.e., program comprehension. Program comprehension is mainly concerned with the cognitive process that developers adopt when trying to understand the behaviour of the program. Developing an understanding of the program involves forming cognitive mental models to draw conclusions about the working of the program [20].

There are various models to describe the cognitive processes in developing a mental model of the information structures in the program. Although the principle behind these models differ in some ways, the key concepts are almost similar. The most widely used set of program comprehension strategies include top-down and bottom-up comprehension strategies and in some cases, a opportunistic mix of both these strategies are adopted.

(a) *Top-down Program Comprehension*: In this case of program comprehension strategy, the developer uses his domain knowledge to map the behaviour of system implementation. The mapping process is aided by the creation, confirmation and refinement of the hypothesis developed by the programmer, based on the software system's visible external functionality. This strategy is mostly used when the software system or the type of software system is familiar.

(b) *Bottom-up Program Comprehension*: By iteratively reading the code, the programmer abstracts a higher-level understanding of the software system. The process of constructing a unique item by bonding together several items of similar or related attributes while iterating over the lower level artefacts is termed 'chunking'. This strategy is considered to be useful because it allows the programmer to test his programming ability while developing chunks of larger size and suitable algorithmic

patterns.

(c) *Opportunistic Program Comprehension*: The opportunistic theory states that understanding is a mixture of 'top-down' and 'bottom-up' strategies. As Letovsky [20] clearly concluded, the program understanding involves forming a knowledge base to encode the domain, application and programming expertise. Based on the knowledge base developed, the programmer opportunistically follows either top-down or bottom-up program comprehension strategies. It has been observed that the developers frequently switch between these two comprehension strategies based on their experience from previously used strategies [20].

Tracing can help in an opportunistic method of program comprehension, i.e., either top-down or bottom-up. Program comprehension becomes more easy when more information is collected from the trace files, rather than top level information about the location and occurrence of the event.

2.4.3 Data mining

In simple terms, data mining refers to *extracting or mining knowledge from large amounts of data.* The reverse engineering process can be taken as a form of *data mining.* Witten and Frank [21] gave the following definition for data mining:

> "Data mining is defined as the process of discovering patterns in data. The process must be automatic or (more usually) semi-automatic. The patterns discovered must be meaningful in that they lead to some advantage, usually an economic advantage."

There are different kinds of patterns which can be extracted from data mining techniques [22]. They include:

(a) *Class descriptions*: Any data can be associated with classes or concepts. Class descriptions provide a method to describe classes of data in summarised, concise and precise manner. Such descriptions can be derived via *data characterisation*, a technique for summarising the general characteristics of the class or data, or by use of data discrimination, a method where the general characteristics of a class are compared with contrasting classes.

(b) *Clusters*: Data are grouped into clusters based on the logical relationship. The basic idea is that objects in a group will be similar and related, while one object within a group will have no relation with an object in another group.

(c) *Associations*: Data can be mined to identify the association between the objects. Association patterns are created by analysis of frequently occurring patterns in data, such as some sequences, substructures and item sets. Mining frequent patterns leads to the discovery of interesting associations between the various data sets, which can be evaluated in the form of association rules.

(d) *Outliers*: The group of data objects which do not comply with the general behaviour of objects in the data set. The outlier detection is of particular interest because it indicates any unexpected behaviour in the data set.

The different patterns that can be extracted from data mining provide the starting point for analysis in trace data. Association between different events in a trace, clustering events from a trace based on their logical relationship and identifying objects associated with specific classes or concepts can help in the analysis of trace files.

2.4.4 How can Trace Analysis help?

The thesis develops and applies the concepts and techniques of reverse engineering and program comprehension to create the model for trace analysis. As discussed in previous sections, software maintenance is of prime need when using multicore processors in embedded systems. Software maintenance mainly depends on program comprehension for the performing the maintenance tasks. Program comprehension is made easier by collecting execution traces from software, which indicate the program behaviour and allow for easy modification of code for maintenance or bug fix. Program comprehension is essential not only for software maintenance, but also for debugging purposes and performance optimisation. Trace analysis supports an opportunistic method of program comprehension allowing either top-down or bottom-up approach. Analysis of execution traces falls under the category of data mining as it involves 'extracting or mining useful information from large amounts of data'. Thus, the principles of software engineering, data mining and reverse engineering form the context to perform analysis on traces.

2.4.5 Filtering on Traces

Execution traces can be extremely large, and a developer may only be interested in one particular facet of an execution. For example, the developer may only be interested in how one component was used in a larger application, or they may only be interested in the object instantiations during execution.

Irrelevant information can be filtered out during the extraction phase or during the visualisation phase. Filtering out irrelevant information at the extraction phase means that the developer has previous knowledge about the behaviour of the program and thus cleverly instruments the program to obtain information. The second approach involves filtering for information depending on the data obtained as a result of instrumentation and it becomes easier to identify the useful data. The former approach is problematic since it requires the developer to make an informed filtering decision (clever instrumentation) with incomplete knowledge about the program behaviour. The latter approach is the approach that we utilise in this thesis as it becomes easier to identify the useful information after the actual program execution.

2.4.6 Maintenance Tasks facilitated by Trace Analysis

A majority of the maintenance tasks are related to system features, which can be supported by trace analysis. Eisenbarth, et. al., [23] define a feature as follows:

> "A feature is a system behaviour that can be triggered by the user of the system and produces some kind of output that is visible to the user."

A feature can be any functionality that the user wants to add to the system behaviour. It can be displaying text in a separate editor, opening a database file to read some value, etc. The fact that the start and end of the feature is known to the user through the trace collection helps in the maintenance tasks. Such tasks are discussed in the following sections:

Feature Location

Changes to the existing software systems are made on the basis of feature requests from the user. These feature requests are incorporated into the software by the developers who convert the feature to the corresponding source code. In order to make the necessary changes to the software, the developer has to first understand the source code components and also about how these components interact with each other. Trace analysis and visualisation helps the developers in this process, as it reduces the vast amount of source code and enables the developer to get a high level view on the executed code.

Fault Localization

Any failure in the software system is caused due to the fact that the software is not doing what it was expected to do. The reason for this unexpected behaviour can be any faults (bugs, defects) like deadlock, race condition within the code. Locating the source of these faults is a complex task, since the functionality of the software components are usually scattered inside the code and in most cases, the component exhibiting the fault might not be the actual source of fault in the system. In order to locate the source of fault, the developer has to identify the starting point of fault and proceed back in time to identify the source causing the fault. Trace analysis and visualisation is a key feature which can record the control flow. The developers can then easily locate the source of the fault location since the trace has events which indicate the state of the system before and after the fault.

2.5 Trace Visualisation

Rather than reading textual statistical information to understand the behaviour of the program, a visualisation of the information might prove to be more intuitive in the un-

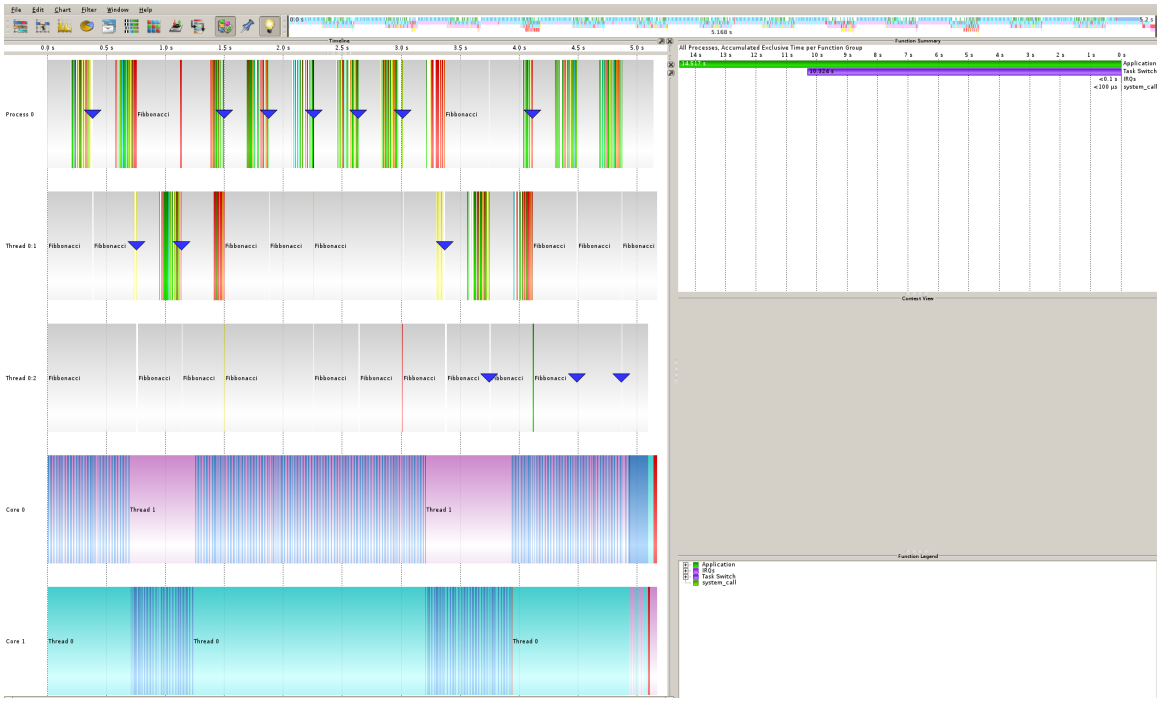

Figure 2.2: Snapshot of a trace file analysed using Vampir

derstanding process. For this reason, several tools exist which provide the functionality of visualising the collected trace data for analysis purposes. A general-purpose trace-visualisation system should incorporate the common aspects of trace processing and display. One such popular tool used for trace visualisation in High Performance Computing (HPC) domain is Vampir.

2.5.1 Vampir Tool Set

The Vampir tool set is a sophisticated performance analysis infrastructure for parallel programs that use combinations of MPI, OpenMP, PThreads, CUDA, and OpenCL. Vampir uses the Open Trace Format as the format file and relies on event trace recording, which allows the most detailed analysis of the parallel behaviour of target applications. During the runtime of an application, VampirTrace generates an OTF trace files, which can be analysed and visualised by the visualisation tool Vampir. The VampirTrace library allows MPI communication events of a parallel program to be recorded [24].

A snapshot of the Vampir tool set captured while analysing a OTF trace file can be seen in Figure 2.2. The five horizontal bands indicate the different processes in the system and each colour in those bands represent a event that happened in the system during the time of trace collection.

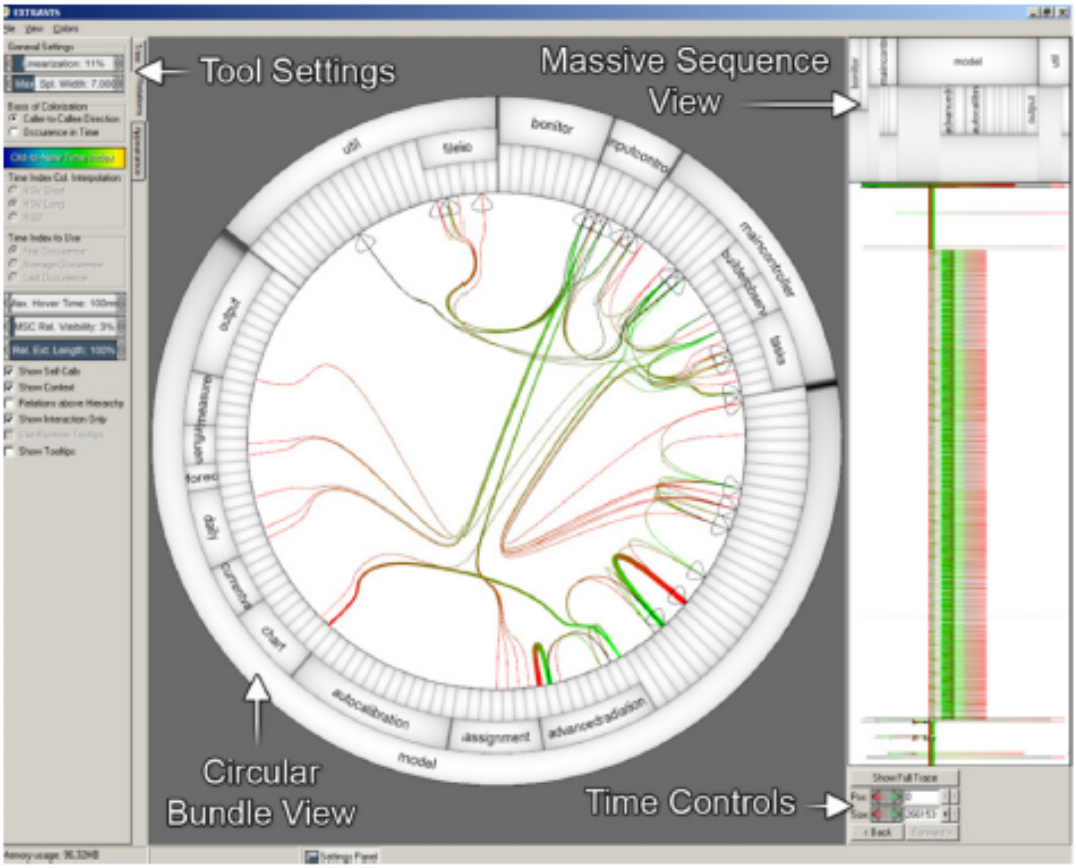

Figure 2.3: Screenshot of EXTRAVIS tool

2.5.2 Other Tools used in Trace Visualisation

The trace visualisation tools have been widely used for quite some time now to visualize the performance of software applications. While many tools provide a clear data abstraction and enhanced viewing of trace data, they do not often support analysis. The most recent work done on providing a visualisation as well as analysis environment of event tracing is the development of the EXTRAVIS tool [10], where they define a series of comprehensive tasks and measure the performance of EXTRAVIS in the Eclipse IDE environment. EXTRAVIS is a tool developed at TU Delft as part of the RECONSTRUCTOR project, which aims at developing state of the art tools in software architecture reconstruction.

The circular bundle view depicts the systems structural elements on the circumference of a circle, including their hierarchical structuring. Within the circle the edge bundles between structural elements in the circumference represent call relations. A trace is explored by selecting a time range in the massive sequence view and analysing on the basis of the circular bundle view which call relations are active within the time range [10]. A screen shot of the EXTRAVIS tool is seen in Figure 2.3.

Another remarkable visualisation tool for traces is the Visualisation Execution Traces (VET) developed at Victoria University of Wellington. VET is a prototype visualisation application which allows the user to manipulate multiple views of execution trace and filter according to their wishes. VET parses an execution trace and converts it to an event-driven interface for visualisation and filter plug ins. It makes the events of an execution trace available through an abstract API [3].

Numerous trace visualisation tools now exist, which focus on specific systems or programming languages. While *Shimbha, JaVis, Ovation, Jinsight* and *DJVis* focus on Java systems and object oriented programming in general, tools like *PV, AVID, Scene* and *Collaboration Browser* have been designed to view the execution traces from specific UNIX systems or kernels [18].

2.6 Related Work

Dynamic analysis of software programs becomes possible by developing precise models of program behaviour. A common practice in computer science methods is to assume that the program consists of finite number of states for the purpose of modelling. An event in a trace is considered as the elementary unit of action upon which the entire model is built. Since the events evolve in time, the program behaviour is nothing but the temporal relationship between events. This temporal relation can be modelled as the partial ordering of events in the semantics of parallel programs and some sequential programs too [25]. The formalisation of properties is essential to describe the intrinsic behaviour of events in a trace. These formal specifications can also aid in testing and debugging purposes.

Many formal specification languages have evolved to describe the behaviour of events in a trace like FORMAN (FORmal ANnotation Language), PARFORMAN (PARallel FORmal ANnotation language), Tiddle, etc. The general notion of events as represented in the description languages of FORMAN and PARFORMAN assume an event which has a beginning and an end. For atomic events, the beginning and end points of the time interval will be the same. These assertion languages or annotation languages specify certain assertions about the expected behaviour or faulty behaviour and then check for any abnormal behaviour based on the general knowledge of bugs and debugging strategies. The assertions can be about events, event sequences and sets and can include quantifiers. These languages are mainly used to locate concurrency errors in programs such as data races, atomicity violations etc. in distributed systems.

The concurrency and distribution aspects of the parallel program can be specified by a partially ordered trace of events executed by the distributed system [26]. The PARFORMAN is a specification language designed based on FORMAN, developed specifically for parallel programs and is used for expressing intended behaviour or known error conditions when debugging or testing parallel programs [27]. Tiddle (Trace Description Language) is a language for describing program execution traces and it can be used to generate small and simple test cases for identifying the concurrency bugs in parallel programs. The Test Behaviour Language (TBL) for traces provides a concise way of

describing and testing for trace properties [28].

Another approach to debug or analyse parallel programs in the event of any concurrency errors is to use model checking tools as described in [26]. A partially ordered trace of events executed by a distributed system is modelled by a collection of communicating automata. Model checking of the modelled automata then yields details about the properties of the system being tested.

In this thesis, we develop a model to classify the trace data in order to understand the program behaviour better. The model can be used to debug programs or analyse program behaviour based on the trace data obtained from the classified information.

The prototype tool box (named Corvus Tracing Tool Box) being developed in this thesis work provides the opportunity to analyse the traces not only based on the physical phenomena but also based on the logical relation between the events in the trace. Analysing traces according to the logical relation is an option that has not been explored before. This classification of logical trace provides an in depth understanding of the program behaviour.

2.7 Summary

This chapter provides the necessary background information about software tracing in multicore environment. It discusses the various steps involved in the process of software tracing, i.e., trace collection, trace data format, trace analysis and trace visualisation. The main focus of this thesis is on the trace analysis by applying a suitable model to classify the trace data. A brief description about the related work in formal methods of modelling events in a trace and its uniqueness is also given. As discussed in Section 2.6, the method of analysis that has been in practice so far depends on specifying assertions using certain assertion/annotation languages and then checking for expected or faulty behaviour. In this thesis, we aim at developing a model which describes the events in general and then classify the trace streams in order to facilitate better program understanding for debugging and optimisation purposes.

The subsequent chapters will provide a detailed description of the modelling of events in a trace and its attributes, which is later adapted for modelling in the Eclipse Modelling Framework (EMF), followed by testing with the OTF trace files.

Event Tracing and its Attributes

3

From Chapter 2, we discussed that there are various steps involved in the process of collecting trace data from an application. The data thus obtained are in huge volumes, and hence we need to analyse them efficiently to obtain useful information. In order to analyse the trace files, we define a model where we describe the events, its attributes, relation between events in a trace and a possible classification of the trace streams which aids in the process of analysis. In Section 3.1.1, the attributes that describe the unique events in a trace and discussed in detail, which is then followed by the description of the classification of the trace streams in Section 3.2. The types of logical relation between the events in a trace which forms a key role in the analysis of trace data is given in Section 3.3.

3.1 Events in a Trace

Event tracing is an important aspect to understand the performance of parallel applications. Event tracing is a technique for obtaining diagnostic information about running code without the use of a debugger. Each event in a trace is considered as an elementary unit of action in the programming model. The event is an abstraction for any detectable action performed during the program execution, such as a statement execution, expression evaluation, procedure call, sending and receiving a message, etc. In common words, an event represents any discrete activity that is of interest, especially with respect to performance. Any particular action in the program may be performed many number of times, but every execution of an action is denoted by a unique event [25].

The increased size and complexity of software programs today makes it nearly impossible to account for all the execution states in the system. This stresses the need for reliability and manageability of the programs by monitoring the states which are of importance. To obtain the useful states from the set of all possible states, instrumentation plays key role. Clever instrumentation of a program can reduce the amount of time spent in debugging the program.

In order to classify and analyse the events obtained as a result of careful instrumentation, we need to assign certain unique attributes to describe them. The attributes of an event are described in the next section.

3.1.1 Attributes of Events in a Trace

The event trace can be considered as a model of program's behaviour temporal aspects since the events are evolving in time. In order to specify the meaningful program behaviour properties, events need to be enriched with some attributes. Like discussed in Section 2.3, most of the trace formats store event records sorted with respect to time. Though the trace formats follow a general pattern to store the data, there is no unique model which can be used to describe the events in all the trace formats. In simple words, the storage format is common, but the data stored might be different. Based on the basic principle behind the storage of trace data formats, widely used in embedded systems now, we have chosen certain attributes to describe the events in a trace and thereby develop a distinct model which can be used for all trace formats. The timestamp and type form the essential attributes of events in a trace, while context and additional data are considered secondary. These attributes put together best describe the properties of an event and therefore form the basis for the classification of the trace streams which will be discussed in the next sections.

The attributes of an event, say A can be modelled as a tuple to express the timing behaviour of the event and characterise the properties associated with an event. The tuple illustrating the attributes of an event is given by $< e_t, e_c$,C,t,d $>$

where $e_t \in E_t$ is set of event types,
$e_c \in E_c$ is the set of event classes,
C describes the context of occurrence of the event.
t is the timestamp of the event, and
d is the data associated with the event.

1. **Event Type**: Classifying an event according to the event type is essential to describe the discrete activity which was part of system change during the program execution. Event type refers to any particular system change which gets recorded as an event and has a unique timestamp. The most common event types which are recorded by the trace data formats include function call events, point to point message events, collective communication events, performance counter samples etc., To state a particular example, consider an event which indicates the entry of the program control into the function and another event which indicates the exit of the program control from the function. These two events are considered to be of the type 'Function call events'.

2. **Event Class**: Event class groups events of a certain event type together. Classifying according to the event class gives information about the interaction of one events with other events. A set of events can be classified into mutually exclusive classes as send and receive, local and rendezvous.

   (a) *Send and Receive classes*: This class of events provides information about the sending and receiving of messages between same or different threads/processes and also between different network devices in the case of automotive. The inter-process and inter-thread communication that occurs through shared

memory and pipes, provides significant information about each send or receive in the communication between the threads or processes. The events under this class may signify any message or data passing between the threads, remote procedure calls, executable code etc. There may be several send or receive messages between the threads or processes, but each event has unique information like message length, data format, data etc. which distinguishes it from the other events.

(b) *Local class*: An event belonging to this class indicates some change in the state of the process or thread. Any event that is local to the thread or process is classified under the local class. Analysing events within the same thread or process gives details about the kernel level and user level operations, which can be used for general debugging purposes. It can be system call, function call, interrupt and trap handling, etc.

(c) *Rendezvous class*: This particular class deals with the events which happen between processes or threads. Although the sending and receiving of a message can happen between different threads or processes, they are classified separately because they are independent of the other tasks in the same thread/process. Synchronisation process between threads is used to control access to a state in which data is available only sequentially. Other operations like scheduling and process management which defines the use of resources by the various threads is also classified under this category.

3. **Context**: Contextual information plays an important role in characterising the event in a trace. Adding context to an event is essential to identify the origin or source of the event. Context of an event gives the minimal set of data needed to clearly distinguish one event from the other. E.g., CPU ID, Thread ID, Process ID and Core ID.

4. **Timestamp**: The notion of time and state plays an important role in the case of multicore systems where multiple threads run independently on different cores which are spatially separated working on a common clock. Processes communicate with each other by exchanging messages over a set of communication channels. Each process in the system generates a sequence of events. Time of occurrence of a particular event is important to understand the relation between different events and also for timing analysis.The concept of temporal ordering of events forms the basis for the analysis of bugs in the program being executed. Timestamp provides a unique identity to the events in a trace.

5. **Data**: In addition to the above mentioned attributes, any additional information about the event is given by the identifier or short message added to the event. This data is event specific and may be present in certain event types like function calls, send and receive messages, etc. Considering a function call event, it contains a function entry and function exit. The function entry event may have the following attributes for data: (1) Function identifier (2) Function argument (3) Details of the object to which the function belongs to. And in the function exit, it may have the following attributes: (1) Function identifier (2) Return type.

In case of send message event: (1) Destination address (2) Message type (3) Message Length.

These attributes of an event described above give details of the physical phenomena associated with them. By physical, we mean the physical location of occurrence, order of occurrence or nature of occurrence. However, these attributes do not give any detail about the logical relation between the events. For this purpose, we describe a classification of the trace streams (containing events) into physical and logical trace streams as seen in the following section.

3.2 Trace Stream

Trace stream can be defined as the collection of trace events from a well instrumented system ordered according to the time of occurrence of events. We propose a classification of the trace streams into physical and logical trace streams in order to make the process of understanding the program behaviour easier. Classifying into physical/logical provides all the necessary information about the trace streams as both the physical aspect and logical aspect are covered completely. Physical trace provides details about the physical attributes associated with the events, while the logical trace gives the logical relation between the events. This classification gives a complete meaning and no other classification is necessary to get a detailed knowledge from the trace data. This ordering provides better visualisation of the events and therefore, aids in the analysis of trace events.

3.2.1 Physical Trace Stream

The term 'physical' in 'physical trace stream' can be attributed to the fact that trace events are recorded in physical order i.e the order in which they enter/queue in the system. In general, the physical trace stream refers to a continuous log of trace events collected per thread or process or core in the system, as obtained during the execution of the instrumented program.

For any given application, there may be traces from many threads and corresponding processes, both at the system level and application level. In order to obtain information about the significant event or event of interest from a pool of events in a trace stream, some kind of filtering mechanism is essential. This filtering mechanism will then help to classify the raw trace stream (containing events) collected during program execution into events arranged according to any particular physical attribute mentioned in the previous section. The events can be filtered according to the various attributes like timestamp, context (process ID, thread ID or core ID), event type and event class. A trace stream is then created for the events filtered according to each of these attributes, i.e., there will be a stream for events filtered according to the process ID and another stream for events filtered according to timestamp etc.

From the results of the filtering, this class of trace stream can provide us with a mapping/correlation between the collection of events (both event type and event class)

to the collection of threads or collection of processes arranged according to time. E.g., it can help to identify that the read event of location A happened in thread A. This is an obvious information which helps to identify, which thread or process a particular event belongs to.

Figure 3.1 illustrates the mapping/correlation feature between events and threads/processes when ordered according to time. Figure (a) shows the events which are recorded during the execution and they are ordered according to time. This is the raw collection of events. Figure (b) shows the same events from Figure (a) which are now ordered according to the threads they occur in. The different bands of colour indicate the different threads in which the events happen and the events are ordered in time. Figure (c) indicates the events from Figure (a) again, but ordered according to processes on which they run. Here again, the different bands indicate the different processes in the system in which the events happen.

The suggested mapping is a useful characteristic for analysis to locate the bugs. For instance, data dependencies like RAW, WAR and WAW hazards between events can be analysed using the information about mapping.

Another useful characteristic is the ability to locate any abnormality or bug by tracing back to a point where the problem started (back tracking). Locating the abnormality is made simple, when the events are ordered according to the thread/process. E.g., when we know that the bug occurred at time t in thread A, we can easily identify the exact location of the bug by back tracking the events in thread A.

3.2.2 Logical Trace Stream

Analysing the logical sequence of events is the best way to understand and visualize how the entire program works. The logical trace stream is the collection of trace events in such a manner that the logical coherence between the events in a trace is illustrated. Here, logic refers to the valid reasoning which connects the events belonging to a certain system change during execution of the program. E.g., a function call job contains the events of transfer of control and arguments to the function (function entry), execution of statements of statements in the function and return (function exit). Logical coherence also indicates a certain relationship between the events in a trace.

The aspect of time plays an important role in understanding the logical sequence or coherency between the events in a trace. In the case of network packet processing, the sequence of events include queuing of the packet, scheduling of the packet, processing the packet and exit of the packet from the system. In the above case, each event connects with the next event by a factor determined by the time spent in the particular event. Consider an example of a function call event, all the events that happen after the function entry and before the function exit event are logically connected because of the fact that they happened in time after the actual function call, which triggered the function entry.

Execution of an event may involve various threads or processes. Each logical trace may connect different physical trace streams. In order to analyse the logical ordering of events in the system, we have to collect information across the different threads and

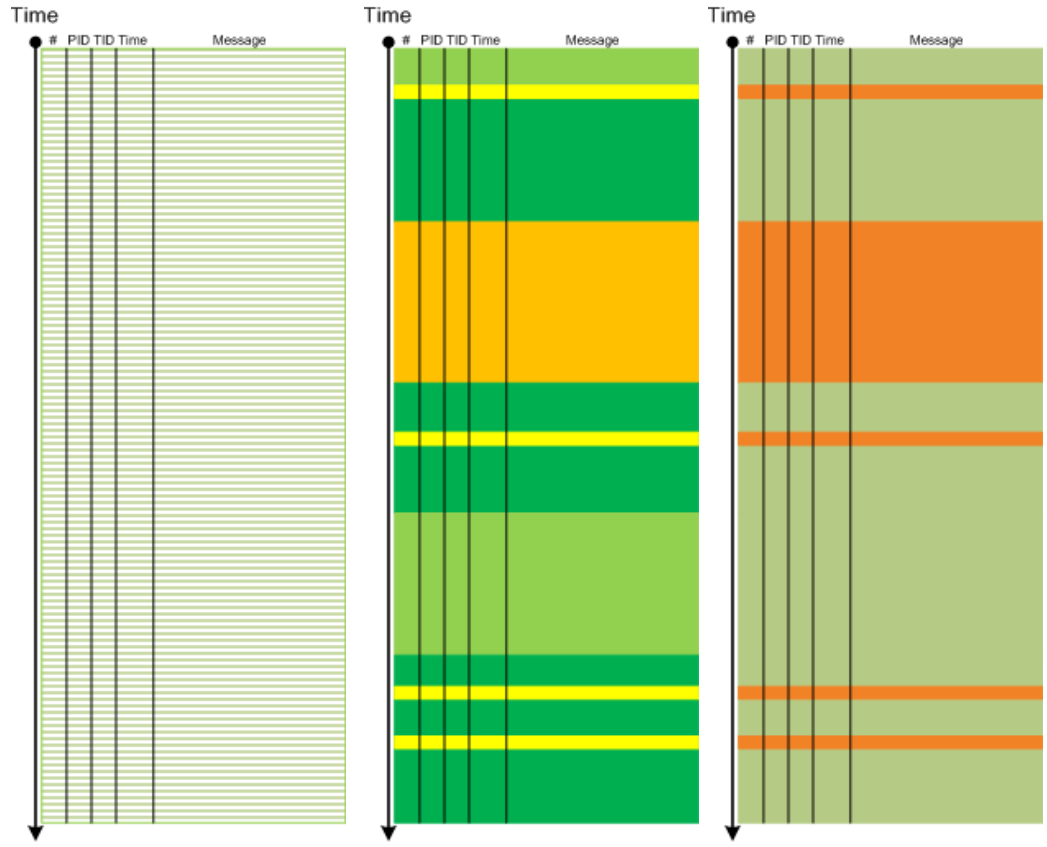

Figure 3.1: Traces separated according to events, threads and processes respectively

processes where the event happens, which again requires filtering to get the required results. In other words, physical trace contains the raw trace data collected from the program execution arranged or filtered according to any particular physical attribute, while the logical trace collects the raw trace data according to some logical relation between them. E.g., a function A call in thread A may call another function B in thread B, in this case both function A and function B are in different threads, but they are connected by a logical relation. The physical trace contains these two events filtered according to their thread of occurrence, i.e., in two different trace streams, while the logical trace contains these events in a single trace stream ordered according to their logical relation in time.

Information from the logical reasoning can also be used to identify abnormal behaviour. Considering that we are looking for a logical relation between the entry and exit of a function, we can identify when the function entered the system and when it exited. In case of any abnormal behaviour, we can identify that the function entered but never exited the system or never entered the system and this data helps in localising the bugs in the program. Similarly, faulty behaviour in the case of communication where the message was sent but never received can be located when analysing logical trace. Another

possible scenario of usage can be in the case of scheduling to study the functionality of the scheduling algorithm implemented.

3.3 Relationship between Events in a Trace

The temporal relationship that exists between events in a trace suggests the need to define a relationship between them. A relationship between two events in a trace is the meaning given to the attribute which connects the two events. The relationship also gives the nature of correspondence between the events. Relationship between events can be defined as the couple <e,r> where e denotes the events participating in the relationship and r is the type of relation between the events under consideration. In the next section, we describe the type of relationships that can exist between events in a trace.

3.3.1 Types of Relationship

There can be numerous types of relationships that exist between events in a distributed system. But the dynamic nature of multicore systems limits the kind of relationships to four possible types [25]. The most dominant among them being the 'happens before/precedence relation' and 'inclusion relation'. The other possible types include 'Causal relation' and 'concurrent/parallel relation'.

1. **Causal Relation**

   The causal relation can be defined as 'the ability of an event to directly or transitively affect another event'. Analysing the causal relationship between the events is also useful for the detection of race conditions and other synchronisation errors. Causal relation determines the sequence in which the events must be processed so that the cause and effect appear in the correct order. Causality also plays an important role in the exploitation to achieve maximum parallelism. The trace data can significantly be reduced by appropriately representing the causal structure of the program [29].

   It can be of three types in specific:

   - ***Direct Causal Relation*** - In most cases, causality between events is defined as the direct consequence of one set of events to another set of events. Given a scenario, when one is able to derive the fact that 'A has caused B' directly, then it is counted as a direct causal relation. Direct causal relation holds exclusively for events which happen in a single transition and it is the most obvious interpretation for the given case. Consider a function call in a program which calls another function or subroutine, there is a direct relationship between both the functions as seen in Figure 3.2.
   - ***Indirect Causal Relation/Chain Relation*** - The causal relation cannot always be directly implied and may involve some additional indirect references on the path. This is caused due to the presence of one or more intermediate

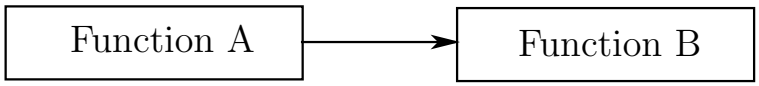

Figure 3.2: Example for direct causal relation

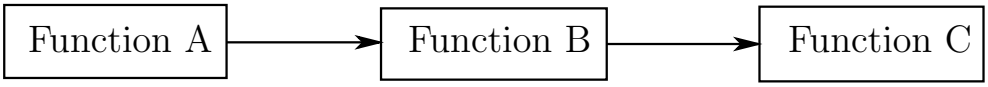

Figure 3.3: Example for indirect causal relation

events that are common to both event A and event B. For instance, consider an event X that exerts a causal impact on event Z, but only through its impact on a third event, Y i.e X affects Y, which in turn affects Z. There exists a chain reaction from event X to event Y to event Z and event Y is common to both events X and Z. Figure 3.3 shows an example where function A makes a call to a function B, which in-turn calls another function C, creating a chain of events.

- ***Bi-directional Relation*** - When creating a model of the relationships between the events, it is essential to capture the links or associations between all the events in the system, for that purpose, we take into consideration the bidirectional relation. Bidirectional relationship is valid in both directions. An event X has a causal influence on event Y, which in turn, has a causal impact on event X. Thread synchronisation operations are the best examples for bidirectional relation. Figure 3.4 shows a synchronization operation of 2 threads where each thread affects the other to produce the final result. Thread A waits for thread B to complete execution of certain function and then synchronises.

2. **Happens before Relation**

   In a distributed system, it is often impossible to say that one of two events occurred first. The relation 'happens before' is therefore always a partial ordering of events in the system. The happens-before relation between events in a trace is the smallest transitively closed relation on operations and exists for each and every event in the trace stream. This relation holds between any pair of events in the trace and it is not always necessary for a dependency to be present.

   An example situation in which the happened-before relation is formally defined as the least strict partial order on events is as follows:

   - If events a and b occur on the same process, a $\rightarrow$ b if the occurrence of event a preceded the occurrence of event b.
   - If event a is the sending of a message and event b is the reception of the message sent in event a, a $\rightarrow$ b.

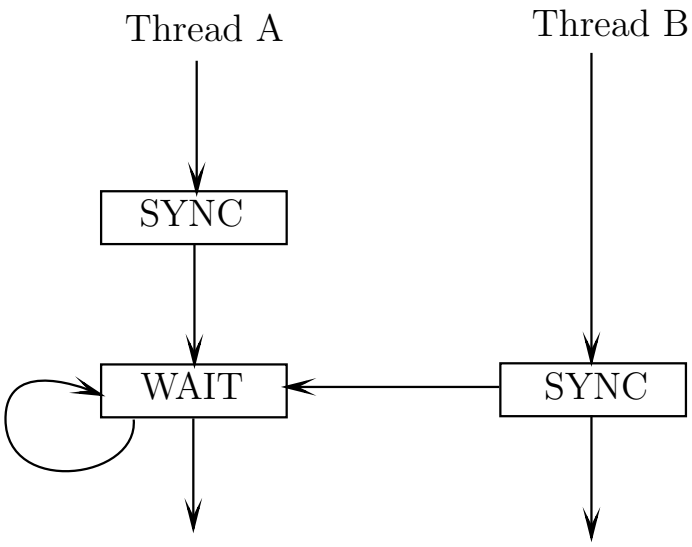

Figure 3.4: Example for bidirectional relation

Figure 3.5: Example for happens-before relation

3. **Concurrent/Parallel Relation**

   The various steps of execution are interleaved between numerous threads and processes in parallel processing systems. Concurrency is synonymous with such systems and is an obvious relation which exists between events in a multicore environment as well. When two events can occur in parallel and are not dependent on one another, it can be said that the two events have a concurrency relationship. All events which are not causally related can be executed in parallel.

   Since all events in a multi-core environment happen concurrently, this category is only mentioned here to give a complete meaning to the classification, but it does not have any effect in practice.

4. **Inclusion Relation**

   Actions performed during the program execution may be at different levels of granularity, some of them include other actions. E.g., the process of calling a subroutine which contains various execution statements as seen in Figure 3.6.

   Inclusion relation can be considered as a meta-class of relation since it entails the previously mentioned causal relation and happens before relation as well. Under this relationship events can be hierarchical objects and it becomes possible to consider program behaviour at appropriate levels of granularity for complete understanding of the program. Inclusion relation can be considered as a meta type of

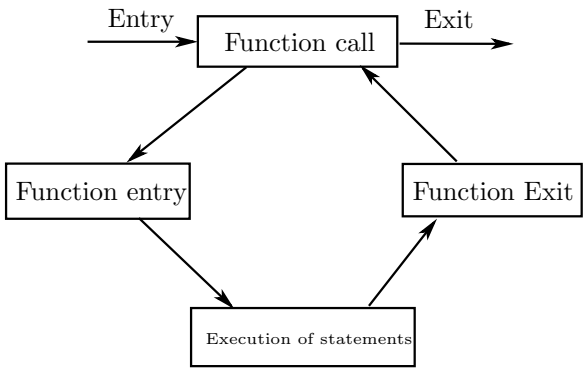

Figure 3.6: Example for inclusion relation

relations, since it can include other types of relation [25] .

Considering an event A and an event B in a single trace file, there may exist more than one relationship between them. If the event A is a 'send event' and event B is 'receive event' and the event A happens in time before event B for obvious reasons. Then there exists a relationship of 'happens before' and 'causal', since only a send event can cause a receive event. In such cases, the relationship which gives the most information about the two events is taken into account. In the above case, the 'causal' relation gives more information about the events than 'happens before' because happens before is an obvious relation that can exist between any pair of events, even those that are million events apart.

3.4 Summary

This chapter provides a complete understanding of the discrete elements of activity in a trace stream, i.e., the events. It describes the various attributes of an event in detail and analyses the importance of each, which then forms the basis of the model being developed. The classification of trace streams into physical and logical trace streams is given a clear definition. The logical relationship between the events in a trace which gives the complete meaning to the flow of program execution is presented and the 4 major possible relations, causal, happens-before and inclusion are explained with examples. The next chapter discusses how the model developed in this chapter is materialised using Eclipse Modelling Framework (EMF).

4 Modelling of Trace Analysis

In Chapter 3, we described the various attributes of an event and the relationship between them. A possible classification into physical and logical trace is suggested, which is aimed at making the process of understanding the program behaviour easier for debugging purposes and performance optimisation. This chapter discusses the need for building a generalised model based on the classification described in Chapter 3 to analyse the event traces. The prototype model for analysis is built using the Eclipse Modelling Framework (EMF) which has unique features as described in Section 4.3. A general description of the EMF model and its various components is discussed in the Section 4.3.2, followed by how the features in EMF are utilised in building a model for trace analysis in Section 4.4.

4.1 Need for a General Model in Trace Analysis

In order to analyse the huge trace files collected from the execution of an application, a generalised model needs to be drawn. This generalised model can then become applicable to any trace file format used in the embedded multicore domain. For this purpose, we develop a unique model for analysing the trace files and choose Eclipse as the platform for implementing this prototype model. Eclipse being a versatile and open source application can support dynamic analysis of various trace files under various conditions. An overview of the prototype tool developed using Eclipse Modelling Framework (EMF) is discussed in the next section.

4.2 Overview of the Corvus Tracing Tool Box

The main aim of the model (in turn a tracing tool box named Corvus Tracing Tool Box) being developed in EMF is to create a physical and logical trace from the trace streams obtained during the execution of the program. A simple overview of the model is provided in Figure 4.1. The trace files collected during the runtime of any application consists of numerous trace streams, which may be indexed for easy access. Each trace stream in turn contains many events which provide details about the actual system change. These events are passed through a filtering mechanism based on the event attributes and relation as described in Chapter 3. Filtering according to the event attributes yields the physical trace and logical trace streams can be obtained by specifying particular use cases of relation. These filtered traces can be used for further analysis of execution traces like statistical or behavioural analysis.

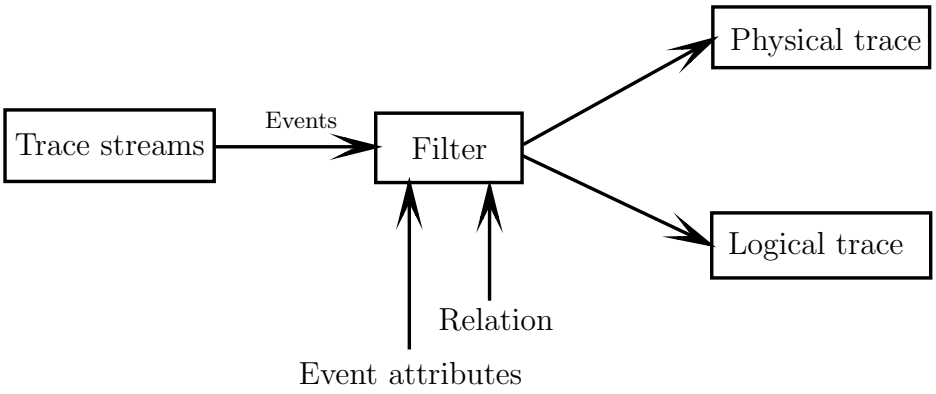

Figure 4.1: Overview of Corvus Tracing Tool Box

4.3 Eclipse Platform

Eclipse is an open source software project, the purpose of which is to provide a highly integrated tool platform. It provides a core of services, which enable controlling a set of tools working together in order to support programming tasks. Eclipse is widely used to build IDE's, because of the basic mechanism of extensibility supported by Eclipse. One such approach is the use of plug-ins, the architecture and development of which, will be discussed in this chapter.

4.3.1 Plug-in based Architecture

A plug-in plays a key role in the integration of tools with Eclipse. Except for the small runtime kernel, everything works as a plug-in in Eclipse. Any new plug-in that integrates to the already existing plug-ins, works very much similar to the other plug-ins already present. Some plug-ins have a higher priority than others. For instance, the Workspace and Workbench are two important plug-ins in the Eclipse Platform, which provide the extension points for most of the plug-ins in Eclipse. Every plug-in requires an extension point in order to integrate and function in the Eclipse IDE. The Workbench and the Workspace are not the only Eclipse components that can be extended by other plug-ins. There are many different components like the Debug component, Team component and Help component which add various functionalities to the Eclipse environment [30].

A plug-in in Eclipse can be considered as a component which provides a certain type of service within the Eclipse Workbench and is the basic unit of function. A component here refers to a system which can be configured at run time. The Eclipse runtime provides an infrastructure which supports the activation and operation of a set of plug-ins working together for development activities. Every running instance of Eclipse, supports a plug-in which is embodied in an instance of a plug-in runtime class. The plug-in class provides configuration and management support for the plug-in instance [31].

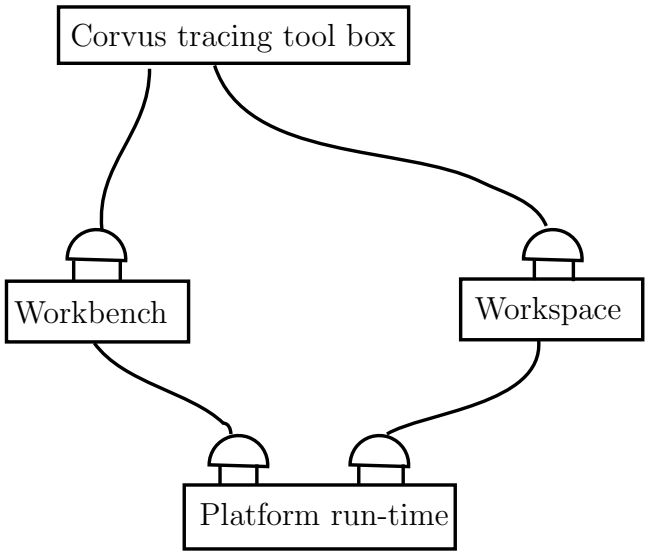

Figure 4.2: Plug-in architecture

4.3.2 Eclipse Modelling Framework (EMF)

Eclipse Modelling Framework (EMF) is a modelling framework which exploits the features provided by Eclipse. A data model or domain model represents the data one wants to work with. EMF can be used to model the data models. EMF makes a clear distinction between the actual model and the meta model. The meta model gives the structure of the model and the actual model is then an instance of the meta-model. The meta-model can be built using different data model artefacts such as XML Schema, the Rational Rose model, the Ecore model or Java annotations. EMF generator creates model code for all the above mentioned artefacts, which includes type-safe interfaces and implementation classes for the data model [30].

The Meta model in EMF consists of the Ecore model and the Genmodel which are described in the following sections:

Ecore Model

The model used to represent models in EMF is called Ecore. The Ecore model contains information about the classes which are defined. There are four basic Ecore classes which are used to represent the data model [32].

1. **EClass** is used to represent a modelled class. It has a name, zero or more attributes and zero or more references.
2. **EAttribute** represents a modelled attribute. Attributes have a name and a type.
3. **EReference** represents any one end of an association between classes. It has a

name and a boolean flag to indicate containment and resource target type(another class).

4. **EDataType** is used to represent the type of an attribute. A data type can be primitive like *int* or *float* or an object type like *java.util.Date*

Genmodel

The generator model or the genmodel contains access to all the additional information needed for code generation. The genmodel is a serialised generator model with cross document reference to the .ecore file. Since the Ecore metamodel is completely independent of the generator model, the Ecore model remains pure and independent of any information related to code generation. Any change in the Ecore model automatically changes the generated code through the generator model, which makes using EMF advantageous.

4.4 Model of Tracing Analysis

In order to create an application which can perform analysis on the event traces, we created a generalised model using Unified Modelling Language (UML) in EMF as shown in Figure 4.3. By generalised model, we mean a model which can be used with the different trace file formats used in the embedded multicore domain. As described in Section 3.1.1, the different trace formats follow the general format of 'events in a trace file' to store the information, however there is no general model which can account for all the trace data formats in common. In simple words, the storage format is common, but the data stored in each format is different. In order to prove the fact that the model being developed in this thesis is applicable to different trace formats we demonstrate the functionality of the model with trace files which follow the Open Trace Format (OTF), Common Trace Format (CTF) and Paraver trace file format. The model consists of several basic blocks, which can be applied to trace files from any application. These trace formats mostly use a call-back mechanism implemented in C/C++ language to store the information. In order to use the EMF model and hence, the Eclipse platform, the trace files of the above format have to be read into the EMF model using a suitable JNI/XML interface which is described in Section 5.3.1.

The model as seen in Figure 4.3 is designed specifically for the trace files which follow the Open Trace Format (OTF). It can be applied to other trace formats like CTF and Paraver, with just the modification in the data types. Each block in the model represents a class and there are various attributes for each class. And operations can be performed on the attributes in the class. The various blocks in the UML diagram for trace analysis are described in the following sections.

1. ***Trace Block***: Like described in the Chapter 3, the execution of the software system is collected in the form of trace files, which consists of many trace streams. Each trace stream consists of a number of events which happen during the execution

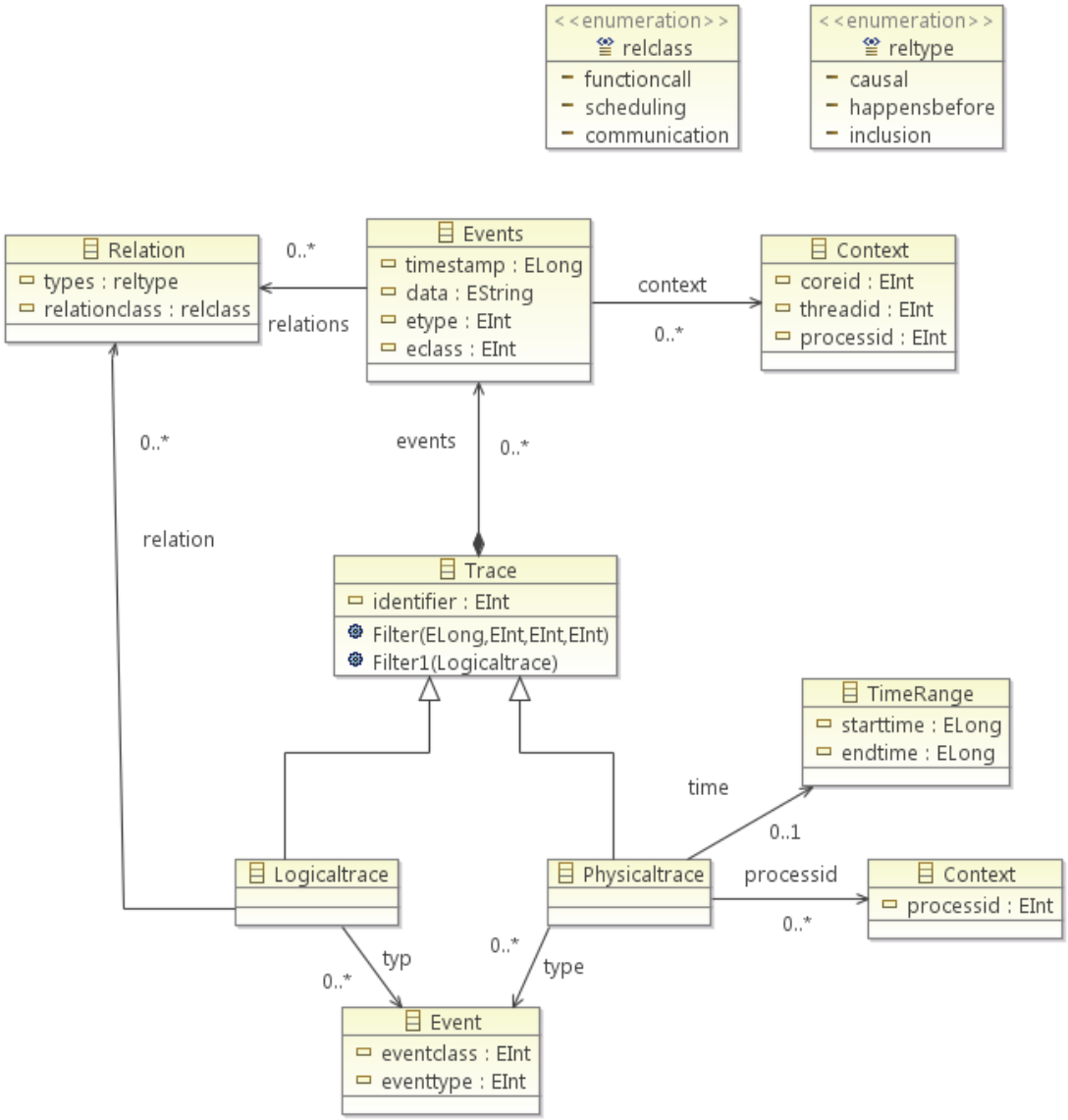

Figure 4.3: Model of Trace Analysis

of the program. The containment relation between the trace block and the events block indicates this fact. In most cases, the trace streams in a trace file are identified by a stream number, which uniquely indicates the location of the trace stream in the file. The attribute 'Identifier' in the trace block signifies the above fact. And the filter operation on the trace files are performed in the Trace Block. The Physical and Logical trace blocks are obtained as a result of the filtering operation. Both the Physical and Logical trace blocks inherit data from the Trace block.

2. ***Events Block***: Each event in a trace stream is considered as the atomic unit of action. Events are identified by the various attributes which was described in

Section 3.1.1. There exists a unique relationship between events which is indicated by the association relation between the Events block and the Relation block. In the UML model, the various attributes are defined as follows:

(a) **Timestamp**: The timestamp is a unique identifier for each event in the trace.

(b) **Context**: The most common context information available from trace files include the CPUID, Thread ID and Process ID. Therefore they are included as attributes in the class of 'Events'.

(c) **Event Class**: The event class is the collection of event types.

(d) **Event Type**: The event type refers to the discrete action which takes place during the execution.

(e) **Data** Any additional information about the event is included in the attribute 'data'.

3. ***Physical Trace Block***: The Physical trace block represents the trace streams which are filtered according to the physical phenomena, i.e., event attributes in most cases. Eg. Timestamp, Context, Event type and Event class. The event attributes provide information about the physical phenomena associated with an event. Filtering according to the different event attributes happens as follows:

   (a) Timestamp: All events between a given time range can be filtered. The time range must have a start time and an end time.

   (b) Context: Events can be filtered according to their context, i.e., CPU ID, Thread ID and Process ID. This gives information about the events which were running on any particular core/process/thread at any given time.

   (c) Event type/class: Events can be filtered according to their event type/event class. In this case, information about when and where, the events of any type or class happened will be known.

4. ***Logical Trace Block***: Logical trace block contains trace stream which are filtered according to the relation between events. The Logical block has an association with the events filtered according to the event type/class and the relation block which specifies the type of relation according to which the events are filtered. Filtering in this context is performed according to some specific use cases which are specified based on the trace file format being used and the information that is considered meaningful for analysis.

5. ***Relation***: The relation block has two attributes, the relation type and the relation class. The filtering in this case is done based on the knowledge of how the events are related for specific use cases. In this case, the specific use cases for the OTF file format include the function call, scheduling and communication which are modelled as enumeration type in the UML model. The relation types that we are looking for include causal relation, happens before relation and inclusion relation.

   The logical relations modelled as the 'relation type' which might be of interest when analysing a trace file include the causal relation and inclusion relation. The

happens before relation is the most obvious previous event and defining a concrete relation between any two random events which happen before one another is not always possible.

4.5 Summary

This chapter materialises a general model to analyse the traces based on the description of events and the classification provided in Chapter 3. The Eclipse Modelling Framework (EMF) is chosen to develop this model and a brief description of the features provided by the EMF are discussed. A model which will be used to analyse the traces from the different trace file formats which was developed using UML and the various blocks in the UML model which represent the various aspects of the classification and hence the analysis are described in detail. The next chapter shows a model implementation of the UML model developed in this chapter with the input from the trace files of OTF format.

Implementation of Corvus Tracing Tool Box

5

In the last chapter, we described a model for analysing the trace files using EMF. This chapter describes the use of Eclipse Modelling Framework (EMF) to analyse the execution traces from real-time applications in embedded multicore domain which follow the trace file formats of OTF, CTF and Paraver. The structure of OTF trace format and the application of OTF to the EMF model is described in Section 5.2. In Section 5.3 the method used for analysing OTF files in Eclipse is illustrated. The application of other trace format files such as CTF and Paraver is discussed in Section 5.4.

5.1 Structure of Open Trace Format (OTF) Files

As discussed in Section 2.3.1, Open Trace Format (OTF) is a scalable trace file format, which makes use of ASCII encoding. OTF uses multiple streams per trace and the ASCII encoding allows for binary search on files and reduced storage sizes for small values. The use of ASCII encoding makes OTF a very powerful format with respect to storage size, human readability and search capabilities on timed event records.

5.1.1 Streams and Files

OTF supports a *stream model*, which helps in fast and selective access to large amounts of performance trace data. Stream model means that single separate units represent segments of the overall data. The OTF streams contain multiple independent processes and each process belongs to a single stream exclusively.

Each stream consists of multiple files which store definition records, statistical records, event records and some status information. A single global master file holds information about the process to stream mappings. Figure 5.1 illustrates the above mentioned information. In a OTF file, the records are always arranged as single lines of text and the detailed structure of every record type is defined separately [1].

The OTF library supports transparent data compression by making use of *ZLib* in addition to the ASCII encoding, which is not most economical by itself. OTF uses a small compression level which allows for fast compression and still produces a good data reduction to about 20 - 35% of the original file size on average [1].

The following sections are referenced from the OTF Specifications article written by Andreas Knüpfer et. al. from TU Dresden [1].

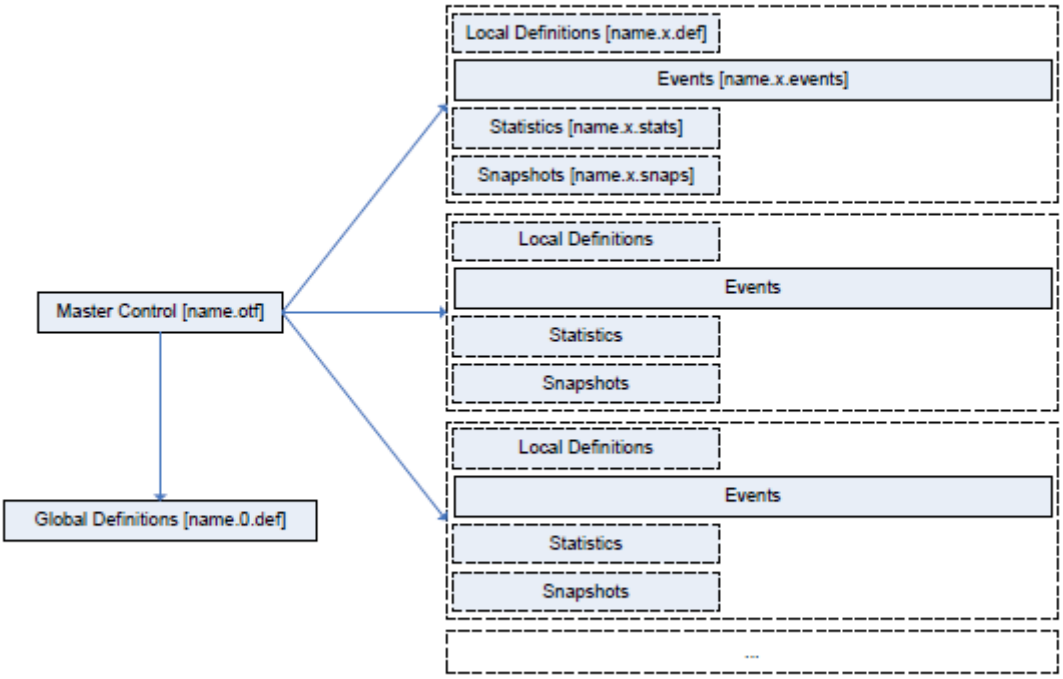

Figure 5.1: File System in OTF trace [1]

Definition Record files

The definition records consists of tokens which are referenced by event records. Use of definition records allows for more efficient encoding. The definition records can be present in a single stream or globally. Any trace file will always contain definition records. All the definition records contain a stream identifier for identifying the scope of the record.

Event Record Files

Event records are the actual payloads for traces. The event records are always sorted in temporal order and a stream can contain only one event record. Events which can be defined using OTF include entering or leaving a function, sending/receiving of inter process messages and measurement of hardware counters.

Statistical Record Files

Summary records or statistical records provide an overview over a whole interval of time. The data provided by statistical records are not explicit values for a particular time interval but are obtained in a differential fashion.

Snapshot Record Files

Snapshot records provide the possibility to access arbitrary data within the event files

by using the call stack mechanism. Snapshots are not generated by the OTF library itself but can be explicitly added. It is possible to add/manipulate/replace/delete snapshot records of a stream without affecting the event data.

5.2 Application of OTF Files to EMF Model

In the case of OTF files, the timestamp assumes the data type 'Long' as seen in the UML model in Chapter 4. The OTF files, mostly have only the Process ID information. The event types supported by OTF library include entering/leaving of functions, sending/receiving of inter process messages or measurement of hardware counters [1]. Table 5.1 shows how the information from OTF files can be classified into Event type and Event class to be used with the EMF model.

The function entry/exit events can be collected from the event records in the OTF files using handlers OTF_Handler_Enter and OTF_Handler_Leave. The Send and Receive events have different handlers in the event record which are OTF_Handler_SendMsg and RecvMsg. In the case of measuring or using counter information, the handler OTF_Handler_Counter is used. More information about how the handlers in OTF work is described in Section 5.3.

EMF Model	**Open Trace File format**		
	Function entry/exit	**Send/Receive events**	**Counter measurement**
Event class	Function group	Message group	Process number
Event type	Function entry/exit event	Send/Receive event	Counter event
Data	Stream number	Message type, Message Length	Counter value

Table 5.1: Event type, Event class and Event data definition for OTF files

5.3 Analysis of Open Trace Format (OTF) Files in Eclipse

The OTF library supports efficient parallel and distributed access to trace data and offers selective reading access. It implements different handlers written in C++ which facilitates access to the different record files as described in Section 5.1. The handlers provide access to OTF traces which consists of multiple streams. The Application Programming Interface (API) consists of several components to support the reading and writing of OTF trace files. The high level trace read and write interfaces called Reader and Writer handles the whole traces. Both these interfaces are controlled by the Master which manages the streams. The read and write access for single streams is done by

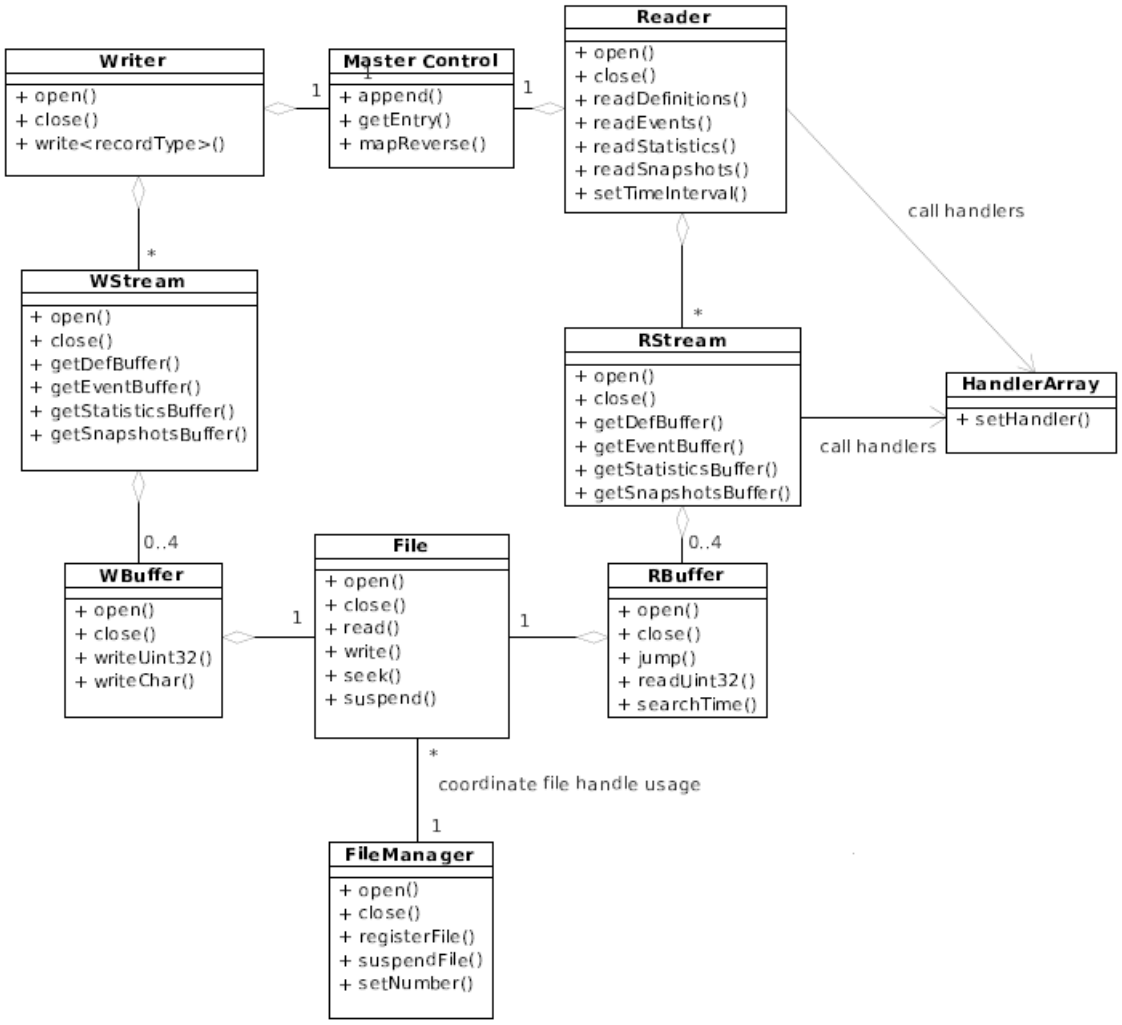

Figure 5.2: Overview of OTF API [1]

the RStream and WStream. The low level access to files is handled by the RBuffer and WBuffer. The FileManager makes sure that only necessary files are opened at any given time. An overview of the different classes and interfaces in OTF API is seen in Figure 5.2.[1]

Apart from the interfaces mentioned above, there are several handlers which can be implemented to read specific file records.

5.3.1 Java Native Interface

In order to read the information from trace files in Eclipse, we make use of Java Native Interface(JNI). The JNI integrates a Java application with legacy code written in C or

C++ and makes language interoperability easy. JNI enables Java code running in a Java Virtual Machine like Eclipse to call and be called by libraries written in C or C++. The language interoperability is provided by writing native methods which can then access the Java objects in the same way that Java code uses these objects [33].

5.3.2 Mapping between Definition Records and Event Records

Like mentioned in the previous sections, the definition records provide references for the event records. To obtain meaningful information by reading the handlers which implement the event records and the definition records, they should be mapped. In the prototype tool being developed, the handlers are employed using native methods in Java to read in the data from the definition records and event records. The data read in is then mapped using the reference numbers. E.g., the event record for function entry/exit may contain information as follows:

ENTER FUNCTION Time = 179399, Process number = 4, Function number = 62, Source code location identifier = 0
LEAVE FUNCTION Time = 975547, Process number = 4, Function number = 62, Source code location identifier = 1

The definition records for the function then contains 2 handlers, one for the function and other for the function group as follows:

Stream = 0, Function number = 62, Function name = switchOut_RT, Function group number = 7, Source code location identifier = 0
Stream = 0, Function group number = 7, Function group name = Task Switch

As seen from above, the reference between the event record and the function definition record lies in the function number. Similarly, the reference between the function definition record and the function group definition record lies in the function group number.

For the filtered information to give a complete meaning of the program being traced, the information between these handlers has to be mapped. Mapping of the event records and definition records along with the filtering according to Physical and Logical trace brings out useful information which can be used for further analysis.

5.3.3 Physical Trace Filtering

As described in the Chapter 3, the physical trace can be obtained by filtering according to the event attributes like timestamp, event type, event class or process number. The event types and event class as applied to OTF trace files are illustrated in Section 5.2.

5.3.4 Logical Trace Filtering

The OTF files used for testing mostly consisted of function entry/exit events, counter measurements and communication. Therefore, a logical relation to test the three scenarios

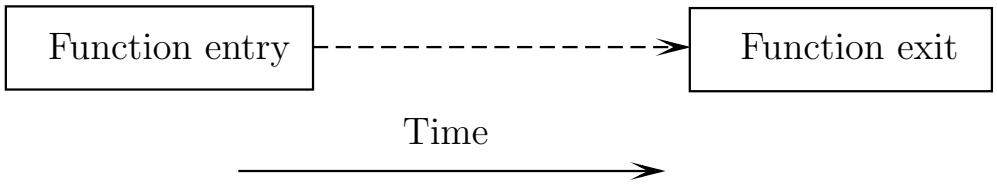

Figure 5.3: Scenario I: Case 1

was drawn up as seen in the following sections.

Scenario I : **Function Call Mapping**

1. *Case 1: Identifying the Entry and Exit of a Function*

   Given a function name and timestamp, we get the process number (context) and locate the event of interest. The events that occurred in the same process, between the time, the function entered and left are obtained. This can illustrate the causal relation between events, i.e., it can show the functions which were called by the function of interest before it left the system. Each function enters using a unique function number and leaves by entering a '0' or the same function number, it used for its entry. When the function enters and leaves by using the same function number, it becomes easy to track the entry/exit of the function. But when the function exits by entering a '0', the first '0' that is encountered after the function enters is treated as exit for the function in question. This assumption is proven correct in most of the cases. System call functions mostly use the '0' to indicate the exit of the function. The scenario is illustrated in Figure 5.3.

2. *Case 2: Identifying the events that occurred between the entry of one function and the exit of another function on same process/different process*

   In multithreaded object oriented programming, asynchronous function calls are used to avoid long stalls due to waiting for a particular method or function to complete execution. When an application makes a call to the asynchronous method, the execution of the asynchronous method happens in parallel to the process which runs the application.

   In OTF files, this behaviour can be analysed by collecting events which happened between the entry of a function and exit of another function. The asynchronous call becomes apparent when a call made to one function, exits by leaving another function. This can be tracked by locating a particular event using the function name and timestamp and looking for all events until another function leaves the system. The 'OR' part in the tool box with the option to enter another function name is meant to solve this purpose.

   Filtering to analyse asynchronous function calls can be done for events that happen on the same process or different process. And both the possibilities are implemented in the tool box. The Figure 5.4 shows the case where both the function 1 and function 2 happen on the same process and the Figure 5.5 shows the case where the functions 1 and 2 happen on different processes.

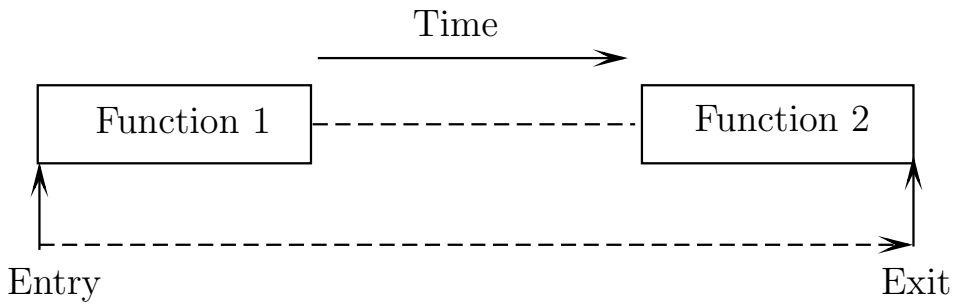

Figure 5.4: Scenario I: Case 2

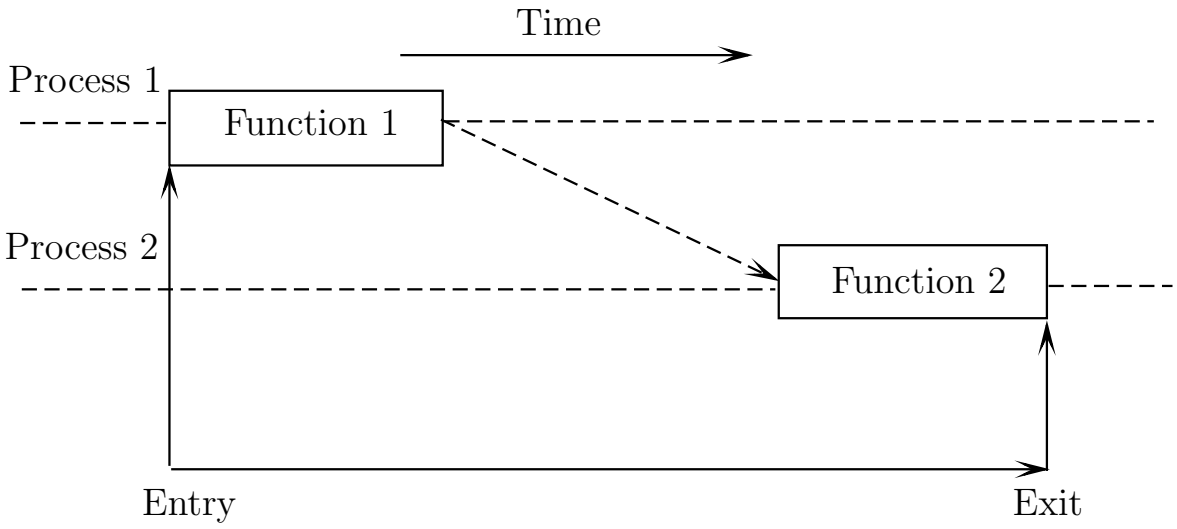

Figure 5.5: Scenario I : Case 2

3. *Case 3: Inclusion Relation*

   Inclusion relation in the case of OTF trace files, can be explained with the grouping of functions or counters or processes. The unique function numbers, counter numbers and process numbers are grouped according to the functionality they offer. This indicates the inclusion relation where a certain includes members of the same type.

   Consider a function group called 'Interrupt requests(IRQ's)', the functions included within the function group include soft IRQ's and IRQ handler. Therefore, analysing IRQ's can be made easier by analysing the functions soft IRQ's and IRQ handler. In the tool box, this analysis can be done by filtering the events according to the event class viz. function group in the case of functions.

Scenario II : **Scheduling**

In a multicore and multitasking environment, resources are shared between vari-

ous threads and processes based on a scheduling algorithm. The load balancing in the system becomes effective and optimal when suitable scheduling mechanisms are used. In the trace files, scheduling is some times recorded by use of counters. The counter measurement shows the migration of threads or processes from one core to another over a period of time. In order to study the migration of tasks on cores and also the effect of thread switching on cores, the counter values are studied.

In the Corvus tool box, when a counter name is given, the behaviour of that counter in the given time is illustrated using the trace visualisation. The causal relation of a thread switching out and another thread switching in can be studied using the obtained information.

Scenario III : **Communication**
Events can be correlated according to the send and receive processes, which indicate a complete communication event and hence a logical relation.

Since the available test cases did not have explicit communication events, the logical relation was not tested. But handlers and functions have been implemented to test for the send and receive communication process as well. In the test cases which were used for testing, most of the send or receive communication were recorded as function call, which can be analysed using the scenario I.

5.4 Applicability of other Trace Formats in Eclipse

The EMF model described in Section 4.4 can be applied to other tracing formats like Common Trace Format (CTF) and Paraver which have almost the same structure as OTF. While the case for physical trace filtering becomes apparent due to the mapping between the different fields in the CTF to the EMF model, for logical filtering specific use cases needed to be implemented which suit the specific trace format. The use cases implemented for the OTF trace files may not be applicable to traces from the CTF and Paraver trace format. The trace file format of CTF and Paraver and their application to the EMF model is discussed in the following sections.

5.4.1 Common Trace Format (CTF)

As described in Section 2.3.2, the Common Trace Format (CTF) satisfies the needs of the embedded, telecom and high-performance communities. CTF uses raw binary encoding to store the trace files and the Trace Stream Description Language (TSDL) provides description of various trace stream layouts.

A meta-data file holds the information about the trace version, trace event types and other related data described using the TSDL. The CTF event model contains event streams divided into contiguous event packets of variable sizes. Each stream has a stream header which is repeated at the beginning of each event packet in the stream. The event stream header is mostly referred to as the ‘event packet header’. The event packet header

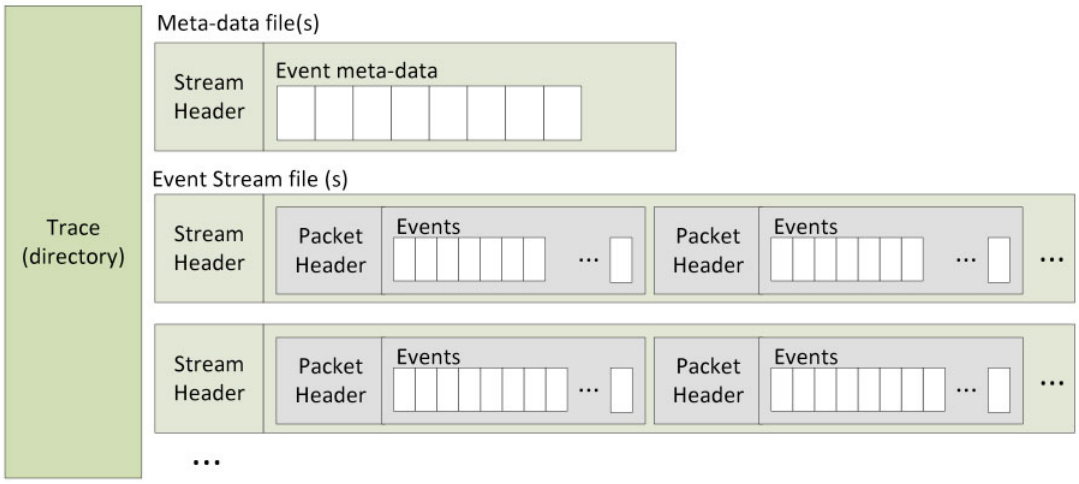

Figure 5.6: CTF file structure overview

then consists of two parts: The first part is the same for all streams in the trace and the second part is the 'event packet context' which is unique for each stream in the trace. Event packets within a stream are of variable size, with optional padding at the end. A simple overview of the CTF trace format is shown in Figure 5.6.

Each event in the trace consists of the following parts [13]:

1. *Event Header*: Event Header is defined in the meta-data file and is a numeric identifier that identifies the 'class' of an event within the trace stream.
2. *Stream Event Context and Event context*: Event context is again a numeric identifier and the definition is use-case specific , it may be a "core", may be a process id, thread id, etc.
3. *Event Timestamp*: The size and meaning of a timestamp is described in the meta-data. It may be omitted if not available for the use case.
4. *Event Payload*: An event payload contains information specific to the given event type and describes the details of the encoding, size, and interpretation of the various fields in the events.

5.4.1.1 Application of CTF to EMF model

The CTF file format is very much suitable to the EMF Model and can be easily applied to perform the analysis on the traces. Table 5.2 illustrates the association between the various terminologies in the CTF format and the EMF model. While the event attributes in the EMF model are matched to different fields from the CTF file for physical trace filtering, the logical filtering can be performed when specific use cases are implemented. These use cases are based on the idea about which information about a logical relation will be helpful when considering the CTF file format for traces.

EMF model	**Common Trace Format(CTF)**
Timestamp	Timestamp
Event type and Event class	Event header
CPU ID, Thread ID or Process ID	Event context and Stream Event context
Data	Event payload

Table 5.2: CTF application in EMF model

5.4.2 Paraver Trace File Format

Paraver traces contain a sequential description of the events and state changes of the application. The basic time unit is the nanosecond, but it can also be interpreted as the CPU cycles.

The Paraver trace file format consists of the header and the three records namely state records, event records and communication records. The header file contains global information about the trace and the communicators. The global information includes the time of generation of the trace, a description of the system and the number of applications shown in the trace. A communicator in Paraver trace format is used to group processes and contains data about the application ID, communicator ID, number of tasks and processes in the communicator.

The state records indicate the current state of the object i.e., idle, running or dead as shown in Figure 5.7. The event records contain information about the timestamp, event type and the event value. Each event type must have a unique event value and multiple event types must be defined in order to hold the multiple values if needed. The format of event record is shown in Figure 5.8.

```
1 : cpu:appl:task:thread : begin_time:end_time : state
```

Figure 5.7: State Record format in Paraver [2]

```
2 : cpu:appl:task:thread : time : event_type:event_value
```

Figure 5.8: Event Record format in Paraver [2]

The communication records contain the logical and physical send and receive times for a communication event. The logical time indicates the timestamps as observed by

```
3: sender : Isend:Psend : receiver : Irecv:Precv : size:tag
```

Figure 5.9: Communication record format in Paraver [2]

the application and the physical times indicates the timestamps as it actually occurs in the machine. The format of the communication record is shown in Figure 5.9.

5.4.2.1 Application of Paraver to EMF model

The association between the communication records and event records in Paraver to the event attributes in the EMF model is illustrated in Table 5.3. As mentioned before, logical trace filtering can be done by implementing case specific use cases for Paraver trace files.

EMF model	**Paraver Trace file format**	
	Event record	**Communication record**
Timestamp	Isend/Psend, Ireceive/Preceieve	Timestamp
Event Type	Send/Receive	Event type
Data	Event value	Message size
Context	CPU ID, Application Number or thread number	Sender/receiver

Table 5.3: Paraver application in EMF model

5.5 Summary

In this chapter, the complete structure of the OTF files is given, followed by a prototype implementation of the EMF model described in Chapter 4 with the OTF format files. The different scenarios and use cases implemented to test the classification of logical trace in the OTF files is provided. The trace format of CTF as well as Paraver are elaborated followed by a description of the possible application of the CTF and Paraver formats to the EMF model. The next chapter gives the results obtained when applying test cases of OTF file format to the EMF model.

6 Results

This chapter discusses the results obtained while testing the EMF model with OTF trace files in Eclipse. The test cases used to test the model are described in Section 6.2. The EMF in EMF is tested for all the use cases and scenarios as described in Chapter 5. The GUI design and the trace visualisation using processing tool are described in Sections 6.1 and 6.5 respectively.

6.1 The Corvus Tracing Tool Box

The GUI design for the Corvus tracing tool box is done using WindowBuilder pro. WindowBuilder is a widely used and easy to use bi-directional Java GUI designer. Layout generated using Visual interface is automatically converted into Java code. With WindowBuilder, it is possible to build Jface, SWT, RCP or XWT applications. The JFace application window layout is used to create the GUI for the tracing tool box. A snapshot of the GUI developed for the Corvus tracing tool box is shown in the Figure 6.1.

At any given time, filtering is possible only for one attribute which can be either from the Physical Trace or the Logical trace. The time range to collect the files from OTF has to be specified compulsorily irrespective of the attribute chosen for filtering.

6.1.1 Reading OTF Files

The OTF files are read into the Eclipse Environment using the Java Native Interface(JNI). On the C++ side, a call-back function mechanism is employed to read the contents of the trace files. This introduces an overhead on the time spent in reading the entire trace files in Eclipse for large files. For small files of size 20MB or less, the time taken in reading into Eclipse is minimal and bearable. When the file size increases, the time spent in reading into Eclipse increases and hence causes a lag in processing the trace files using the tracing tool box. To compensate for this lag, we read the trace files only within a certain time range specified by the user. This way, the amount of time spent in the reading through the call back in the native methods is reduced considerably. The time interval set by the user is handled using the handler for global reader, **OTF_Reader_setTimeInterval**. Since the trace files are mostly very large in size, specifying a time range to read the files into Eclipse makes processing faster.

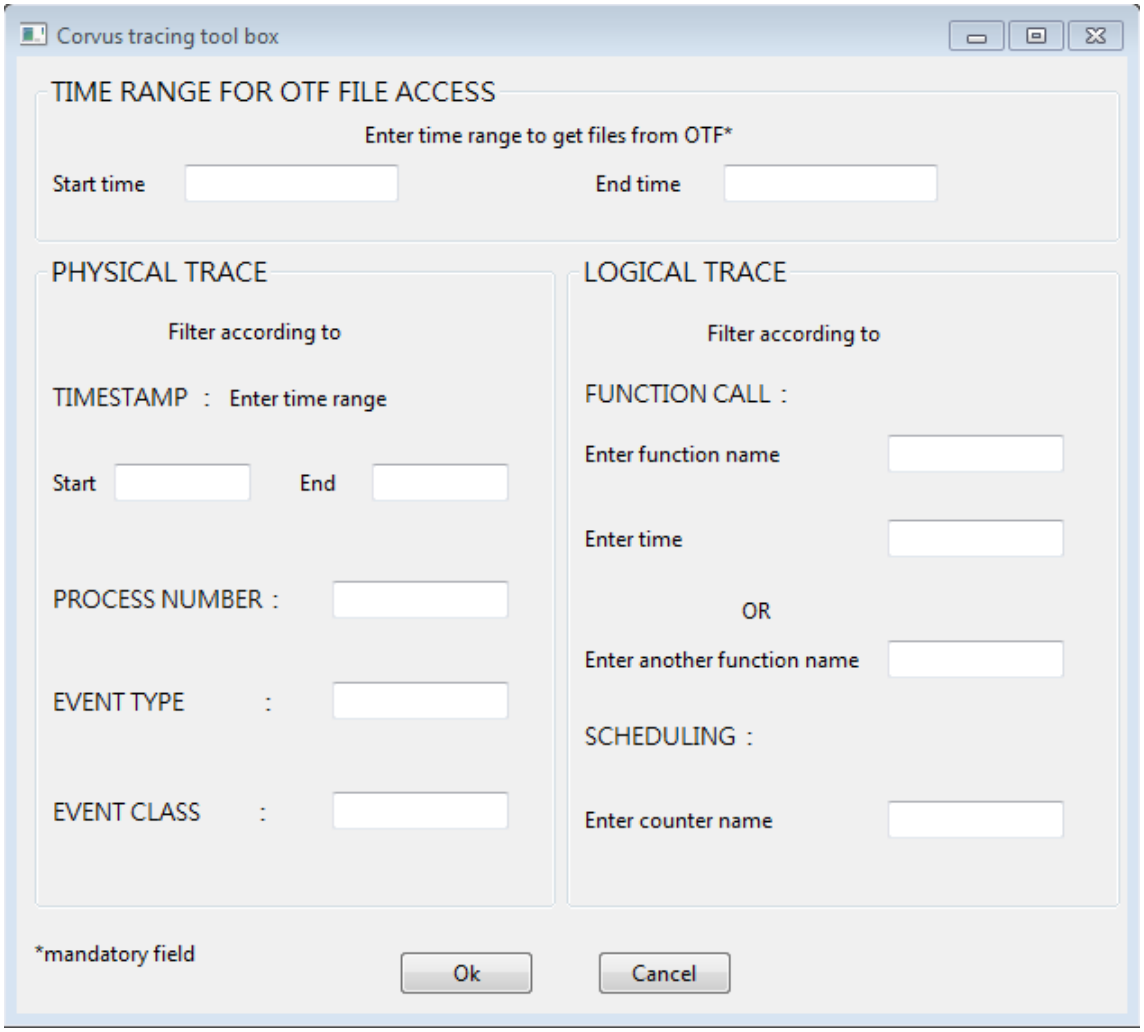

Figure 6.1: Corvus Tracing Tool Box

6.2 Test Cases

Two OTF trace files were used to conduct tests on the various scenarios as described in the Section 5.3. A detailed description of the application from which the traces where obtained is given in the following sections.

6.2.1 Test case I

The first test case trace was from a simple application which was running on a dual-core machine. There were two kind of tasks, the General Purpose (GP) tasks and the Real Time (RT) tasks. Software interrupt requests are used to handle these tasks and marker records are generated to locate when the tasks miss a certain deadline while executing.

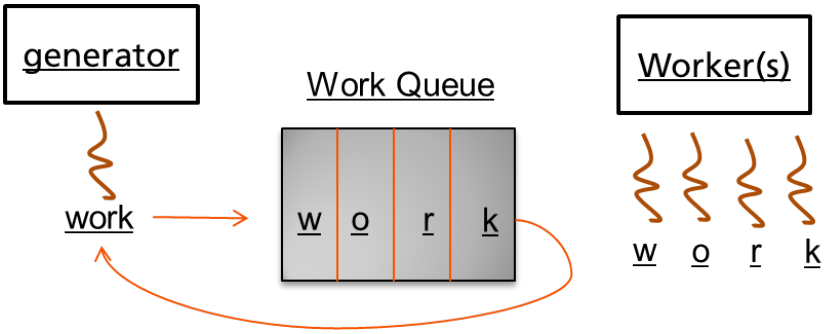

Figure 6.2: Overview of sample trace file used for testing

6.2.2 Test case II

The second trace file is based on a multithreaded application with Mutexes and also involves scheduling of tasks. The Generator threads adds a work item in to the queue, which is then retrieved and processed by the worker thread. A simple representation of the working of the second trace file is shown in Figure 6.2. And a work represents jobs that are scheduled and executed by workers. The first file is of size 1MB, while the other file is of size 89MB.

6.3 Sample Trace File

This section shows the sample records from the trace files in Test case I and Test case II. Figure 6.3 shows the event record file from the test case I. The figure shows the entry or exit of a function by the keywords 'Enter' or 'Leave', the timestamp values, the process number, the event type (in this case, the function number) and the Source Code Location (SCL) identifier (which refers to the 'Data' in the EMF model).

Figure 6.6 shows the counter record files from the trace files in Test case II. The figure shows the timestamp, the process number, the counter number and the counter value fields. Figure 6.4 shows the definition record files for the same trace file, the function numbers in the event trace file correspond to the function name in the definition record files. Similarly, Figure 6.5 shows the Function group definition records and the function group numbers in the function definition record files are assigned a name in the function group definition records.

We will use the test case I to show the physical trace filtering and the logical trace filtering according to function call and asynchronous function call. The logical filtering for scheduling is illustrated using the test case II. Since the test cases contain information about only function calls and counter measurement, we treat those only in the following sections and the case of communication is not illustrated.

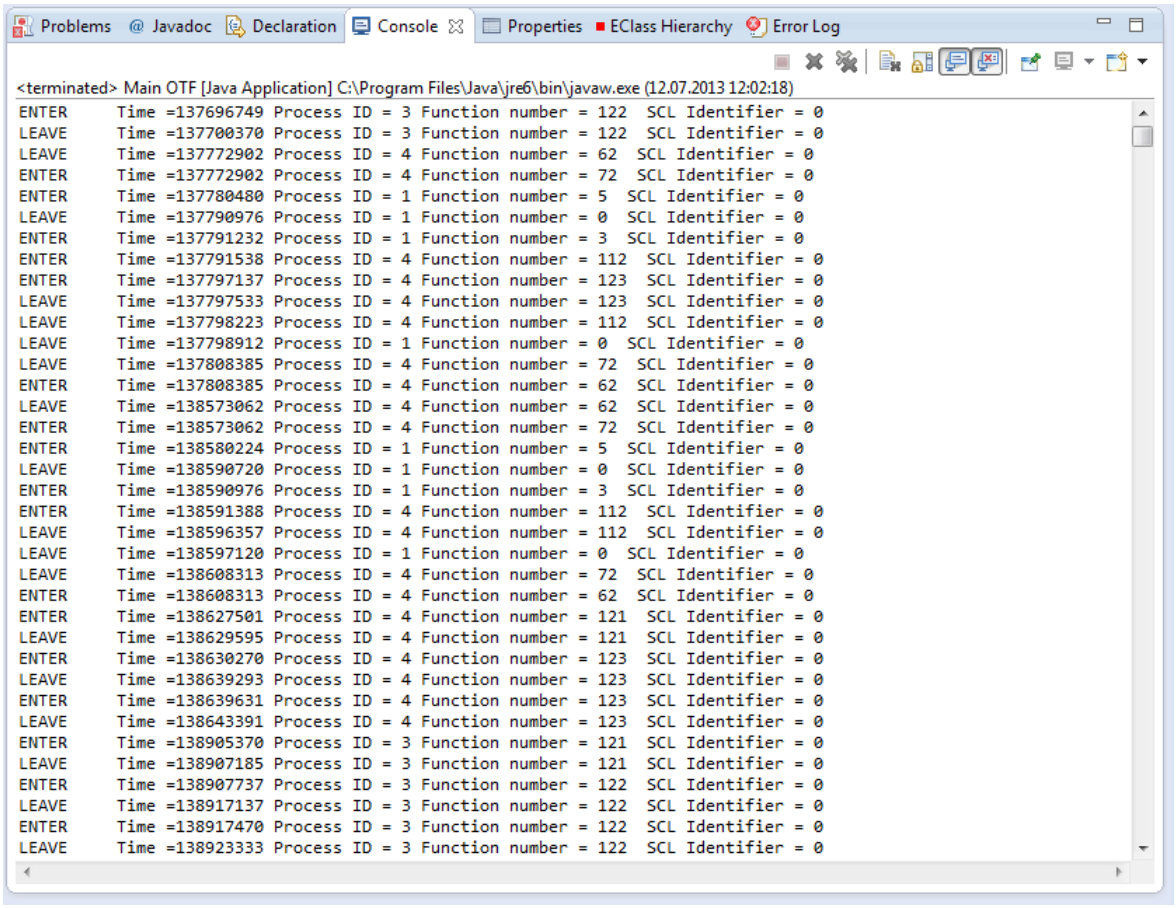

Figure 6.3: Sample event trace file from Test case I

6.3.1 Physical Trace Filtering

The physical trace filtering according to the event attributes is performed as seen in the following sections.

1. *Time Range Filtering*: For the given event record from test case I, we can filter for any given time range. E.g., we can specify the start time as 137000000 and stop/end time range as 138500000 and obtain all events that happened between this time range.

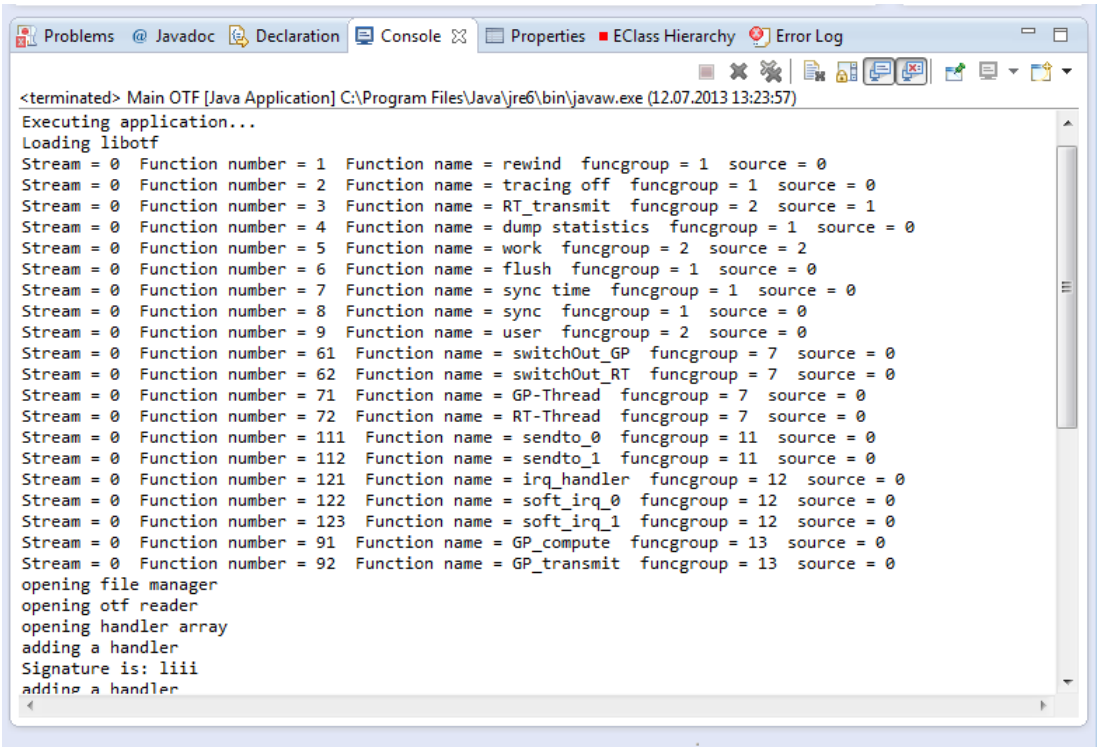

Figure 6.4: Sample function definition record file from Test case I

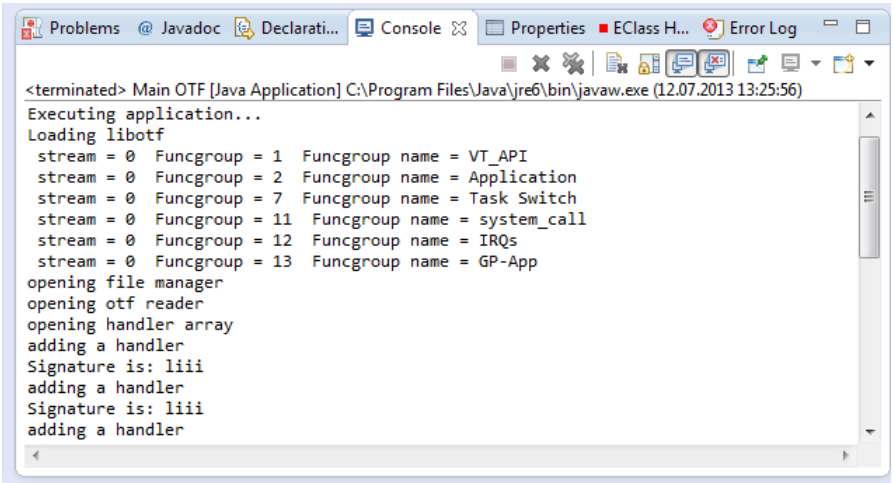

Figure 6.5: Sample function group definition record file from Test case I

2. *Process Number Filtering*: The given snapshot of the event record from test case I shows the process numbers 1, 3 and 4. By specifying any one of these numbers, we can filter the events which happen on that particular process for the entire trace file.

3. *Event Type Filtering*: Event type filtering in OTF can be regarded as filtering

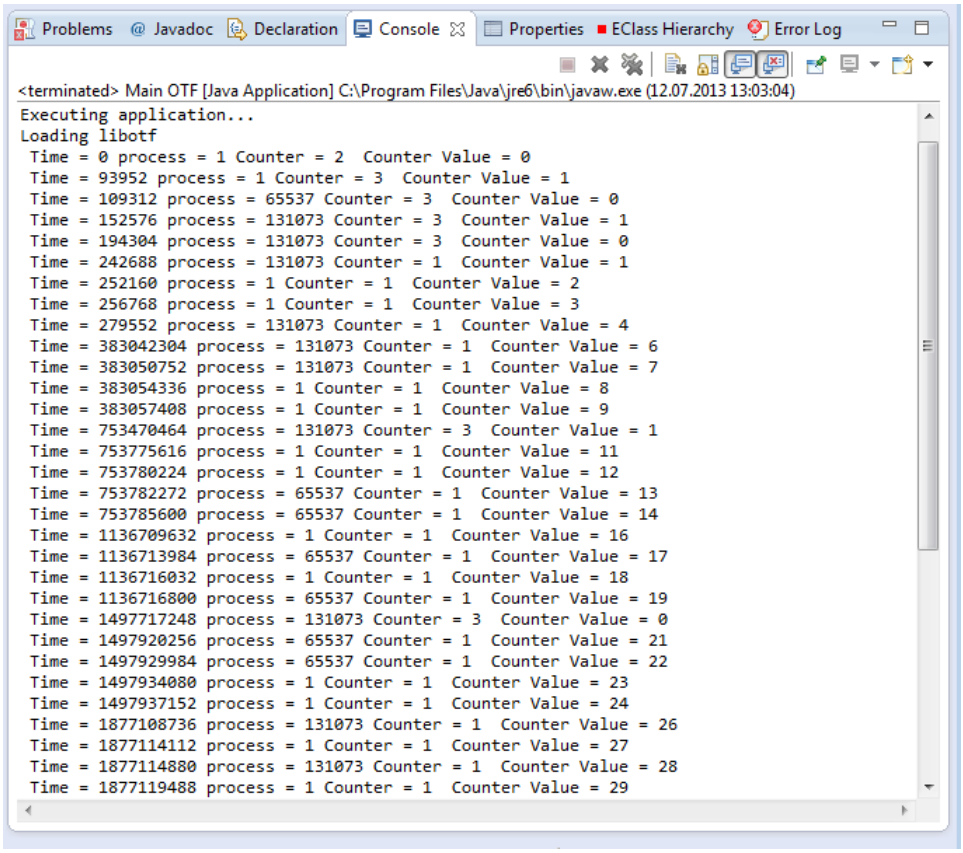

Figure 6.6: Sample counter record trace file from Test case II

according to Function number(name), Send/Receive event and Counter event. In the Figure 6.7, we show a filtering operation for the function entry/exit scenario where we specify a function name work and obtain all occurrences of that event in the trace file.

4. *Event Class Filtering*: Like Event type filtering, Event class filtering in OTF files can be done by giving the Function group name, Message group name or Process number as input to the filter function. In the given snapshot of function group definition records from test case I, we can see that there are several function groups and their corresponding function group names. By giving any of those function group names as input, we can obtain the functions that are contained in that group and the time of occurrence and other attributes from the event record.

6.3.2 Logical Trace Filtering

Logical trace filtering in the OTF files can be done for the three scenarios as described in the previous chapter.

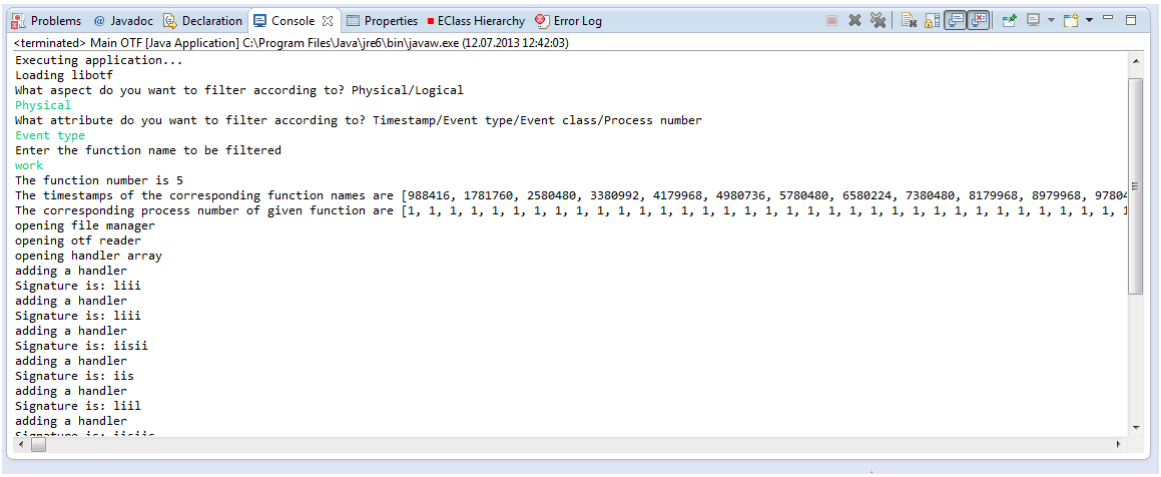

Figure 6.7: Result obtained when filtering according to a given Event type

1. *Scenario I*:

   (a) Case 1 Filtering for events between the entry and exit of a function is possible. In Figure 6.8, we filter for all the events between the entry and exit of the function RT-Thread which corresponds to the function number 72.

   (b) Case 2 Filtering according to asynchronous function calls, can be done similar to the filtering according to entry and exit of functions. Only difference here is that we look for exit of different function rather than the same function.

   (c) Case 3 Inclusion relation in the case of function calls is exactly the filtering according to 'Event class' attribute in the physical trace filtering.

2. *Scenario II*: Information about scheduling can be obtained by looking at the counter values as described in Section 5.3.4. The process migration between core 0 and core 1 is illustrated in the Figure 6.9.

3. *Scenario III*: The scenario of communication is not explicitly seen in the given test cases. However, the sending and receiving process is recorded as functions in these test cases, which can be filtered using the 'Event type' attribute in the physical trace.

6.4 Trace Analysis using the Corvus Tracing Tool Box

The prototype tool can not only be used to filter the trace data and classify them. It can also be used for debugging and analysis purposes. By using logical filtering, we can identify any abnormal behaviour and filtering according to physical phenomena

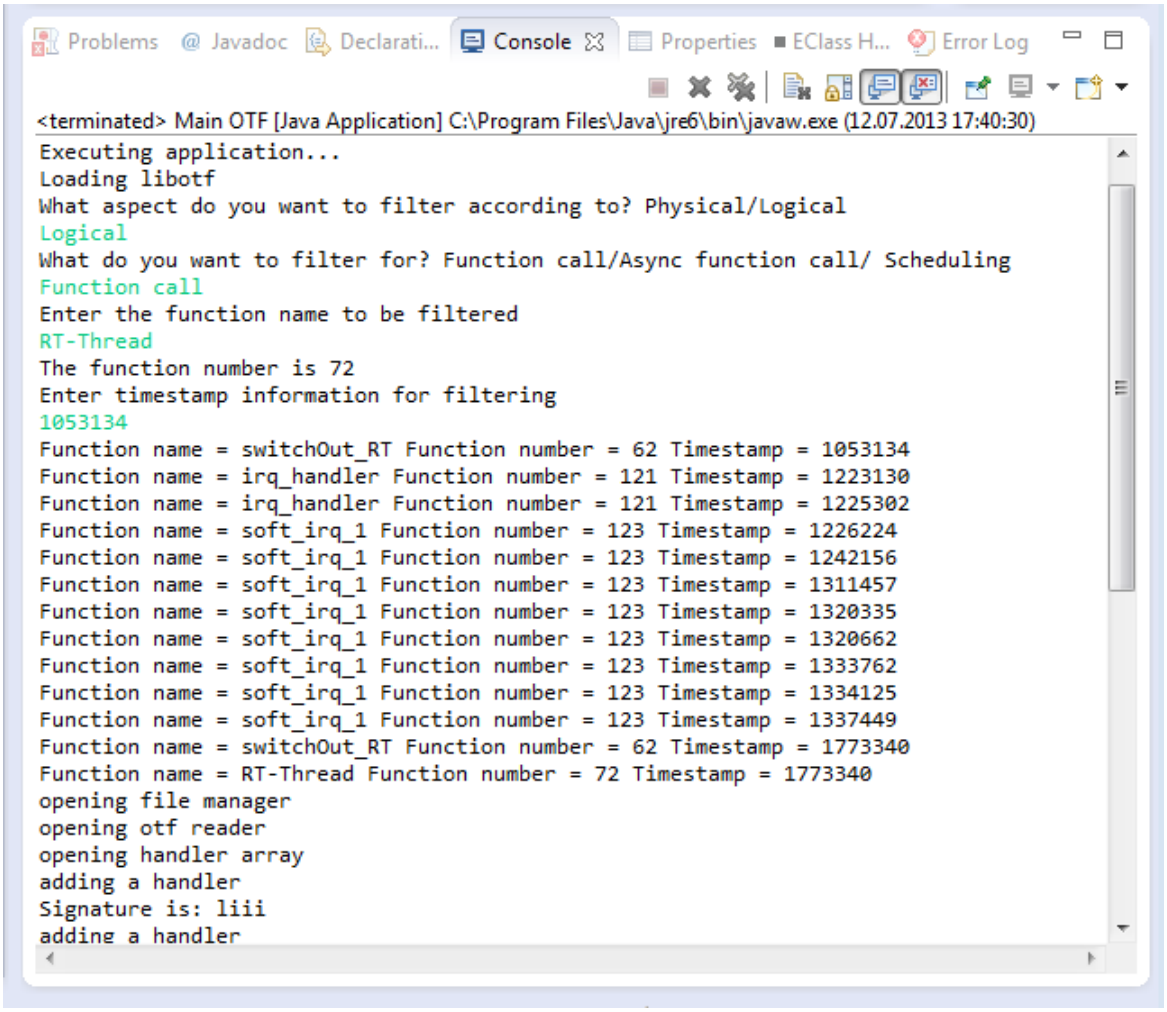

Figure 6.8: Result obtained when filtering according to a given Function name

can provide details about RAW, WAR and WAW hazards in the given trace file. In the case of function call logical filtering, abnormal behaviour can be identified when the control shows entry of a function but no exit.Similarly, abnormal behaviour of scheduling algorithms can be identified using the scheduling logical filtering.

6.5 Trace Visualisation

A simple trace visualisation is created for the filtered information from the tracing tool box using the Processing tool. Processing is an open source programming language and environment used to create images, animations and interactions. The visualisation in this case, is a series of rectangular blocks displaying the filtered information. This format of trace visualisation has been inspired by the Vampir toolset which uses a similar structure. However, the drawback behind such a visualisation using Processing is that it does not

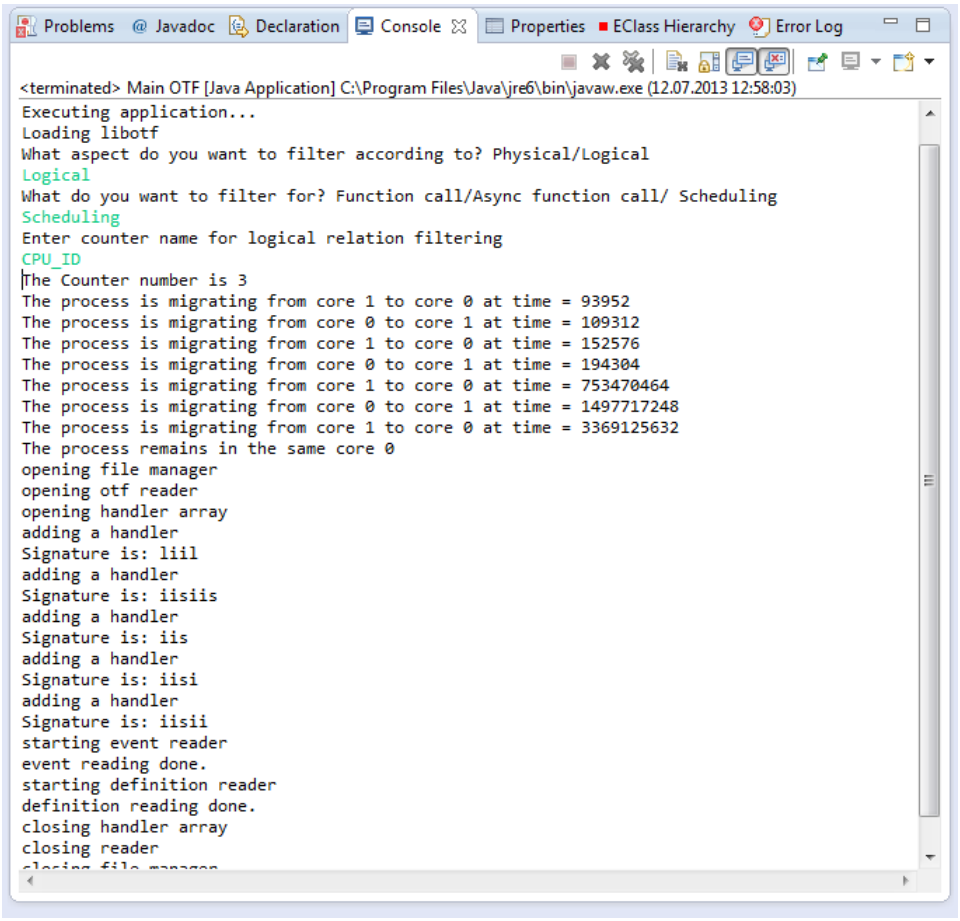

Figure 6.9: Result obtained when filtering according to a given counter name

a unified time line to represent the filtered information, instead each rectangular box is considered as an event happening at the given time.

The aim of this thesis was classifying the trace files into Physical and Logical trace streams and not visualisation of the filtered information. Hence, no concrete method for visualisation was implemented. The filtered information was viewed using the console in the Eclipse platform.

An example is illustrated in the Figure 6.10 which contains the filtered information according to the Timestamp attribute in Physical trace. Each block represents a unique event in the filtered interval. The numbers 1, 4, 4, 1, 4 in the figure represent the process numbers where the event actually took place. The long data values in the figure indicate the timestamp values and the function names such as RT_transmit, send_to_1 and RT_Thread are indicated in the next line.

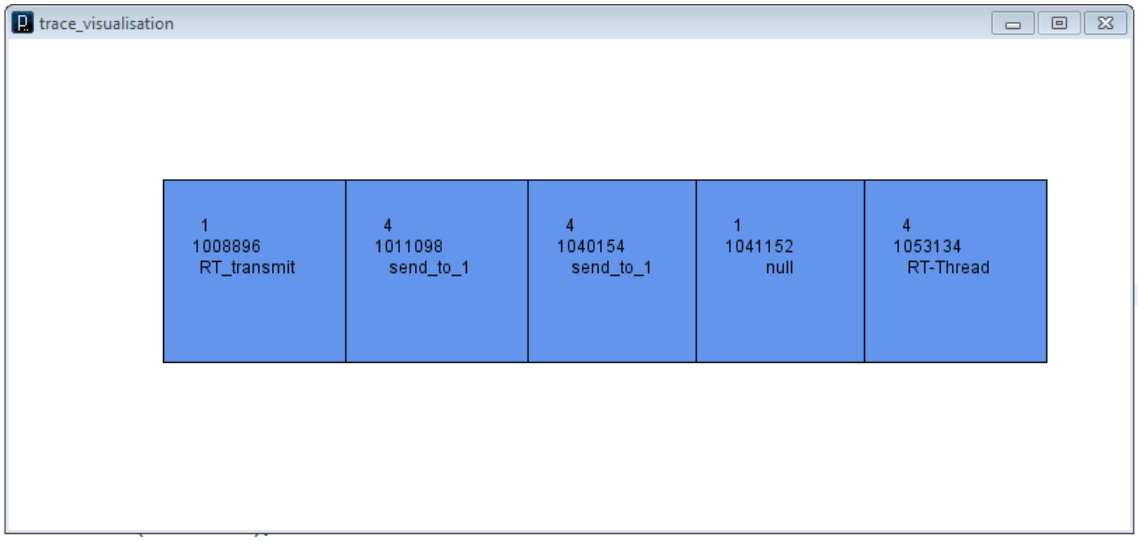

Figure 6.10: Trace Visualisation example

6.6 Summary

This chapter discusses in detail the results obtained when testing the EMF model with the OTF files. The trace streams from the OTF files are classified into physical and logical trace streams based on the event attributes and the logical relation between them. The logical relation is defined by specific scenarios and the results obtained are discussed in Section 6.3.2. The GUI Design for the Corvus tracing tool box and the simple trace visualisation using processing is described in brief.

7 Conclusions

This chapter concludes the thesis work highlighting the goal of the project that was achieved and the main contributions in Section 7.3. A summary of the previous chapters is provided in Section 7.1 and the problem statement is revisited in Section 7.2 and the answer to the questions in the problem statement are answered. Section 7.4 gives some recommendations for future work.

7.1 Summary

A summary of all the previous chapters is provided here followed by an explanation of how the goal was reached.

Chapter 2: This chapter provides the necessary background information about software tracing in multi core environment. It discusses the various steps involved in the process of software tracing, i.e., trace collection, trace data format, trace analysis and trace visualisation. The main focus of this thesis is on the trace analysis by applying a suitable model to classify the trace data. A brief description about the related work in formal methods of modelling events in a trace and its uniqueness is also given. As discussed in Section 2.6, the method of analysis that has been in practice so far depends on specifying assertions using certain assertion/annotation languages and then checking for expected or faulty behaviour. In this thesis, we aim at developing a model which describes the events in general and then classify the trace streams in order to facilitate better program understanding for debugging and optimisation purposes.

Chapter 3: This chapter provides a complete understanding of the discrete elements of activity in a trace stream, i.e., the events. It describes the various attributes of an event in detail and analyses the importance of each, which then forms the basis of the model being developed. The classification of trace streams into physical and logical trace streams is given a clear definition. The logical relationship between the events in a trace which gives the complete meaning to the flow of program execution is presented and the 4 major possible relations, causal, happens-before and inclusion are explained with examples.

Chapter 4: This chapter materialises a general model to analyse the traces based on the description of events and the classification provided in Chapter 3. The Eclipse Modelling Framework (EMF) is chosen to develop this model and a brief description of the features provided by the EMF are discussed. A model which will be used to analyse the traces from the different trace file formats which was developed using UML and the various blocks in the UML model which represent the various aspects of the classification

and hence the analysis are described in detail.

Chapter 5: In this chapter, the complete structure of the Open Trace Format (OTF) files is given, followed by a prototype implementation of the EMF model described in Chapter 4 with the OTF format files. The different scenarios and use cases implemented to test the classification of logical trace in the OTF files is provided. The trace format of Common Trace Format (CTF) as well as Paraver are elaborated followed by a description of the possible application of the CTF and Paraver formats to the EMF model.

Chapter 6: This chapter discusses in detail the results obtained when testing the EMF model with the OTF files. The trace streams from the OTF files are classified into physical and logical trace streams based on the event attributes and the logical relation between them. The logical relation is defined by specific scenarios and the results obtained are discussed in Section 6.3.2. The GUI Design for the Corvus tracing tool box and the simple trace visualisation using processing is described in brief.

7.2 Revisiting the Problem Statement

We can conclude the complete work by answering the research questions described in Section 1.4.

1. **Question 1**: How to use the trace data in a meaningful way?

   Answer: In software tracing, it is not always possible to anticipate the program behaviour and do a clever instrumentation of the program to obtain information. As a result, the trace data collected during the program execution has information which might not be useful. In order to identify the useful information from the traces, we need to locate the useful information and classify them by using a generalised model as described in Chapter 3. The model described in this thesis is based on the concept in which the trace data are stored, i.e., many events in a trace file. The events are assigned certain attributes which help in identifying the useful information in a trace.

2. **Question 2**: How to classify and analyse the vast amount of information collected as execution traces?

   Answer: After identifying the useful information from the event traces, it is essential to describe a classification method which can then be applied to all types of trace files in embedded multicore domain. For this purpose, a classification of the events into 'Physical Trace' and 'Logical Trace' is suggested. The main idea behind this classification is to give a complete understanding of the traces in both the physical and logical aspect. The 'Physical Trace' contains trace files filtered according to the physical aspects like Timestamp, Process Number, Event type and Event class as described in Chapter 3. The 'Logical Trace' provides information about the logical relationship between any two events in a trace. Logical trace filtering can be done based on specific use cases mentioned for the type of information stored in the trace files.

3. **Question 3**: Will a generalised model help to analyse the trace data?

   Answer: A generalised model based on the classification of traces is developed and a prototype implementation is given in Chapters 4 and 5 respectively. The notion of 'generalised model' is based on the fact that the model developed can be tested with trace files which follow the formats of OTF, CTF and Paraver, which are commonly used in the embedded multicore field. By using the generalised model developed in Eclipse Modelling Framework (EMF), the trace files of the OTF format were tested for classification into physical trace and logical trace and the filtered information provided details of the program behaviour and helped in debugging the program.

7.3 Main Contributions

The goal of this thesis work was to propose a method to classify and analyse the trace stream information, which in turn helps in the better understanding of the program behaviour. For this purpose, a theoretical model which explains the characteristics of events in a trace and the relationship between them was given and thereby a prototype tool which implements the model was produced. The goal was successfully achieved and the main contributions made as a result of this thesis work are as follows:

1. A model and a classification method for trace data which meets the needs of embedded multicore domain was developed. The classification of trace streams into physical and logical traces helps to understand the program behaviour better, which in turn is helpful in debugging purposes and performance optimisation in the embedded multicore field.

2. A prototype implementation to illustrate classification was created and tested for various test cases using Eclipse Modelling Framework (EMF).

3. Use of the prototype tool for debug and analysis purposes was demonstrated.

7.4 Recommended Future Work

The logical filtering for the trace analysis using OTF files can be improved with other different use cases and tested. Logical relation adds a complete meaning to the classification of traces and hence all possible logical relation scenarios should be tested to provide complete functionality for testing purposes. The EMF model has been tested only for OTF trace files. Application of other trace formats like Common Trace Format (CTF) and Paraver to the EMF model can be tested when a suitable JNI interface is written to read from the trace files. It is important to test the EMF model with files from other trace formats, in order to obtain maximum advantage of the generalised model and its classification.

The thesis work concentrates on implementing the developed model in the prototype tool. The aspect of statistical and behavioural analysis on the traces can be used to

model the system behaviour over a period of time. By analysing the trace data over a period of time, a machine can be made to identify between the expected behaviour and abnormal behaviour. The model developed in this thesis can facilitate modelling of a learning behaviour. The principles of machine learning may be employed to perform behavioural analysis on the filtered information from the EMF model. The results of studying event traces will be helpful in building intelligent systems and software in the future, which can learn from the behaviour of the program to locate the abnormalities in the system.

Bibliography

[1] H. B. A. Knüpfer, H. Brunst, A. D. Malony, and S. S. Shende, "Open trace format api specification version 1.1," *Center for High Performance Computing University of Dresden, Germany*, 2006.

[2] Aramirez, "Paraver trace file format." `http://pcsostres.ac.upc.edu/eitm/doku.php/paraver:prv`. [Online; accessed 01-June-2013].

[3] M. McGavin, T. Wright, and S. Marshall, "Visualisations of execution traces (vet): an interactive plugin-based visualisation tool," in *Proceedings of the 7th Australasian User interface conference-Volume 50*, pp. 153–160, Australian Computer Society, Inc., 2006.

[4] Wikipedia, "Tracing(software)." `http://en.wikipedia.org/wiki/Tracing_(software)`. [Online; accessed 21-June-2013].

[5] D. Garlan, F. Bachmann, J. Ivers, J. Stafford, L. Bass, P. Clements, and P. Merson, *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2nd ed., 2010.

[6] X. Zhang and R. Gupta, "Whole execution traces," in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pp. 105–116, IEEE Computer Society, 2004.

[7] M. Kreutzer, "Development of complex multicore systems: Tracing challenges and concepts," tech. rep., Mentor Graphics, 2009.

[8] B. Cornelissen, A. Zaidman, A. van Deursen, and B. Van Rompaey, "Trace visualization for program comprehension: A controlled experiment," in *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pp. 100–109, IEEE, 2009.

[9] S. P. Reiss and M. Renieris, "Encoding program executions," in *Proceedings of the 23rd International Conference on Software Engineering*, pp. 221–230, IEEE Computer Society, 2001.

[10] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk, "Execution trace analysis through massive sequence and circular bundle views," *Journal of Systems and Software*, vol. 81, no. 12, pp. 2252–2268, 2008.

[11] M. Kreutzer, "Software tracing tools and techniques for advanced multicore development," tech. rep., Mentor Graphics, 2009.

[12] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel, "Introducing the open trace format (otf)," in *Computational Science–ICCS 2006*, pp. 526–533, Springer, 2006.

[13] M. Desnoyers, "Common trace format (ctf) specification." `http://git.efficios.com/?p=ctf.git;a=blob_plain;f=common-trace-format-specification.txt;hb=master`. [Online; accessed 20-June-2013].

[14] V. Pillet, V. Pillet, J. Labarta, T. Cortes, T. Cortes, S. Girona, S. Girona, and D. D. D. Computadors, "Paraver: A tool to visualize and analyze parallel code," tech. rep., In WoTUG-18, 1995.

[15] P. Kruchten, *The Rational Unified Process: An Introduction*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 3 ed., 2003.

[16] H. A. Mller, M. A. Orgun, S. R. Tilley, and J. S. Uhl, "A reverse-engineering approach to subsystem structure identification," *Journal of Software Maintenance: Research and Practice*, vol. 5, no. 4, pp. 181–204, 1993.

[17] M. D. Ernst, "Invited talk static and dynamic analysis: synergy and duality," in *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '04, (New York, NY, USA), pp. 35–35, ACM, 2004.

[18] J. Bohnet, *Visualization of execution traces and its application to software maintenance*. PhD thesis, 2010.

[19] S. Mamone, "The ieee standard for software maintenance," *SIGSOFT Softw. Eng. Notes*, vol. 19, pp. 75–76, Jan. 1994.

[20] T. A. Corbi, "Program understanding: Challenge for the 1990s," *IBM Systems Journal*, vol. 28, no. 2, pp. 294–306, 1989.

[21] I. H. Witten and E. Frank, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[22] J. Han, M. Kamber, and J. Pei, *Data mining: concepts and techniques*. Morgan kaufmann, 2006.

[23] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Trans. Softw. Eng.*, vol. 29, pp. 210–224, Mar. 2003.

[24] T. Ilsche, J. Schuchart, J. Cope, D. Kimpe, T. Jones, A. Knüpfer, K. Iskra, R. Ross, W. E. Nagel, and S. Poole, "Enabling event tracing at leadership-class scale through i/o forwarding middleware," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pp. 49–60, ACM, 2012.

[25] M. Auguston, "Building program behavior models," *Engineering Automation for Reliable Software*, p. 35, 2000.

[26] S. Boroday, H. Hallal, A. Petrenko, and A. Ulrich, "Formal modeling of communication traces," in *ISTA*, pp. 97–108, Citeseer, 2003.

[27] M. Auguston and P. Fritzson, "Parforman-an assertion language for specifying behaviour when debugging parallel applications," in *Parallel and Distributed Processing, 1993. Proceedings. Euromicro Workshop on*, pp. 150–157, IEEE, 1993.

[28] C. Sadowski and J. Yi, "Tiddle: a trace description language for generating concurrent benchmarks to test dynamic analyses," in *Proceedings of the Seventh International Workshop on Dynamic Analysis*, pp. 15–21, ACM, 2009.

[29] R. Schwarz and F. Mattern, "Detecting causal relationships in distributed computations: In search of the holy grail," *Distributed computing*, vol. 7, no. 3, pp. 149–174, 1994.

[30] L. Vogel, "Eclipse modeling framework(emf) tutorial." `http://www.vogella.com/articles/EclipseEMF/article.html`. [Online; accessed 20-June-2013].

[31] D. Gallardo, "Developing eclipse plug-ins," tech. rep., IBM.

[32] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Addison-Wesley Professional, 2008.

[33] S. Liang, *The Java TM Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, 1999.