UPTEC IT 08 010
Examensarbete 30 hp
Juni 2008

# Correlation and Graphical Presentation of Event Data from a Real-Time System

Tobias Hedlund
Xingya Zhou

UPPSALA
UNIVERSITET

**Teknisk- naturvetenskaplig fakultet**
**UTH-enheten**

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

# Abstract

## Correlation and Graphical Presentation of Event Data from a Real-Time System

*Tobias Hedlund and Xingya Zhou*

Event data from different parts of a system might be found recorded in event logs. Often the individual logs only show a small part of the system, but by correlating different sources into a consistent context it will be possible to gain further information and a wider view. This would facilitate in finding source of errors or certain behaviors within the system.

This thesis will present the correlation possibilities between event data from different layers of the Ericsson Connectivity Packet Platform (CPP). This was done first by developing and using a test base application for the OSE operating system through which the event data can be recorded for the same test cases. The log files containing the event data have been studied and results will be presented regarding format, structure and content. For reading and storing the event data, suggestions of interpreters and data models are also provided. Finally a prototype application will be presented, which will provide the defined interpreters, data models and a graphical user interface to represent the event data and event data correlations. The programming was conducted using Java and the application is implemented as an Eclipse Plug-in. With the help of the application the user will get a better overview and a more intuitive way of working with the event data.

Keywords: logs, log file analysis, log file structure, event data, data modelling, event data representation, event date correlation, event data profiling, graphical user interface, CPP

Handledare: Daniel Flemström
Ämnesgranskare: Kjell Orsborn
Examinator: Anders Jansson
ISSN: 1401-5749, UPTEC IT 08 010
Sponsor: Ericsson AB

Tryckt av: Reprocentralen ITC

# Preface

This is a Master Thesis conducted by Tobias Hedlund at the Department of Information Technology of Uppsala University, and by Xingya Zhou at the Information and Communication Technology Department of the Royal Institute of Technology. The thesis work has been carried out at a lab in Mälardalen University for Ericsson AB. Other people involved in this project include:

Alf Larsson, Project Manager for this project at Ericsson
Axel Jantsch, Examiner and Supervisor at the Royal Institute of Technology
Anders Jansson, Examiner at Uppsala University
Kjell Orsborn, Supervisor at Uppsala University
Daniel Flemström, Technical Supervisor at the Ericsson Lab

During this project Xingya Zhou's work effort has been focused on areas concerning OSE and CPP, such as the programming of our test base application and collecting log files; while Tobias Hedlund has been working with areas close to the end user, such as the graphical user interface and visualizations. Work that was conducted collaborative includes the research part for background theory, related work, log file analysis, data model analysis and graphical representation analysis. During the event data analysis and graphical representation analysis parts, we divided the work on the different log file types. Xingya Zhou was responsible for the Execution Address Profiler and TSL log file types, and also TSL related elements in the Trace and Error log file type, while Tobias Hedlund worked with the Kernel Trace and Trace and Error Log files; the analysis parts concerning the CPU Load log file type was conducted by both of us. Even though many of the parts were divided between us the work was still conducted in a highly collaborative manner. Details about each of our contributions can be seen in *Appendix A* at the end of the report.

The terminology and abbreviations used throughout this thesis report is explained in the *Terminology* chapter.

## Acknowledgements

We want to thank all the people that were involved in this project. Special thanks goes to Alf, our project manager at Ericsson AB in Älvsjö, for sharing his valuable time, providing us with necessary tools and pushing the staff at Ericsson to helping out with various necessities of the project. Daniel Flemström should also have special thanks for providing us with contacts and pointing us at the right directions.

People that was not directly involved in the project but still deserves thanks are: Mikael Krekola that assisted Daniel Flemström in helping us with the CPP related issues at the Ericsson lab; Ravi Kumar Akkisetty for implementing the Rose Real Time part of the Test Base Application and for being very helpful if we ever had any questions about Rose Real Time and the TSL log file; and finally Magnus Larsson for giving us an informative seminar in the Rose Real Time background theory for this project.

*Tobias Hedlund, Xingya Zhou*

# Table of Contents


**1 Introduction ..... 1**
1.1 Problem Overview ..... 1
1.2 Purpose and Criteria ..... 1
1.3 Delimitation ..... 1
1.4 Method Description ..... 1
1.5 Contributions ..... 2

**2 Background ..... 3**
2.1 Embedded and Real-Time Systems ..... 3
*2.1.1 OSE Real-Time Operating System ..... 3*
2.2 Event Data and Event Data Correlation ..... 3
*2.2.1 Event Data ..... 3*
*2.2.2 Event Data Correlation ..... 4*
2.3 Connectivity Packet Platform ..... 4
*2.3.1 CPP Software Structure ..... 5*
*2.3.2 CPP Hardware Structure ..... 5*
*2.3.3 CPP Execution Platform ..... 6*
2.4 Tools Used To Collect Logs ..... 6
*2.4.1 Remote Debug Support ..... 6*
*2.4.2 Trace & Error Package ..... 7*
*2.4.3 Profiling ..... 7*
2.5 Rational Tools ..... 8
*2.5.1 Rational Rose Real Time ..... 8*
*2.5.2 Rational ClearCase ..... 8*
2.6 Eclipse and Eclipse Plug-ins ..... 9
*2.6.1 Eclipse Workbench ..... 9*
*2.6.2 Standard Widget Toolkit ..... 9*
*2.6.3 JFace ..... 9*

**3 Related Work ..... 10**
3.1 Existing Tools and Approaches for Correlation of Event Data ..... 10
*3.1.1 RuleCore CEP ..... 10*
*3.1.2 Logsurfer ..... 10*
*3.1.3 SEC ..... 10*
*3.1.4 Event Log Analyzer ..... 10*
*3.1.5 Eclipse TPTP Tracing and Profiling Tools ..... 11*
*3.1.6 Conclusions ..... 11*
3.2 Common Base Event ..... 11

**4 Event Data Analysis ..... 12**
4.1 The Assigned Set of Log Files ..... 12
*4.1.1 The Trace and Error Log ..... 13*
*4.1.2 The Kernel Trace Log ..... 13*
*4.1.3 The CPU Load Log ..... 13*
*4.1.4 The TSL Log ..... 13*
*4.1.5 The Execution Address Profiler Log ..... 14*
4.2 Log Collection via Test Base Application ..... 14
*4.2.1 Test Base Application ..... 14*
*4.2.2 Log Collection ..... 16*
4.3 Correlation Analysis ..... 16
*4.3.1 Common Event Data ..... 17*
*4.3.2 Correlation Possibilities ..... 17*
*4.3.3 Correlation Accuracy ..... 24*
4.4 Graphical Representation ..... 25
*4.4.1 Table Representation ..... 25*
*4.4.2 Representing Log Information Summary ..... 25*
*4.4.3 Time Chart Representation ..... 26*
*4.4.4 Node Graph Representation ..... 26*

*4.4.5 Statistical Representation* ... *27*
*4.4.6 Detailed TSL Specific Representation* ... *27*

**5 Implementation of the Correlation Tool** ... **32**
5.1 Interpreters ... 32
5.2 Data Model ... 32
*5.2.1 Specific vs. Generic Data Model* ... *33*
*5.2.2 The Generic Data Model* ... *33*
*5.2.3 The Specific Data Models* ... *34*
5.3 Graphical User Interface ... 35
*5.3.1 Approach* ... *35*
*5.3.2 User Analysis* ... *36*
*5.3.3 Implemented Functionality* ... *36*
5.4 Software Architecture ... 43
*5.4.1 Views* ... *43*
*5.4.2 Actions* ... *43*
*5.4.3 Dialogs* ... *44*
*5.4.4 Editors* ... *44*
*5.4.5 Engine* ... *44*
*5.4.6 Parsers* ... *44*

**6 Comparison with the TPTP Tracing and Profiling Project** ... **45**
6.1 Using the TPTP Tracing and Profiling Project ... 45
6.2 Advantages using the TPTP Tracing and Profiling Project ... 45
6.3 Advantages using the Correlation Tool ... 45

**7 Future Work** ... **46**
7.1 Correlation Analysis ... 46
7.2 Implementation ... 46
*7.2.1 Extension of the Correlation Tool* ... *46*
7.3 Log File Evaluation and Suggestions ... 48
*7.3.1 Log File Structures* ... *48*
*7.3.2 Log File Formats* ... *49*
*7.3.3 Log File Content* ... *50*

**8 Concluding Discussion** ... **52**
8.1 Summary of the Results ... 52
8.2 Complications ... 52
8.3 Alternative solutions ... 53
*8.3.1 Pre-processing of Event Data* ... *53*
*8.3.2 Data Model vs. Data Base* ... *53*

**9 References** ... **54**

**10 Terminology** ... **56**
10.1 Abbreviations ... 56
10.2 Definitions ... 56

**Appendix A: Individual Thesis Contributions** ... **58**

**Appendix B: Data Models** ... **59**

**Appendix C: The Correlation Tool** ... **62**

**Appendix D: Log Files** ... **66**

## Index of Figures


Figure 1: CPP System Areas 5
Figure 2: Connect to a CPP node 5
Figure 3: Processor hierarchy in a CPP node 6
Figure 4: Trace with RDS and Host environment 7
Figure 5: The process that logs an event using Trace Support 7
Figure 6: Relation between log files and system 12
Figure 7: Test Base Application design, OSE to OSE communication 15
Figure 8: Test Base Application design, OSE to UML communication 16
Figure 9: Common event data between the log files 17
Figure 10: Common event data with the TSL log file 17
Figure 11: Load Module interactions 19
Figure 12: Process / Controller interactions 19
Figure 13: Actor interactions 19
Figure 14: Capsule states and transitions 20
Figure 15: TSL Event Type 4 example output 21
Figure 16: Trace and Error TRACE5 example output 21
Figure 17: TSL Log Event Type 1 example printout 22
Figure 18: T&E Log STATE CHANGE and REC SIG example printout 22
Figure 19: TSL Event Type 1 (transiton cost), example output 24
Figure 20: EAP log, transition example output 24
Figure 21: Table representation of Kernel Trace entries 25
Figure 22: Table representation of Kernel Trace processes 25
Figure 23: Time chart view representing events in relation to time 26
Figure 24: Node graph representing Kernel Trace signal propagation 27
Figure 25: Two different TSL log browser view structures 28
Figure 26: Example table view of Event Type 1 and 3 29
Figure 27: Example table view of Event Type 5, 6 and 7 29
Figure 28: Example table view of Event Type 10 and 11 29
Figure 29: Example table view of Event Type 4 29
Figure 30: TSL node graph of signal propagation tree 29
Figure 31: TSL event data represented on a time line with suggested filters 30
Figure 32: Event data correlation via time lines 31
Figure 33: Extending the application with a new log file type 33
Figure 34: The generic data model 34
Figure 35: The TSL specific data model 35
Figure 36: The Navigation View 37
Figure 37: The Table Editor showing the Kernel Trace event data entries 38
Figure 38: The Table Editor showing the Kernel Trace event data processes 38
Figure 39: The Time Chart Editor together with the Table Editor 39
Figure 40: The Time Chart Editor showing a line chart diagram 40
Figure 41: The Node Editor 41
Figure 42: Interactions between event data representations in the GUI 41
Figure 43: Various GUI functionalities 42
Figure 44: Software architecture of the Correlation Tool 43
Figure 45: Kernel Trace, Trace and Error and CPU Load data models 59
Figure 46: The Common Base Event class hierarchy 61
Figure 47: YAML log structure 76
Figure 48: Log file structure showing TSL Block 2, Event Type 1 76
Figure 49: Text log file structure showing TSL Block 2, Event Type 1 76

## Index of Tables

Table 1: Trace and Error trace groups ..... 13
Table 2: TSL Block 1 content ..... 14
Table 3: TSL Block 2 content ..... 14
Table 4: Log file criteria ..... 48
Table 5: Log file formats, advantages and disadvantages ..... 49
Table 6: Individual thesis contributions ..... 58
Table 7: Decision points, specific vs. generic data model ..... 60
Table 8: Data that can be extracted from the KTR, T&E, CPU Load and EAP logs ..... 71
Table 9: Data that can be extracted from the TSL Log ..... 71
Table 10: Common event data between the TSL and other log files ..... 72

# 1 Introduction

## 1.1 Problem Overview

Ericsson AB frequently makes use of the Connectivity Packet Platform (CPP) System. This is an event driven complex real time system for packet switching where event data from different parts of the system is recorded into logs. These logs are often designed for individual purposes and they record only the required system information which is a small part of the overall system. To examine a wider scope of system information, log files with event data from different parts of the system could be studied. By also correlating the event data further information could be gained. Today correlation possibilities have not yet been analyzed and it is not apparent what further information could be gained by doing this. Further, the log files are studied in a static manner through command line printouts which can be a tedious job when analyzing and comparing the event data. They are also not consistent which implies that relationships between captured event data can be hard to find.

## 1.2 Purpose and Criteria

The purpose of this project is to correlate event data from different parts of the system and represent these in a consistent context with the help of a graphical user interface. Correlations between the event data would facilitate in finding source of errors or certain behaviors within the system; while the graphical user interface facilitates in getting a better overview of the system and a more intuitive way of working with the event data. The criterion's given by Ericsson AB are that data model and interpreters are defined for the log files, and that the interface implementation is to be conducted as an Eclipse Plug-in.

## 1.3 Delimitation

We are aware of the wide spectrum of this thesis and instead of doing any deep studying in any specific parts we have focused on the entire process to give an overview and a proposal that Ericsson can use for a future implementation. The project will focus on the correlation analysis and graphical representation as the main results, while the graphical user interface will be used as proof of concept to earlier results and as a prototype application to demonstrate the user interactions with the event data and correlations.

This project will focus on retrospective event analysis only and analysis of system monitoring will not be considered. There is also no input of system specific processes or behaviours to this project, why the correlation analysis is conducted on an observer level. This means that event data attributes and relations will be analysed, but not the system specific behaviour or rules. The correlation analysis will be conducted on event data from the same board on the CPP node. Correlation of event data from different boards will follow the same concept in many cases but we will not do any analysis in this area.

Due to time restrictions of the thesis an infrastructure for storing configurations and results will not be provided.

## 1.4 Method Description

In the beginning of the project a crash course was provided, where we got insight in how the CPP system is built up and how the OSE system works. When we had general knowledge of the system the proceeding method was mainly a bottom-up approach[a]. We started from the low layer of analyzing the logs and what data that can be extracted and correlated. When the

[a] Bottom-up parsing is a strategy for analyzing unknown data relationships that attempts to identify the most fundamental units first, and then to infer higher-order structures from them

event data was analyzed and extracted the data model was created for storing this information. Further analysis was conducted on how event data is best represented and how the data model should be structured and implemented. To be able to put the event data into the data model the logs first need to be interpreted and parsed. Different ways of doing this was considered and finally implemented. After these sections analysis of how the event data and correlations can be represented graphically was conducted followed by analysis and implementation of the application and graphical user interface. In latter parts of this project this implementation will be referred to as the Correlation Tool.

Concurrent with earlier mentioned steps a Test Base Application was created to better understand the relations between the log files and to be used as analysis ground when representing the information in the Correlation Tool.

For background and research part of the thesis some of the information was gathered from meetings at Ericsson AB, internal documentation, or from research articles and books. We have used sources that we consider to be reliable throughout the project and we have not made use of internet resources other than for concept definitions and product specific information.

## 1.5 Contributions

The main contributions of this thesis are the following:

- Analysis results of how the defined set of logs can be profiled and correlated
- Analysis results of how the event data can be represented graphically
- Suggestion of data model for the event data that is analyzed during this thesis
- A prototype plug-in for correlation of event data that demonstrates different ways of representing the event data correlation.
- Suggestions of improvement for the log file formats

# 2 Background

## 2.1 Embedded and Real-Time Systems

An embedded system is a special purpose computer that is built into a larger device [1]. Embedded systems often reside in machines that are expected to run continuously for years without errors and in some cases recover themselves if an error occurs. Many embedded systems also have real-time constraints that must be met, for reasons such as timing requirements, safety, usability and fault-tolerance [2].

A real-time system should be able to manage time-dependent applications. The validity of a real time embedded system is affected by two main issues: one is the results that it produces and the other is the time when the results are generated.

Today's real-time embedded systems are becoming more and more complex, and more requirements are met such as scalability across multiple CPUs, multiple communicating in multi-core and distributed environments. A real-time operating system, which must be able to schedule tasks at or after certain specified time, is designed for supporting those complex real-time applications.

A real-time operating system (RTOS) has three types of main duties: resource management, time management, and inter process communication [1]. There are many RTOS available at the market that currently support complex real-time applications, such as OSE RTOS, Symbian OS, ThreadX (Express Logic's advanced RTOS) [3], LynxOS RTOS [4], etc.

In this project, OSE RTOS is used as the working environment. More information about this operating system follows in the next section.

### 2.1.1 OSE Real-Time Operating System

OSE is a powerful platform produced by Enea for the design of real-time embedded systems. It is deployed in approximately half of the worlds 3G mobile phones and base stations [5]. The main features of OSE platform are reliability, scalability and simplicity of direct message passing system [5].

Enea has been making great efforts to support reliable system deployment and maintenance. Currently, OSE provides several ways to ensure reliability such as multi-level facilities for error detection, built-in monitoring of critical tasks, fault tolerant system, etc [5].

Message communications between different processes plays a significant role in real-time embedded system designs. OSE uses a memory pool to allocate for message buffers and has a direct message-passing model which provides fast, asynchronous inter process communication. As a result, many program errors that may occur during inter process communications could be avoided.

## 2.2 Event Data and Event Data Correlation

### 2.2.1 Event Data

The word event is being used more and more frequently in many different areas of computer systems. In a very general sense, an event means an action or occurrence that could be detected by a program. For example, events can be user actions, such as a mouse clicking in a graphical user interface, a key pressing to the keyboard or system occurrence such as hardware of software failures.

Events play a significant role in system designs and implementations. Many large systems are designed according to event-based architectures. Some complex applications such as real-time embedded systems or distributed systems are designed to respond to events, these systems are called event-driven systems [6]. In this kind of system, an event could be some message, specified signal, token, value or marker that can be identified by the ongoing processes.

Events are very useful when monitoring complex systems, such as telecommunication networks, air traffic and stock markets [7]. Some events are instantaneous, most occur over an interval of time [8]. In order to better understand the behavior of a system, much effort has been made to monitor and trace these events. When a system encounters an event, it could emit an event data (or event message) that describes the event [9]. These event data are stored to a local or remote event log. For example, when a disk of a server becomes full, the server could generate a timestamped "Disk full" message for appending to a local log file or for sending over the network as an SNMP[a] trap. In most cases event data are appended to event logs in real-time, so event logs are an excellent source of information for monitoring the system [9].

Event data are also essential to better understand the activities of complex systems and to analyze problems, particularly in the case of applications with little user interaction (such as server or network applications). However, in most cases, the logs are esoteric or too verbose and therefore hard to understand; they need to be subjected to log analysis and correlation in order to make sense of them.

### 2.2.2 Event Data Correlation

As a baseline, correlation is defined as the drawing of a causal, complementary, parallel or reciprocal relationship between different events based on specific criteria [10]. Generally, correlations could be interpreted as establishing or finding relationships between different entities. When analyzing and monitoring a complex system such as telecommunication networks, air traffic or stock markets, designers always generate some logs to help them get a better understanding of the system behaviours. Designers working in different areas of a system usually design and examine their own logs. These logs are sometimes not well organized, information might overlap and complement might exist among them. Thus the event data from different sources need to be correlated so that useless information can be filtered out, important information can be cross-referenced into a consistent context, and new information could be generated.

## 2.3 Connectivity Packet Platform

Ericsson Connectivity Packet Platform (CPP) [11] is a platform that is designed for accessing and transporting user traffics in mobile and fixed networks. It was first designed for Asynchronous Transfer Mode (ATM) and Time Division Multiplex (TDM) transport, but since then more and more support has been added such as multimedia services for the third generation of mobile telephony and Internet Protocol (IP) transport. The CPP platform is also very flexible and can be configured with different types of circuit boards according to different design requirements. Based on this platform, it is possible to develop different high availability applications such as ATM and IP based nodes, Radio Base Station (RBS), Media Gateway (MGW), and Radio Network Controller (RNC) of the Universal Mobile Telephony System (UMTS) network.

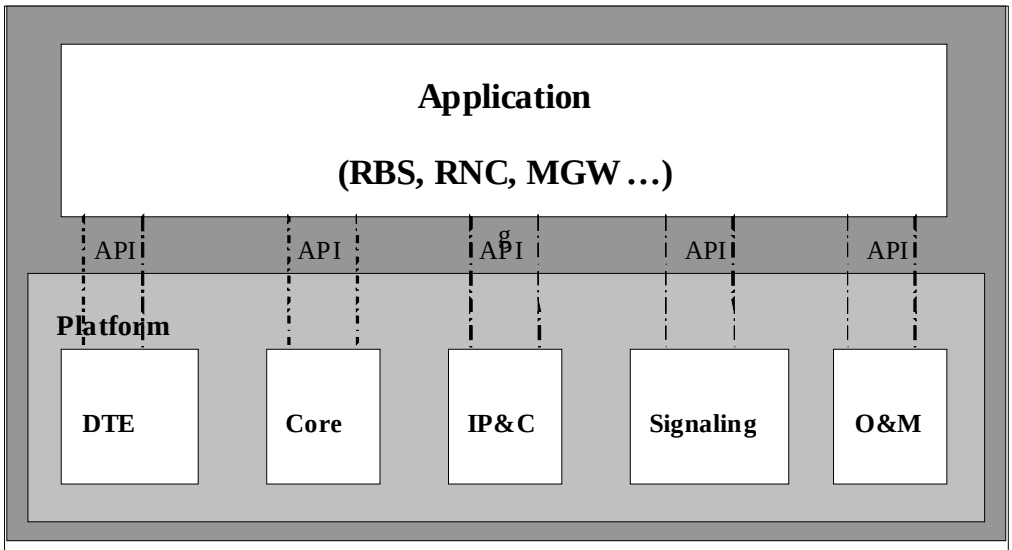
A CPP node contains two parts, an application part and a platform part [11]. The application part handles the software and hardware that is application specific. The platform part handles

[a] Simple Network Management Protocol

common functions such as internal communication, supervision, synchronization and processor structure. This project has a focus on the platform.

### 2.3.1 CPP Software Structure

The CPP platform consists of five system areas as shown in the figure below. The areas are: Development and Trouble shooting Environment (DTE), the Core, Internet Protocol and Connectivity (IP&C), Signalling and Operation and Maintenance (O&M).




API – Application Programming Interfaces
DTE – Development and Troubleshooting Environment
IP&C – Internet Protocol and Connectivity
O&M – Operation and Maintenance

***Figure 1: CPP System Areas***

DTE is software development environment for both application software and CPP software. The tools can be used for debugging and building load modules. A load module consists of software that that can be executed a board.

### 2.3.2 CPP Hardware Structure

A CPP node can vary from the smallest node consisting of a single subrack to a large node consisting of several subracks. A subrack which consists of 28 slots is the smallest building unit. It can be physically configured and updated with different types of circuit boards such as general purpose board, switch core board, media stream processor board and special purpose processor board. The CPP node that was used in this project consists of one subrack and 11 boards. There are two ways to connect to it, either via terminal server or via TCP/IP. The boards and connection possibilities are shown in Figure 2.

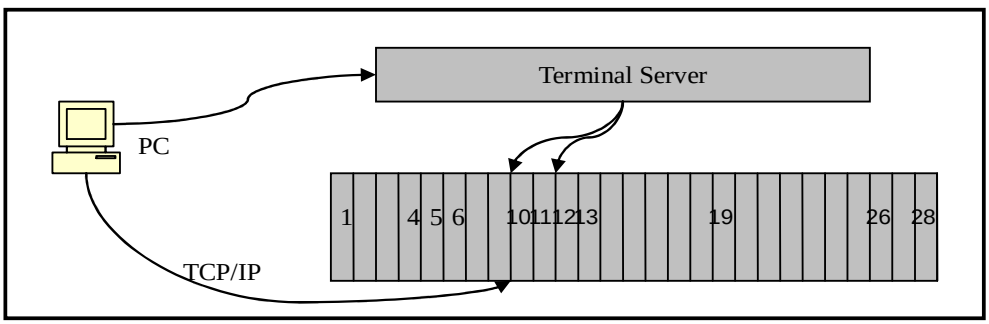



***Figure 2: Connect to a CPP node***

### 2.3.3 CPP Execution Platform

The CPP system execution platform consists of the hardware and system software that applications need to execute correctly in a multi processor system. CPP offers applications an execution platform comprised of the following:

- A number of processors and communication between them
- A distributed real-time OS, supporting robust application design
- Operation and Maintenance (O&M) support for applications
- A fault-tolerant real-time database
- A robust fault-tolerant file system
- Java Virtual Machine (for management applications)
- A space switch

Different types of boards contain different processors. The processors in the execution platform have a hierarchical order as shown in Figure 3. The processors in the Main Processor Cluster have the highest rank. These processors are referred to as Main Processors (MPs) and are interconnected in a full mesh. The MPC is the center in a star topology with Board Processors (BPs) at the end of the rays. The execution platform can be extended beyond the BP domain by connecting one or more subordinate Special purpose Processor (SP) or Media Stream Processor (MSP) to BPs [11].

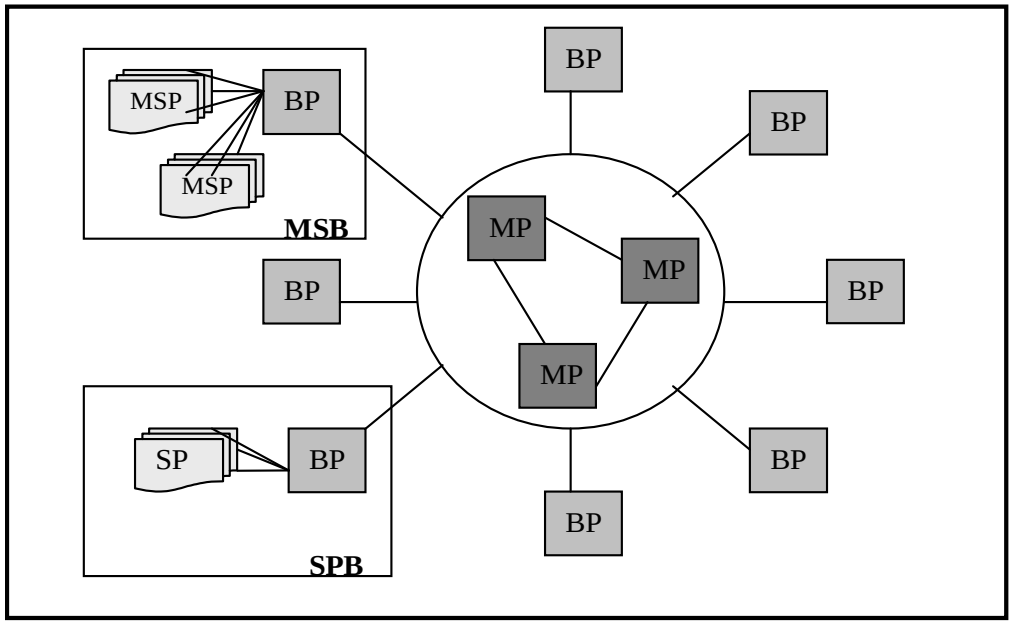



***Figure 3: Processor hierarchy in a CPP node***

## 2.4 Tools Used To Collect Logs

### 2.4.1 Remote Debug Support

Remote Debug Support (RDS) is a system level debugger for the CPP node. It is used for tracing OSE specific events such as signals between processes and the creation and killing of OSE processes. The command interface of the debugger is the OSE shell and the users should enable the trace actions themselves from the command line interface [12]. The tracing results will be stored in a log file called the Kernel Trace log which is kept in the OS kernel trace buffer. Signal target connection is using the default from Autodds. Figure 4 below shows the procedure of signal tracing with RDS and host environment.

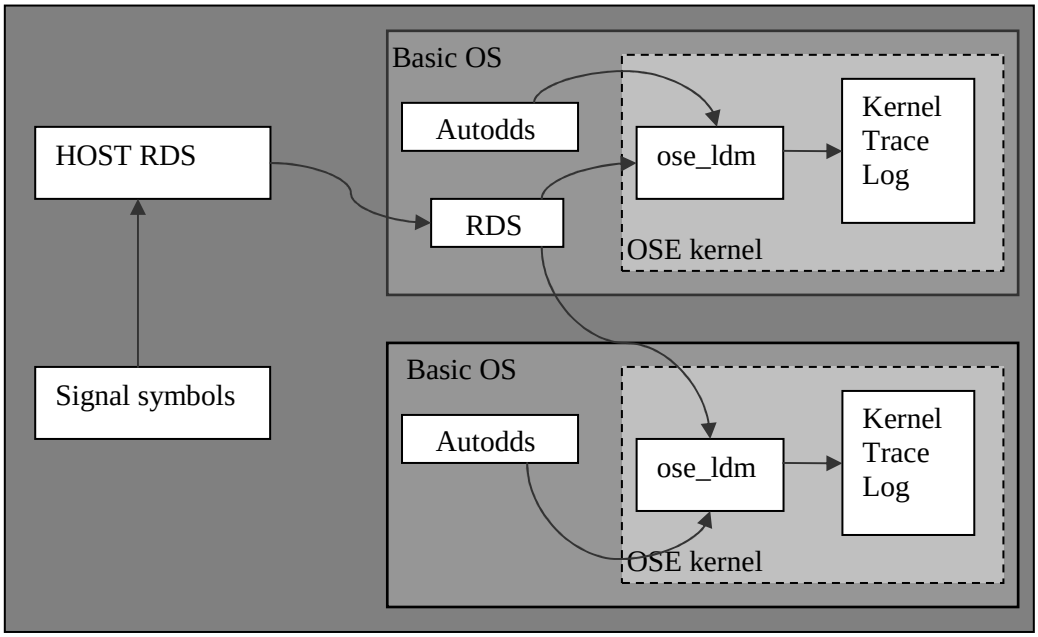



***Figure 4: Trace with RDS and Host environment***

### 2.4.2 Trace & Error Package

All software development meets kinds of errors or faults during implementation. Trace & Error package is designed to detect and handle errors of programs running on a CPP node [13]. This tool could be used as a complement to other debug support tools. There are two functionalities supported in the T&E package: *the tracing functionality* and the *error handling functionality*. In this project, only the tracing functionality is used. With the help of this functionality the system and functional behaviours can be traced and reported at software development. As shown in Figure 5 below, events to trace are found within the process code and traced by means of "Trace Macros". A macro is responsible for logging of one event. Most macros allow a message to be added to the log and the message is given as text string.

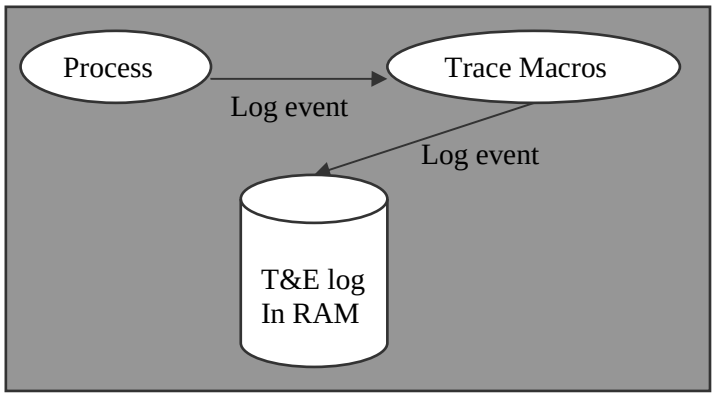



***Figure 5: The process that logs an event using Trace Support***

### 2.4.3 Profiling

Profiling is a system function used to find out bottlenecks in software programs running on a CPP node [14]. There are two areas of profiling functionality: *Sample Based Execution Address Profiler* and *Capsule based Profiler*. Both of them have the same purpose as to find out bottlenecks on a CPP node, but the scope between them is a bit different. In the following sections, an overview of the two different profiling tools is presented.

#### *2.4.3.1 Capsule Based Profiler*

Capsule based profiler (CBP) is a profiling tool for Unified Modeling Language (UML) applications running in a CPP node. It could be reached both from the command line interface and from the application UML model. CBP currently collects ten different event types such as total OSE signal dispatch count, transition cost and message latency. The entry keys of these events are based on either capsule classes or actor instances. An event type is a measurement type which is triggered by a specific UML event such as message received, message sent or state change [14]. This tool generates the Target Service Library (TSL) log file that is one of the log files which was used in this project.

#### *2.4.3.2 Execution Address Profiler*

The Execution Address Profiler is a sample-based tool that is used to measures the CPU usage of different user defined memory areas or different C/C++ functions. This is done by interrupting the CPU with a fixed periodic sample frequency and taking samples of the current execution address [15]. This tool will extract memory regions for C/C++ functions with the help of a configuration generator called *execprofpp* and perform execution address profiling on the CPP node with the help of the *execprof* profiling tool.

## 2.5 Rational Tools

There are two different Rational tools that are important to know about for this project. One is Rational Rose Real Time (RoseRT) and the other is Rational ClearCase. These are developed by the Rational Software division of IBM and they will be briefly described in the following sections.

### 2.5.1 Rational Rose Real Time

Rational Rose Real Time (RoseRT) is software development environment which is designed to meet the demands of real time software. Developers could use RoseRT to create models of the software system based on the UML constructors. It is necessary to introduce some main concepts here which will be mentioned frequently throughout the report.

The *Capsule* is one of the most important concepts of RoseRT. It provides coordinate behaviour in the system and encapsulates the flow of events. Capsules also give transparent concurrency, easy thread assignment, state diagram generation and message passing. It could communicate with other capsules via ports and protocols. A state transition is executed by a capsule when a specified trigger signal arrives. Each capsule can have hundreds of states and transitions.

Another important concept in RoseRT is the *actor*. These are the instances of capsule classes when the program is running on the target. One capsule can have multiple actors executing at the same time.

A third concept that is important to know about is the *threads* (also called *controller*). All capsules should be incarnated on logical threads that are mapped to a physical thread of the memory area in the target.

### 2.5.2 Rational ClearCase

Rational ClearCase is a software tool for revision control (configuration management etc) of source code and other software development assets. ClearCase forms the base of version control for many large and medium sized businesses and can handle projects with hundreds or thousands of developers [16].

ClearCase can run on a number of platforms including Linux, Solaris and Windows. It can handle large binary files, large numbers of files, and large repository sizes. It handles

branching, labeling, and versioning of directories [16]. ClearCase has some unique features such as VOB (Versioned Object Base), Configuration Record, Build Avoidance, Unix/Windows Interoperability, and Integration with other products.

## 2.6 Eclipse and Eclipse Plug-ins

Eclipse [17] is an open-source development framework that provides a common user interface and workbench model for working with tools. The platform is built in layers of plug-ins, each one defining extensions to the extension points of lower-level plug-ins [18]. This extension model allows plug-ins to be developed with a variety of functions to the basic tooling platform and provides a nice integration with already defined tools. By working on an already defined platform developers can focus on the specific task instead of worrying about integration issues such as different runtime environments.

To further understand the required implementation of an Eclipse plug-in based user interface, a short overview of the Eclipse workbench, the Standard Widget Toolkit (SWT) and the JFace toolkit is presented below.

### 2.6.1 Eclipse Workbench

From a high level perspective the workbench is a window through which all of the visual elements of an application are organized. This is the same window that is used for the Eclipse development environment. The visual parts fall into the two major categories *views* and *editors*. Views allows the user to navigate, view, or provide further information about objects that the user is working with in the workbench, while Editors allows the user to browse a document or input-object. Editors also allow the user to edit and save objects, while the views can save their states for the next time they are opened [18].

From a lower level perspective the workbench is a framework that is supplying additional toolkits for building the user interface. This framework also defines extension points for plug-ins to contribute user interface function to the platform. Many of these extension points are implemented using the Standard Widget Toolkit (SWT) and the JFace framework [18].

### 2.6.2 Standard Widget Toolkit

The Standard Widget Toolkit (SWT), is a set of Java class libraries created to provide platform native user interfaces, and this is the graphical tool kit used for Eclipse graphics. The toolkit immediately reflects changes in the underlying Operating System GUI look and feel while maintaining a consistent programming model on all platforms. It substitutes the Java AWT[a] and the Swing toolkit when implementing widgets, layouts and events [18].

### 2.6.3 JFace

JFace is a user interface (UI) toolkit that provides helper classes for developing UI features that can be tedious to implement. It is designed to provide common application UI functions on top of the SWT library and provides an API[b] to build MVC[c]-based user interfaces with the help of components referred to as viewers. Basic functionalities include populating, sorting, filtering and updating widgets. JFace helps the developer to focus on the implementation of the specific plug-in function, rather than focusing on the underlying widget system or solving commonly occurring UI application problems [18].

[a] Abstract Window Toolkit
[b] Application Programming Interface
[c] Model-View-Controller

# 3 Related Work

## 3.1 Existing Tools and Approaches for Correlation of Event Data

The event logs play important roles both in analysis in real time and analysis at a later stage. There is lots of research taking place in both of these two areas with both commercial and open source projects in development. This thesis will only focus on retrospective logs analysis and correlation, but it is still related to the real time, which is why both kinds are researched and mentioned in this section.

For the real time analysis the event logs are excellent for monitoring systems, since the event messages in the logs usually are recorded in real time. For retrospective analysis of the logs collected from a running system, it is extremely helpful to better understand and analysis the behaviour of the system. As a result, the designers could find out the weakness of a system and make decisions on how to improve it.

There are some tools already available for event logs correlation and monitoring in the market. Some of them are open sources and some of them are commercial products that can be very expensive. In the following part, a few interesting tools are introduced. These are Rule Core Complex Event Processing (CEP) Server [19], Logsurfer [20], Simple Event Correlator (SEC) [21], ManageEngine Event Log Analyzer [22], and TPTP Trace and Profiling Tools [23].

### 3.1.1 RuleCore CEP

The ruleCore CEP Server is the solution to the problem of how to know when a critical situation has happened so that users can start a process to manage it. This is done by providing real-time behaviour tracking and tracing of any events that are critical to the system. It uses the Simple Rule-based Event Correlation approach for performance management. Rule-base Event Correlation means to specify some rules such as if-else statements for event data monitoring and correlation. For example, in ruleCore CEP Server, some simple rules are specified like accepting input events that include only name-value pairs and taking events from a specified place.

### 3.1.2 Logsurfer

Logsurfer is one of the most useful tools for monitoring system logs in real time and reporting on the occurrence of events. It also uses rule-based approach as the ruleCore CEP Server does. Its rules simply provide instructions on what to do when it detects a particular line in the incoming stream of log messages.

### 3.1.3 SEC

SEC is an open source platform independent tool for rule-based event correlation. It was created to be a lightweight tool that can be used for a wide variety of event correlation tasks. The SEC configuration is stored in text files as rules, each rule specifying an event matching condition, an action list, and optionally a Boolean expression whose truth value decides whether the rule can be applied at a given moment. SEC has been successfully applied in various domains like network management, system monitoring, data security, intrusion detection, log file monitoring and analysis, etc.

### 3.1.4 Event Log Analyzer

Event Log Analyzer is a web based, real time, event log and application log monitoring and management tool. It collects, analyzes, reports, and archives Event Logs from different places such distributed windows hosts, syslog from devices and application logs from web

servers and so on. It helps system administrators to troubleshoot performance problems on hosts, select applications, and the network.

### 3.1.5 Eclipse TPTP Tracing and Profiling Tools

The TPTP[a] Tracing and Profiling Project is in contrast with the previous mentioned tools aimed for retrospective analysis of log files. It is a sub project for the Eclipse TPTP Project, and it addresses the tracing and profiling phase of the application lifecycle [23]. It also provides a framework for analyzing and correlation log files, has extension points from where log parsers can be created, and already defined views for analyzing and correlating event data. For representing the event data it makes use of the Common Base Event standard that is explained further in the *Common Base Event* section.

### 3.1.6 Conclusions

Most of the available tools are developed for real time monitoring such as Rule Core Complex Event Processing (CEP) Server [19], Logsurfer [20], Simple Event Correlator (SEC) [21] and ManageEngine EventLog Analyzer [22] which are described above. All of them consider a rule based approach for event data correlation in real time and are for this reason less interesting for the context of this project. In addition, ManageEngine Event Log Analyzer is a commercial product, it costs money; and some special tools like HP openView were designed for one particular network management platform only. Among these existing tools, TPTP is the most interesting one. This framework was however found at the end of the thesis project and because of this it was never used in the implementation. Instead it will be compared with our own tools in the *Comparison with the TPTP Tracing and Profiling Project* chapter and it will also be mentioned in the *Future Work* section.

## 3.2 Common Base Event

The Common Base Event allows the use of a common format for any log records from any supported proprietary log files [18]. The proposal comes from IBM and the goal is to standardize the format of events to assist in designing robust, manageable and deterministic systems [24].

Entries stored in the Common Base Event are defined by properties that are collectively referred to as the 3-tuple, consisting of the following elements:

1. Id of component that reports the situation
2. Id of component that is affected by the situation (which may be the same as the component that is reporting the situation)
3. The situation itself

The data collected for the above 3-tuple are properties such as the reporter component, situation, creation time, severity, property, message, extended data element, and sequence number. For more complex logs the extended data element is used for including product-specific attributes which allows user-supplied extensions for any attributes not defined in the Common Base Event. The class hierarchy diagram with further structural details can be seen in appendix section *B.3*

Using the Common Base Event doesn't mean that the application generating the log files needs to be re-written, instead parsers can be used to translate it into the new standard when accessed. There are already defined tools in the TPTP framework (see *Eclipse TPTP Tracing and Profiling Tools,* section 3.1.5) to facilitate in such a translation.

[a] Test and Performance Tools Platform

# 4 Event Data Analysis

This chapter will present the event data analysis part of the project. The sections included in this chapter will present the log files that contain the event data; the Test Base Application that was developed to generate the different log files; correlation analysis of the event data; and finally graphical representation for event data and correlations.

## 4.1 The Assigned Set of Log Files

The logs that are considered during this project are the following:

- Target Service Library (TSL) log
- Execution Address Profiler (EAP) log
- Trace and Error (T&E) log
- Kernel Trace (KTR) log
- CPU Load log

These log files represent information in different parts of the CPP hardware and software layers. The figure below shows a simplified diagram of the different system layers in relation to the analyzed log files.

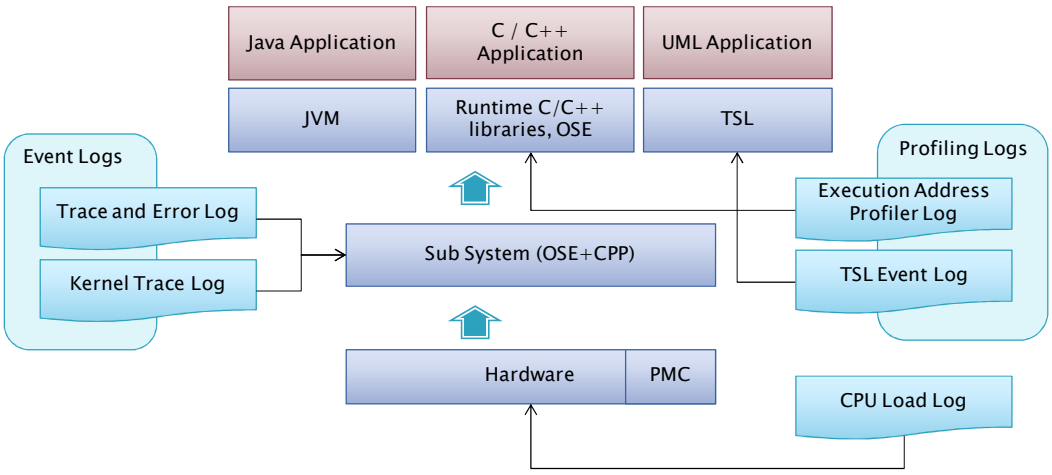



***Figure 6: Relation between log files and system***

The log files that are grouped as *Event Logs* are recording events taking place in the sub system (OSE + CPP), while the log files grouped as *Profiling Logs* are tracing and recording information from higher level applications. The CPU Load Log is a global log, showing the overall status of the system.

The following sections will give a short overview of the different assigned log files. Example outputs from the same log files are shown in appendix section *D.1*, and a table of data that can be extracted from these log files can be found in appendix section *D.2*.

The log files have also been analyzed considering the structure and syntax from a parsing and correlation perspective, but since this part is not directly related to the focus of the project, this analysis with results will instead be mentioned in the *Future Work* section of this thesis.

### 4.1.1 The Trace and Error Log

The Trace and Error (T&E) log shows a history of recorded trace and error events on the system. The events are recoded with the use of macros, and it is frequently used by designers for troubleshooting. The user can decide to print information in messages that belongs to predefined categories referred to as *trace groups*. Depending on the trace group the message can contain either a simple user-defined string or a formatted string that provides further attribute information. The group and message will be recorded together with a time stamp, load module and source component of the event. Each of the trace groups can individually be switched on or off. Some of the groups that are important to know about later in the report are explained further in the table below.

| Trace Group | Group information |
|---|---|
| STATE CHANGE | Used to print the state change information of capsules in a RoseRT application |
| SEND SIG | Contains information of the signals sent to a RoseRT capsule |
| REC SIG | Contains information of the signals received by a RoseRT capsule |
| TRACE5 | Part of the TSL profiling and also called Event Type 9. This group contains actor specific information |
| TRACE7 | Trace events related to OSE signals for RoseRT applications. |

***Table 1: Trace and Error trace groups***

### 4.1.2 The Kernel Trace Log

The Kernel Trace (KTR) Log records process specific OSE events that occur on the node. This includes events such as sent signals, received signals, created processes, killed processes and error events. The extended version of this log file also contains the binary that is sent with a signal.

### 4.1.3 The CPU Load Log

CPU Load log stores the CPU utilization for different measuring objects such as process type, process name or priority. There are 4 types of CPU-load logs available according to different measuring objects or measuring ways. The most commonly used one is the CPU peak load log which stores the information of the top hundred CPU-load measurements. It is measured by the system itself once the system starts running. CPU utilization could also be measured according to user specified measuring objects. This could be done from the command line.

### 4.1.4 The TSL Log

The TSL log collects ten different UML based event types currently. There are two blocks in TSL log: *Block 1* and *Block 2*. Block 1 contains Event Type 4, 10 and 11 for all started controllers. Block 2 contains Event Type 1, 3, 5, 6, 7, 8, 10 and 11 for each started controller. To be noticed, Event Type 9 is recorded in T&E log so it is introduced in *The Trace and Error Log* section. The TSL Log content in *Table 2* and *Table 3* gives an overview of the TSL log structure. Some example output of different event types is shown in *appendix D.1.4*.

| Event Type | Details |
|---|---|
| Event Type 10 | Internal queue peak size for different priorities<br>External queue peak size for different priorities<br>Defer queue peak size |
| Event Type 11 | Total number of received OSE signals<br>Total number of received UML process external messages<br>Total number of received UML process internal messages |
| Event Type 4 | Signal propagation tree |

***Table 2: TSL Block 1 content***

| Event Type | Details |
|---|---|
| Event Type 1 | Transition Cost |
| Event Type 3 | UML Message Latency |
| Event Type 5 | UML Message Receive Counter |
| Event Type 6 | UML Send Counter<br>UML Invoke Counter |
| Event Type 7 | State Change Counter |
| Event Type 8 | RTMutex contention count |
| Event Type 10 | Internal queue peak size for different priorities<br>External queue peak size for different priorities<br>Defer queue peak size |
| Event Type 11 | Message Receive Counter |

***Table 3: TSL Block 2 content***

### 4.1.5 The Execution Address Profiler Log

The Execution Address Profiler (EAP) log collects the CPU utilization per predefined memory area or per C/C++ function. There are two types of execution address profiling logs. One is generated by execution address profiling configuration tool which uses an .elf that is generated when building a product as input and generates a .reg file as output. The other is generated from the tool *execprof* (see *Tools Used To Collect Logs* section) which uses the .reg file as an input when the measured object is running on the target. Example log file output can be seen in appendix *section D.1*.5.

## 4.2 Log Collection via Test Base Application

Before analyzing the event data correlation possibilities the log files first need to be generated to contain information of the same scenario. This was done by developing and using two test base applications that will be described throughout this section.

### 4.2.1 Test Base Application

The applications were designed based on studies of OSE, Rational RoseRT and a simple pingpong application which has two simple processes that communicates with each other. Documents that were studied include internal documentations of Ericsson such as *Design Rules for Trace and Error Users* [13], *Execution Address Profiler User Guide* [15], *Users Guide for RoseRT Target Service Libraries* [25].

The two test base applications that we used were developed using OSE to OSE communication design and OSE to UML communication design respectively. The OSE parts of these test base applications were developed with the help of OSE programming, while the UML part is programmed with RoseRT.

In both test base applications, T&E handlings were built into the program in order to do a logging of the type of error or interesting events related to signals like where and when it was detected. This was done by adding T&E macros such as SEND SIG and REC SIG to each process. When the application is running on the target the processes will keep track of which trace group that is currently active.

#### *4.2.1.1 OSE to OSE Communication Design*

There are two load modules (LM) used in this design: one is called master and the other is called slave. They communicate with each other via sending and receiving OSE signals. The diagram showing the communication can be seen in *Figure 7*. In the LM master, there is one OSE priority process which is called *master_request*. In the LM slave, there are three OSE priority processes called *slave_wait*, *read_valid* and *read_ready*.

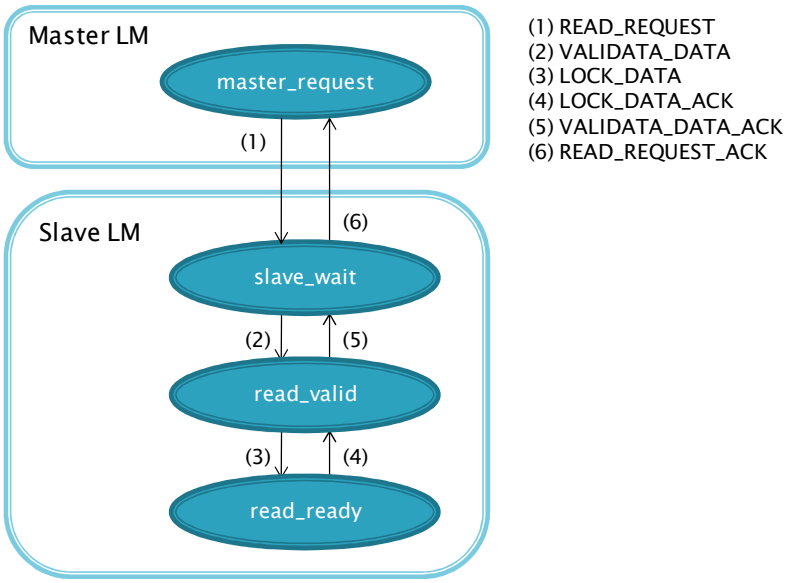



***Figure 7: Test Base Application design, OSE to OSE communication***

These processes communicate with each other via OSE signals. The picture above shows the structure of the OSE to OSE Communication applications and the signal dependencies between the OSE processes. This simple application simulates the communication handshake between a real master and a real slave. Before *master* reads the data from *slave*, it should first send a request to the slave and then wait for the acknowledge signal from slave. Other signal communication takes place inside the slave LM. Only the handshake for read request was simulated. *Master* sends *READ_REQUEST* to *slave* once per second. The order of the signals should follow the order of the signal number representations from (1) to (6) in the picture in *Figure 7*.

#### *4.2.1.2 OSE to UML Communication Design*

The OSE to UML communication design also consists of two LM: one is master and the other is slave. They communicate with each other via sending and receiving OSE signals. In the LM master, there is one OSE priority process called *master_request*. In the LM *slave*, there are five threads called *thread1*, *thread2*, *thread3*, *thread5*, *time* and *main_thread*. These LM are presented in *Figure 8*, but only *thread1, 2, and 3* are shown here since these threads have the similar function as *slave_wait*, *read_valid*, *read_ready* in the OSE slave (described in the previous section). Each of the treads shown in *Figure 8* contains one capsule class.

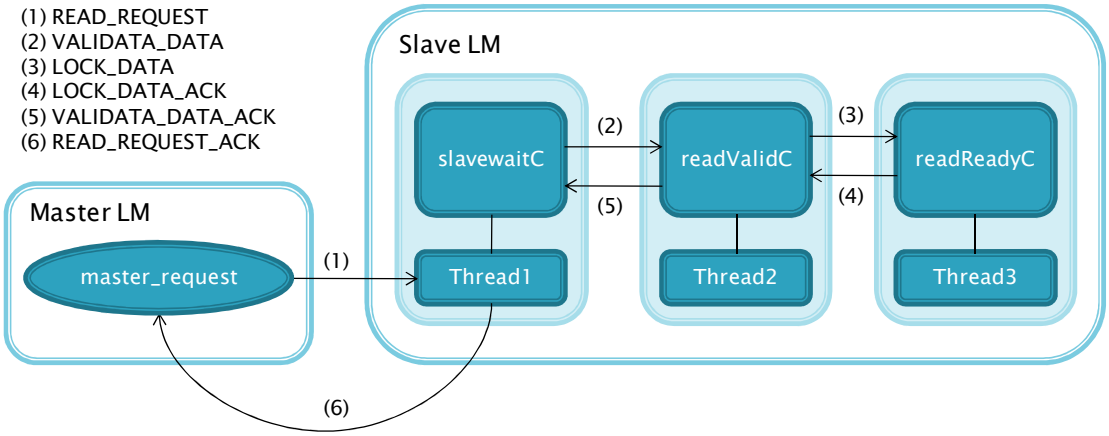



***Figure 8: Test Base Application design, OSE to UML communication***

In this test base application, the handshake between master and slave has the same protocol as the OSE to OSE communication application, but here only signal (1) and (6) are OSE signals. The signals inside the slave LM are UML signals that are delivered by the actors representing the capsules in each of the thread.

### 4.2.2 Log Collection

After completion of the test base application, another important task is to collect the log files. We need the logs generated for the same test case that is taking place in the same time interval in order to provide sufficient information for correlation analysis of the event data. The logs were collected by running the same application several times and then collecting the log files for each of these times.

From the OSE to OSE applications, Kernel Trace Log, Trace and Error Log, Error Log, Execution Address profiling log and CPU load log could be collected in the same time interval. From the OSE to UML application, all required logs could be collected including TSL log.

All logs should be cleared every time after log collection. In this way, it could assure that all logs collected are from the same time interval. The buffers used to store logs are limited, when the buffer is full, the new coming logs will replace the oldest logs entries which follow the algorithm FIFO (first in first out). As a result, the Test Base Application should be controlled to execute in a suitable time interval to make sure the buffers are not full or just full and no log entries are replaced. This could be done by running the application several times and find out the best suitable running time when the complete logs could be collected. In this way, the logs collected will have the same starting and stopping time.

## 4.3 Correlation Analysis

Individually, the log files will only provide limited information of events taking place in the system, but by correlating them a wider view and further information can be obtained. Especially interesting is correlation between the event and profiling logs (see *Figure 6*). These two groups provides information about the system layer and the application layer respectively, and by correlating them it will be possible to help bridge the gap between system problem determination and debugging of applications. In other words, it would be possible to gain further information when looking for problems in different products.

In *The Assigned Set of Log Files* section the log files where introduced, and the data elements that can be extracted from these are presented in appendix section *D.2*. To be able to correlate the event data of the different log file types, data elements describing the same events and either directly or in-directly corresponds to each other should be found. These are

the elements through which the event data can be correlated and they will be show below in the *Common Event Data* section. After this section each of the different correlation possibilities will be presented in more detail. The last sub section will cover the correlation accuracy of these correlation possibilities.

### 4.3.1 Common Event Data

Based on the table shown in appendix *section D.2*, the common event data amongst the log files can be found. Since the TSL log file type has many common data elements with all of the other log files we will describe these in a separate table.

The common elements of all the log files except the TSL log are described in *Figure 9* shown below. In this figure all the event data directly corresponds to each other through name or value.

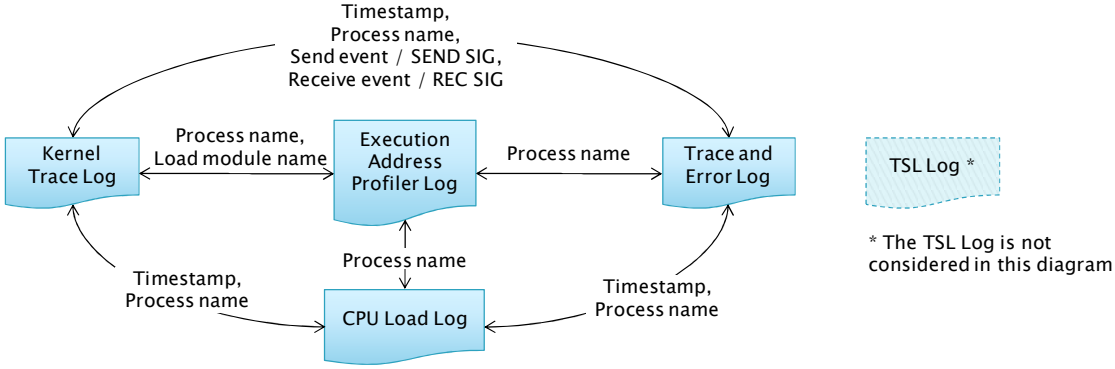



***Figure 9: Common event data between the log files***

*Figure 10* describes the common event data between the TSL and the other log files. In this case the different event data doesn't always directly correspond to each other. More details about the TSL common event data can be seen in the table presented in appendix section *D.3*. It is from this table that the diagram below was extracted.

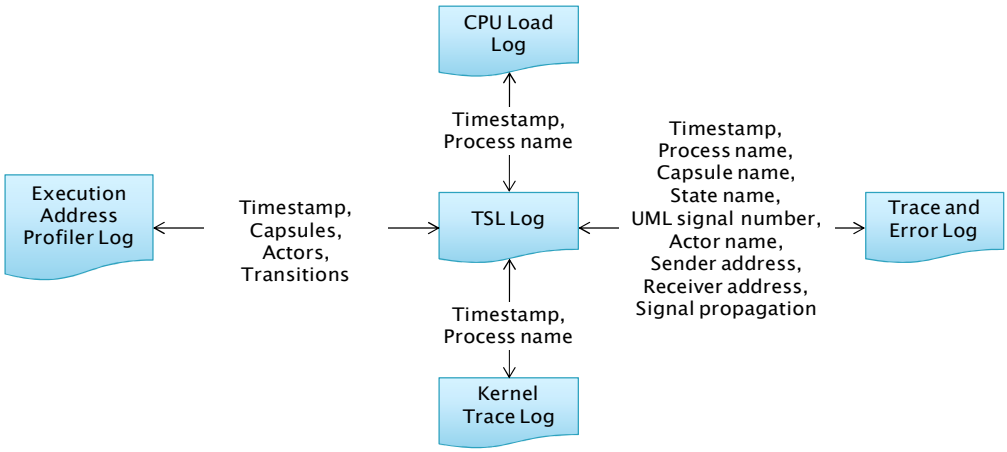



***Figure 10: Common event data with the TSL log file***

### 4.3.2 Correlation Possibilities

#### *4.3.2.1 Correlation over Time*

All the events that have a timestamp can be correlated over time. This could be useful since there is a big chance that the events taking place at the same time is dependent on each other or at least relates to each other in some way. Through the CPU Load log these events can also be compared to the CPU load at the time that the events take place.

The TSL log differs from the other log files since the event data is cumulative for each timestamp. Before being correlated with the other log it should be handled either by being compared with the accumulated information from another event type, or by subtracting the information from previous TSL time stamp event before comparison.

Before the event data can be correlated over time the timestamp first needs to be normalized to the same representation and same reference time. Converting time representation should be handled in the interpreting step and then stored in the data model in a unified format.
To be able to use the same reference of time some extra thoughts are required since this is handled in different ways for different log files. The time reference for KTR log is represented by a 32 bit integer to represent micro seconds. The integer will restart from zero when there are no more bits to represent the time[a] or on warm or cold restart of the system. The time reference for the other log files are represented by date and time and will continuously progress from time when the counter was last reset. In the case of the T&E log the time counter will be reset only on cold restarts.

Through the *Syslog*, which is not handled further in this project, it would theoretically be possible to automate a synchronization calculation. This log file keeps track of all different kinds of system restarts and has a timestamp that is not itself affected by these. However, when considering that different nodes will have different restarts, which would make things even more complicated, a better and much simpler solution is to simply let the user manually synchronize the time through a time delta in the user interface.

#### *4.3.2.2 Correlation over Process*

By collecting information about a certain process from different log files and at different times it will be possible to get a better view of how different processes interact with each other and with the system. Some of the information that can be collected for the processes include the T&E messages, received and sent signals and dependencies to other processes. The process dependencies are especially interesting since the including processes might be a possible explanation for a symptom in the system.

Different layers have different representation of processes. In the hardware layer a process is represented with a memory address, in the OSE layer the process is represented with the defined process name, and finally in the application layer the representation is type specific. For the TSL event data terms such as capsule names and actor names are used. It could be useful with the possibility to search one of these representations (e.g. memory address) and also get information about the others representations (e.g. process name) for the same process. In the set of logs that was analyzed, the physical memory address of a process can be found in the EAP log. If the memory address is mapped to the process name, the processes in KTR, T&E and TSL log would be searchable also on memory address.

How the processes are interacting with each other through signal propagation is also interesting and will be discussed in the next section.

#### *4.3.2.3 Correlation over Component Interaction*

***OSE Signal Propagation***

By combining information from different log files it is possible to track how OSE signal propagation relates through different parts of the system layers. These propagations and the event data that can contribute with further information are described below together with diagrams showing the interactions. The information in these diagrams is created from our Test Base Application.

[a] A 32 bit integer representing microseconds corresponds to approximately 71 minutes.

Since the KTR event data shows signal propagation between processes and information on what load module they belong to, it will be possible to see component interactions on a load module level similar to the diagram shown in *Figure 11*.

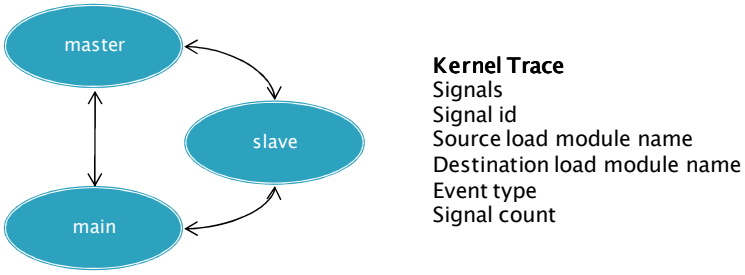



***Figure 11: Load Module interactions***

Through KTR event data it is also possible to see the interactions between the processes themselves (see *Figure 12*). Further, if the load module is created in RoseRT it will be possible to get the same interactions from the TSL log together with the delivery latency for the signal propagation. Further information about the processes can be gained from the T&E log by the group and message attributes.

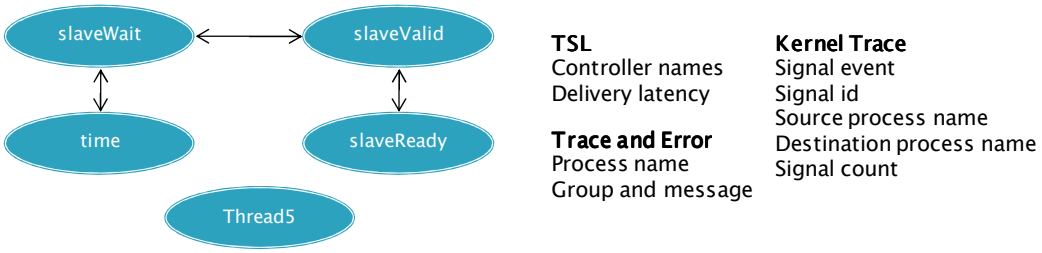



***Figure 12: Process / Controller interactions***

If the interacting processes / controllers are part of a Rose RT product it will be possible to find out how the signal propagates within this component (see *Figure 13*). Here the OSE signal is propagated through UML messages between RoseRT actors.

Information on the actor interactions is found in the TSL log together with the capsule they belong to. This information only provides the actor memory address, but with the help of the T&E log through the TRACE7 trace group it is possible to get the actual names of these actors. Through the EAP event data with the present name conventions this information can be further complemented with the capsule hit ratio.

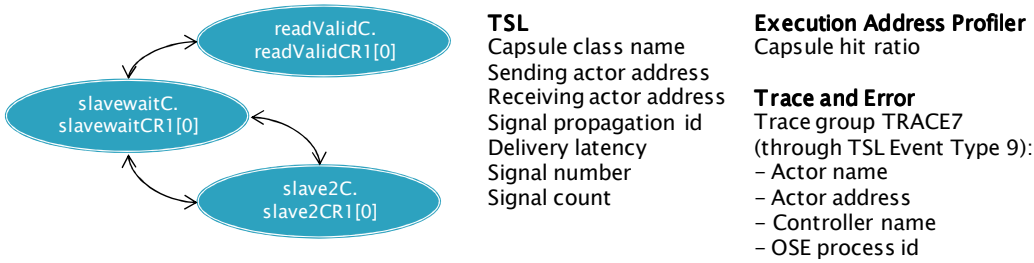



***Figure 13: Actor interactions***

### *Capsule States and Transitions*

If the model explained in the above section is extended further, the capsule states and transitions can be traced (see *Figure 14*). All these different states can be traced and recorded in the T&E log file through the STATE CHANGE trace group. Some of these transitions can

also be found in the TSL log and in this case information about the transition cost and message latency can be provided. By default only the 128 most cost expensive transitions will be shown in the TSL log, but this number might be re-defined by the user [25].

Each of the transitions corresponds to a symbol listed in the EAP log file. However, due to the EAP naming convention it is impossible to know what transition corresponds to what function. If transitions and functions could be mapped it would be possible to get information about the hit ratio for transition corresponding functions and also the physical address for these functions / transitions.

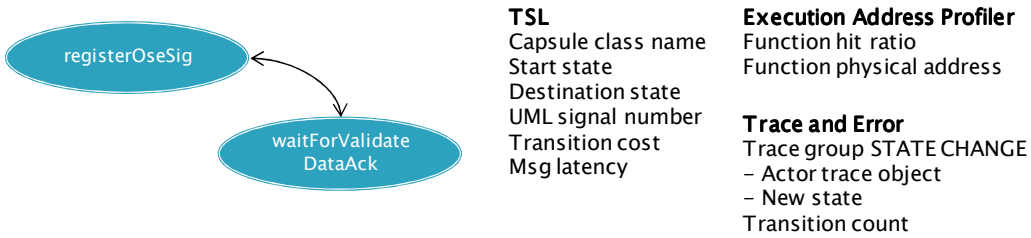



***Figure 14: Capsule states and transitions***

#### *4.3.2.4 Correlation between different runs*

If data is collected from different runs, these could be compared to each other. The anomalies could either be manually compared through a graphical view or they could be collected and shown to the user through a difference report.

#### *4.3.2.5 TSL Specific Correlation*

As has been introduced in previous part, TSL Log collects ten different events to help the designer to get a better understanding about the RoseRT application. For example, when tracing and logging the event data with the most expensive transition cost (Event Type 1) and message delivery latency (Event Type 3), it tells the designer what the worst case behavior of the model is and where to improve the RoseRT application.

Besides RoseRT Profiler, designers also use Trace & Error Macros to trace system and functional behaviour on a capsule level at RoseRT program development. This tool could be seen as a complement to RoseRT Profiler. Sometimes, information provided by a single debug tool is limited and not sufficient enough when observing complex problems. For example, TSL log provides the accumulated information such as the amount of state changes or how many OSE signals that has been received, while it does not provide any detailed information like when the states change took place or when each OSE signal was received. This information can instead be found in the T&E log, and by correlating these event data the designer might obtain further information that might aid when debugging the application.

There are some other debug tools such EAP Profiler and RDS that were not designed for RoseRT applications but could also be adapted by them. RoseRT designers do not use those tools for debugging or troubleshooting as they develop their programs. However, both the EAP log and K&T log contain common event data that could be abstracted from TSL log. Further useful information may be generated by correlating those common event data from independent sources.

In this section, the correlation possibility between TSL log and other logs (T&E log, EAP log, CPU load log and KTR log) will be analyzed based on the common event data that previously was shown in *Figure 10*.

***Correlation between TSL Log and T&E, KTR Log***

In Trace & Error log or KTR log, there is no event data related to signal priority, message queue or RTMutex, so the correlation analysis of Event Type 8 and 10 was skipped here.

***Event Type 4 and Event Type 9***

From *Figure 10*, it shows that both the TSL Log and T&E log contains the event data sender address and receiver address. Those addresses are the physical memory addresses of different actors. They are collected by Event Type 4 in TSL Log and Event Type 9 in T&E log. In the following part, the correlation via physical memory addresses is analyzed according to the simple example outputs of Event Type 4 in *Figure 15* and Event Type 9 in *Figure 16*.

```
       ***Signal Propagation Tree***
Signal propagation ID: 1       Signal: 5        sent by: 0
        received by: slavewaitC.1207479648      delivery latency: 1891
```

***Figure 15: TSL Event Type 4 example output***

```
 [1970-01-02 00:03:24.932] slavewaitCR1 ../../initializeAll.cc:182 TRACE5:
       [RTProfiler EVENT_TYPE_9 - Actor Information]
         Actor: slavewaitCR1 [0]
         Actor adress: 1207479648
         Physical thread (controller name): Thread1
         OSE Process: 66591
```

***Figure 16: Trace and Error TRACE5 example output***

From the above figures, it shows that both of them contain the physical memory address *1207479684*. With this address, all actors in the signal propagation tree could be mapped to their corresponding controllers and OSE process id. The signal propagation is described in actor level, and only T&E log provides information related to actors, thus only T&E log has possibility to correlate with signal propagation tree. While after mapping the address the signal propagation tree could be expressed in OSE process (physical thread) level and it will provide possibility to correlate event data in K&T log via processes. In addition, the capsules could also be mapped to their actors. From the above figures, it also shows that the capsule class *slavewaitC* has the actor *slavewaitCR1*. This result will be further used in the latter *Correlation Analysis* part of this report.

***Event Type 1 and Event Type 3***

In TSL Log, transition (Event Type 1) is described by a capsule name, two states, and the UML signal number that triggers the transition. Look at the example log sheets in *Figure 17* and *Figure 18* below.

In *Figure 17*, it shows that during the time interval from when the RoseRT Profile was started to the logging time *00:05:34*, one most expensive transition took place at capsule *slavewaitC* in *Thread1*. From a designer point of view, some questions that might arise when obtaining this information include: when did this expensive transition take place? Why is it so expensive? Is it possible to make the expense smaller? No answers could be provided by TSL log because it only stores accumulative information. While some of the questions could be answered with the help of related event data recorded in T&E Log.

In the T&E package, the trace groups STATE CHANGE and REC SIG can be used to describe a transition when they are combined together. By doing this, detailed information about transition could be obtained. However, the trace groups STATE CHANGE and REC SIG are based on actor level, while the transitions recorded in TSL log are described based on capsule level. This means that to be able to correlate them the capsule name has to be mapped with the actor name as discussed in the previous section. This could be done by mapping the physical memory address from Event Type 4 and Event Type 9.

From *Figure 17*, it shows that the most expensive transition took place when capsule *slavewaitC* changed state from *waitForValidateDataAck* to *sendReadReqAck* after receiving UML signal 4. *Figure 18* shows that the UML signal which triggers the transition was received at time *00:05:30.916* and that the transition was triggered at time *00:05:31.144*. It implies that there might be some bottlenecks between time *00:05:30.916* and time

*00:05:31.144* since the most expensive transition cost took place within this period. Further log analysis should be focused on this period.

```
                [RoseRT Profiler Data] Fri Jan 2 00:05:34 1970
…
***Profiler (Controller = Thread1) ***
 Profile Collect Time    Seconds: 129   nanoseconds: 746279000
        Key: slavewaitC: waitForValidateDataAck_sendReadReqAck_4
        Value: min: 4507010     max: 5131949    med: 4969294
```

***Figure 17: TSL Log Event Type 1 example printout***

```
…
[1970-01-02 00:05:30.916]
slavewaitCR1../src/target/Cello/RTActor/enterState.cc:59 STATE CHANGE:
waitForValidateDataAck
[1970-01-02 00:05:30.916]
slavewaitCR1../src/target/Cello/RTActor/logMsg.cc:76 REC SIG: Signal:
VALIDATE_DATA_ACK, Port: slaveWait [0], Sender: readValidCR1 [0]
…
[1970-01-02 00:05:31.144]
slavewaitCR1../src/target/Cello/RTActor/enterState.cc:59 STATE CHANGE:
sendReadReqAck
```

***Figure 18: T&E Log STATE CHANGE and REC SIG example printout***

From the above example, it shows that by correlating transition from TSL log and T&E log, the designer could obtain the integrated knowledge about what the worst case is and when it took place and the improvement could be done according to further analysis.

The message delivery latency (Event Type 3) is represented in the same way as transition cost in TSL log, while in the T&E log, message delivery latency needs to combine information collected from trace group SEND SIG, REC SIG and STATE CHANGE. When this is done it could be correlated in a similar way as the transition cost.

#### *Event Type 5, 6 and 7*

Typical question could be answered with profiling Event Type 5 (uml message send count), Event Type 6 (uml message receive count) and Event Type 7 (state changes count) is "what is the most frequent executed actors within a RRT application". It could help the designer to get a good understanding about the behaviors of a RRT application. If the designers want to obtain more detailed information like what kind of message was sent/ received by the most frequent executed actors, they have to examine the T&E log.

The Trace and Error macros provided different types of trace group for RoseRT applications on capsule level or actor level. In RoseRT application, event data related to uml message received and sent for each actor could be traced and logged by the trace group REC SIG and SEND SIG respectively, and the event data related to state changes for each actor could be traced and logged by the trace group STATE CHANGE.

More detailed information like received signal number, signal time and the sender of the signal for the most frequent executed actors could be abstracted from the T&E log with the help of those trace groups. By combing these detailed event data together with the accumulative information got from Event Type 5, 6 and 7, the designer can get a better understanding of the application behavior.

#### *Event Type 11*

Event Type 11 collects the number of received messages (the internal messages, external messages and OSE messages) within a controller. There are one counter for each type of messages. By counting and logging the received messages, the designers can obtain the knowledge about communication frequency for each controller. However, if the designers want to do a deeper analysis to the communication behaviour of a RoseRT application,

event data collected by this event can be a bit deficient. There is no information about the senders, signal contents or message sending time existing in the TSL log file. While these detailed event data could be abstracted from T&E log. Thus, the designers will have to correlate Event Type 11 from TSL log with related event data from T&E log in order to get a deeper understanding of the communication behaviours. The ways to perform the correlation will be a bit different in terms of the message types under correlating.

The trace group TRACE7 is used to record the reception of OSE signals for each thread. With the help of two attributes TRACE7 and thread name, all entries related to OSE signal reception could be found out from T&E log. By doing this, the following information could be abstracted: received OSE signal number, sender process id, and the message receiving time. Among the assigned log files, only T&E log contains OSE process id, it should be mapped to the corresponding OSE process name in order to enable further correlation. This could be done with the help of RDS. All processes executing on the target could be displayed together with corresponding process type, id and priority by typing the command *"rds display process"* to OSE Shell. The process id abstracted from T&E log could be correlated with the one got from RDS; as a result, the designer could get the following integrated information: received thread name, received OSE signal count, and received OSE signal number, sender process id, sender process name and the message receiving time. As introduced in previous section, Kernel Trace log also contains sender name, OSE signal number and receiver name. Those common event data could be further correlated to find out all related entries in Kernel Trace log from which the signal sending time and the load module name of the sender could be abstracted. When this is done the designer could get an integrated knowledge about the communication behaviour of a RoseRT application with another application.

To be noticed, dynamic processes could have several different process ids for every time when it was created, a new process id will be assigned. While static process always uses the same id until the application is restarted. The event data got from the command *"rds play process"* just contains current process id of a dynamic process. Thus, it is recommended to log process name instead of process id in T&E log so that the mapping procedure from process id to process name could be avoided.

Trace & Error macros provide the trace group type *REC SIG* to log received UML signals (both of UML internal and UML external signals) in an actor level. From Event Type 11 in TSL log, the designer could get the information about how many UML internal or external signals were received for each controller. The logged controller name of Event Type 11 could be correlated with event data got from the trace group TRACE5, by doing this all actors name within this controller could be abstracted. After knowing all actors name within a controller, all logged entries related to UML signals could be found with the help of the trace macro *REC SIG*. By analyzing those entries together with related event data in Event Type 11, the designer could understand the UML communication behaviours of a RoseRT application at a higher level.

***Correlation between the TSL log and EAP log***

From Figure 10, it shows that both TSL Log and EAP log contains event data about timestamps, capsules, actors and transitions. The time stamp could be correlated directly, while it could not help too much for the system understanding without further event data correlation because both of the two logs store accumulative information and too many entries will be got if only timestamp is correlated.

If the transition described in TSL log and EAP log could be correlated, it will provide the designer with more important information. In TSL log, some most expensive transitions or message delivery latency are collected, in EAP log the executing frequency for each transition is collected. If they could be correlated into a consistent context, the designers

could get a much more integrated view about the system behaviours. By doing this, the designers could obtain the knowledge about what the worst case of a RoseRT application is in terms of transition cost or message delivery latency and what the executing frequency is of the worst case. Base on this result, designer might do some improvements to the application. The first hot point in improve will be the transition with high frequency.

It is useful to correlate transitions from TSL log and EAP log, while it is impossible with current collected event data in two log files. In TSL log, transition was described with capsule name, two states, and one UML signal number that trigger the transition between two states, which is shown in *Figure 19*. While in Execution Address Profiling log, transition is expressed as a symbol name including capsule name, actor number, transition number, and UML signal content, which is shown in *Figure 20*.

```
Key: slavewaitC: waitForValidateDataAck_sendReadReqAck_4 Value: min:
4147847 max: 13643694 med: 9475573
```

***Figure 19: TSL Event Type 1 (transiton cost), example output***

```
Slave1: _ZN16slavewaitC_Actor30transition3_gotValidateDataAckBaseE 1  0.00%
Slave1: _ZN16slavewaitC_Actor25chain3_gotValidateDataAckEv         1  0.00%
```

***Figure 20: EAP log, transition example output***

In *Figure 19*, it shows one of the expensive transitions stored in TSL log. The transition took place between state *waitForValidateDataAck* and *sendReadRedAck* when *slavewaitC* received the UML signal which was represent by number 4. In *Figure 20*, it shows that the transition represented by the symbol name `_ZN16slavewaitC_Actor30transition3_got ValidateDataAckBaseE` was hit once during measuring period. It is so difficult to know if they are describing the same transition in *Figure 19* and *Figure 20*. Thus we suggest adding some extra information about the states and received UML signal number just as what have been done in TSL log to the symbol name in EAP log when it describes transitions. By doing this, the two aspects of the transitions from TSL log and EAP log could be combined together and provide a better view of the system behaviour for the designers.

#### *4.3.2.6 Further Correlations*

Countable attributes can be compared and shown in relation to each other. Signal count between components can for example be compared to the component hit rate or to the CPU load for a certain process or priority. If signal count is put in relation to signal latency this might give further information for tracking system performance.

Another form of correlation is filtering of attributes to show only the interesting set of information. This will be mentioned more specific in the *Graphical Representation* and *Graphical User Interface* sections.

### 4.3.3 Correlation Accuracy

A general limitation when correlating timestamps is that the events many times are not granular enough to sufficiently trace a chain of events between log files. T&E log has the maximum timestamp accuracy of milliseconds while the KTR and TSL log has microseconds as maximum accuracy.

If attributes are compared over a string variable, there is a chance that the attribute has the same value even though they represent different elements. In the assigned set of logs, the process, method and symbol names can avoid these circumstances to some extent by also compare through the load module or method package that the component belongs to. Similar precautions can also be made for the string attributes in the TSL log.

## 4.4 Graphical Representation

Humans are very good at recognizing patterns and anomalies in a visual context [26]. By representing the event data graphically an overview can be presented from where the user more easily can navigate and analyze the data. Based on this together with results from previous sections, analysis about visualization will be given. The results from these sections will later be used as a base for the Graphical User Interface (GUI) implementation of the Correlation Tool and the GUI implementation will in turn validate the theoretical results concluded in this section.

Since the there is no GUI implementation for the TSL log file in this project, extended graphical analysis about this log type is done in the *Detailed TSL Specific Representation* sub section.

### 4.4.1 Table Representation

All the event data analyzed in this project can in some way be represented in a table. From here the entries can be sorted by sequence number, timestamp, component name or any other attribute according to the user preferences for the moment. It is also a great way for navigating, filtering and searching properties of large amounts of associated event data. An example of a table showing the entries of a KTR log file is shown in Figure 21.

```
…
(21) Time:    1279612.079 ms
  Receive <1001> From: m15.ppc:master_request To: s15.ppc:slave_wait
(22) Time:    1279612.091 ms
  Send <1002> From: s15.ppc:slave_wait To: s15.ppc:read_valid
…
```

| Sequence number | Time (ms) | Event | Signal ID | Source | Destination |
|---|---|---|---|---|---|
| … | | | | | |
| 21 | 1279612.079 | Receive | 1001 | m15.ppc:master_request | s15.ppc:slave_wait |
| 22 | 1279612.091 | Send | 1002 | s15.ppc:slave_wait | s15.ppc:read_valid |
| … | | | | | |

***Figure 21: Table representation of Kernel Trace entries***

Another use of the table view is to show collected information for frequently occurring attributes such as component name or event type. The collected information could contain statistics such as number of occurrences for components or event types, the component dependencies, etc. Figure 22 shows a table representation of process information collected from the KTR log file.

| Process | Occurrences | Receive events | Send events | Number of dependencies | Dependencies |
|---|---|---|---|---|---|
| … | | | | | |
| s15.ppc:slave_wait | 10 | 5 | 5 | 2 | master_request, read_valid |
| s15.ppc:read_valid | 10 | 5 | 5 | 2 | slave_wait, read_ready |
| … | | | | | |

***Figure 22: Table representation of Kernel Trace processes***

### 4.4.2 Representing Log Information Summary

It would be good for the user if he in some way could get an overview of the different log files without actually have to open and look through the data. Information that would be useful to know is how many entries the log file contains, what time span the events take place, what different event types it contains etc. This kind of information could be shown either in a log properties view, a static view or as a dialog. Further information could be a short description of the log file type, and if multiple files are selected, information on how they can be correlated.

### 4.4.3 Time Chart Representation

An intuitive way in representing event data in relation to time is with the help of a time line chart. By doing this, the events can easily be correlated over time by being represented in relation to the time line and to each other. An overview can be presented that would be impossible to see through raw text or a table representation. With various colors or shapes it would also be a fast way to spot entries that stands out from the usual patterns. The data can be navigated through a zoom function where more detailed information can be given at a certain time interval. The figure below shows an example time chart with events in relation to time.

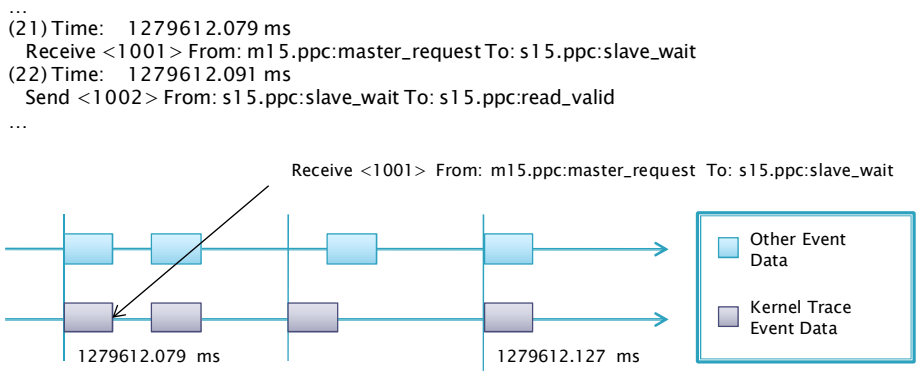



***Figure 23: Time chart view representing events in relation to time***

If the event entries specifies countable data such as the case for CPU Load Log, this data can be represented with a diagram showing how the value changes over time. The diagram can be shown together with the event representations explained above to put these in the same context. The user would in this way be able to see the events that were recorded at the time of anomalies in the countable event data. For CPU load event data it could also be valuable to compare diagrams from different runs to each other.

Further functionality could be tool tips for the events, clickable icons to show further information, a legend for the time charts and a filter where the user can specify what data to show.

### 4.4.4 Node Graph Representation

Events that contain source and destination components can be represented graphically by nodes and connections similar to the graphs in the *Correlation over* section (section 4.3.2.3). The figure below shows a simple example of KTR event data being represented by nodes and connections.

```
...
(21) Time:    1279612.079 ms
  Receive <1001> From: m15.ppc:master_request To: s15.ppc:slave_wait
(22) Time:    1279612.091 ms
  Send <1002> From: s15.ppc:slave_wait To: s15.ppc:read_valid
(23) Time:    1279612.109 ms
  Receive <1002> From: s15.ppc:slave_wait To: s15.ppc:read_valid
(24) Time:    1279612.113 ms
  Send <1003> From: s15.ppc:read_valid To: s15.ppc:read_ready
(25) Time:    1279612.127 ms
  Receive <1003> From: s15.ppc:read_valid To: s15.ppc:read_ready
(26) Time:    1279612.132 ms
...
```

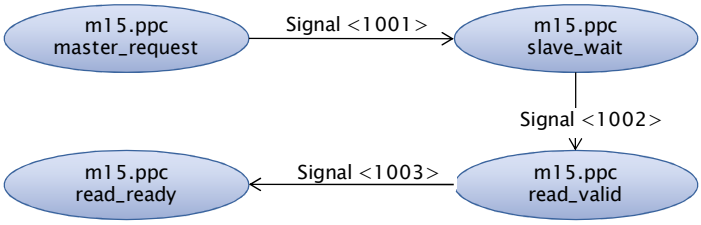



***Figure 24: Node graph representing Kernel Trace signal propagation***

At a lower level of event propagation the information would be too comprehensive to show all at once, why a filter function is crucial to use in these cases. Again, as mentioned in the Correlation over *Correlation over Component Interaction* section, the information given by our assigned set of logs can give information of interactions from a load module level, through process/controller and capsule level, down to the transition level of Rose RT capsules. Since all this information is given it would be possible to traverse the levels of representations through children or parent components. If for example a load module node is clicked, the interactions between processes within this component can be shown. Further, if the process is created by RoseRT, this component can be clicked to show information of the capsule interactions taking place in that particular process; and finally an actor can be clicked to show information of the state transitions taking place in the controller from where the actor was instantiated from.

This way of interacting with the event data can be used if the user wants to see the OSE signal propagation graphically to get an overview of the system. It would also be useful if the graphics can be connected to the actual values represented by the event data. The connections could be connected to information such as signal number, amount of signals sent, and delivery latency; while the nodes can be connected to information such as process name, capsule hit ratio, or any other information provided by the log files.

### 4.4.5 Statistical Representation

Event data representing statistical information can be shown through various charts such as bar charts, line charts, pie charts and others. This representation might be nice to look at, but for the set of log files created with the Test Base Application, we found these representations less useful. It could however be a good idea to use at times when the data is too comprehensive to view in a table or for finding anomalies when comparing data recorded from different runs.

### 4.4.6 Detailed TSL Specific Representation

According to the event data and correlation analysis of TSL log in the previous section, some suggestions about how to represent the result graphically are given in this section. Because of time limitation, the GUI of TSL log is not implemented. But our suggested graphical presentation could provide some valuable results for next project work.

#### 4.4.6.1 TSL Log Browser View

Before going into the details of TSL log, it is better to have an overview such as how many controllers there are and what they are about respectively, which types of events are collected for each controller or all controllers.

Two alternative ways about the TSL log browser view are suggested in the following. The first way is to show an overview according the original structure of the TSL log files. There are 4 different objects: log, block, controller and event in TSL log. So a log file structure with 4 depths could be used. This structure is shown to the left in *Figure 25*.

Another way to show an overview is a simplified structure according to different event types stored in the log files. Three depths are used in this type of log file view: log, event, and controller. Blocks are neglected here and all controllers collect the same event will be gathered together and listed under the event type. This structure is shown to the right in *Figure 25*.

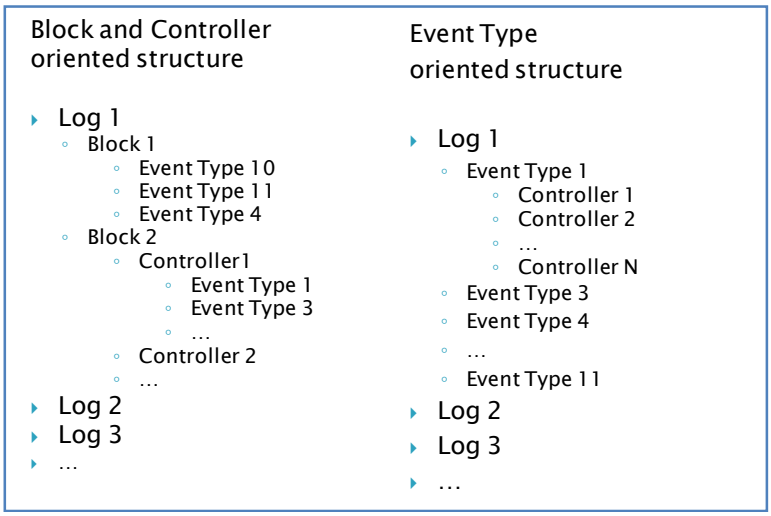



***Figure 25: Two different TSL log browser view structures***

Both of the two suggested log browser view structures have their strong points. For the block and controller oriented structure which is shown to the left in *Figure 25*, it is easy for the user to identify how many different controllers there are and what they are about respectively in the log file, and what event types are collected by every controller. In this way, it is not easy for the user to identify how many controllers collected one specified events if the controller number is a bit large.

For the event type oriented structure it is easy to identify how many controllers that contain a specified event. However, in this way it is not so convenient for the user to identify the event types that are collected for each controller. The best way is probably to present this in both ways and let the user make the decision on how to show the content.

#### 4.4.6.2 TSL Table view

As mentioned in *Table Representation*, section 4.4.1, all event data can in some way be represented in a table view, including event data in TSL log. It is however not a very good idea to represent all event data abstracted from TSL log in only one table because of its complexity. A better solution would be to represent event types with different structures in different tables. According to the content of the different TSL event types, four tables are suggested to be used for storing the event data. Event data in Event Type 1 and 3 can be presented together in one table as shown in *Figure 26*; event data from Event Type 5, 6 and 7 can be presented in another table as shown in *Figure 27*; event data from Event Type 10 and 11 can be put in a table as shown in *Figure 28*; and finally the event data from Event Type 4 should be presented in separate table.

| Controller | Capsule name | 1'st state | 2'nd state | UML sig num | Transition cost | | | Latency | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | min | max | med | min | max | med |
| main | Slave | wait | stop | 1 | 165 | 165 | 165 | 112 | 112 | 112 |

***Figure 26: Example table view of Event Type 1 and 3***

| Controller | Collect time (sec) | Actor name | Actor index | Rec msg | Sent msg | State change |
|---|---|---|---|---|---|---|
| main | 106.422747000 | slave | 0 | 1345 | 0 | 1103 |

***Figure 27: Example table view of Event Type 5, 6 and 7***

| Controller | Inter queue per priority | | | | | | | | Intra queue per priority | | | | | | | | Defer | OSE signal | UML intra | UML inter |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | | | |
| slave | 2 | 3 | 3 | 1 | 2 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 3 | 1 | 1 | 1 | 2 | 20 | 34 | 23 |

***Figure 28: Example table view of Event Type 10 and 11***

| Sig prop id | UML sig num | Send by | Received by | | Cost |
|---|---|---|---|---|---|
| | | | Capsule name | Actor memory addr | |
| 1 | 5 | 0 | slaveWaitC | 1250078048 | 872 |
| 1 | 3 | 1250078048 | readValidC | 1250079136 | 1603 |
| 1 | 3 | 1250079136 | readReadyC | 1250078672 | 890 |
| 1 | 4 | 1250078672 | readValidC | 1250079136 | 582 |
| 1 | 8 | 1250078048 | slaveTslTopC | 1250077376 | 4161 |

***Figure 29: Example table view of Event Type 4***

### *4.4.6.3 Node graph*

In *Figure 30*, the node graph of a simple signal propagation tree is shown. As has been analyzed in the previous part, TSL signal propagation tree could be represented in two layers: actor layer and controller layer. Both of the two layers are shown in *Figure 30*. The actor based signal propagation tree, which is abstracted from original log file, is isolated from OSE. The process based signal propagation tree on the other hand could be traced and logged both through the KTR and the TSL log.

With the help of this node graph, it is much easier to identify the signal dependencies among different capsules and different controllers. A sample output of signal propagation tree represented as a node graph can be seen in *Figure 30*.

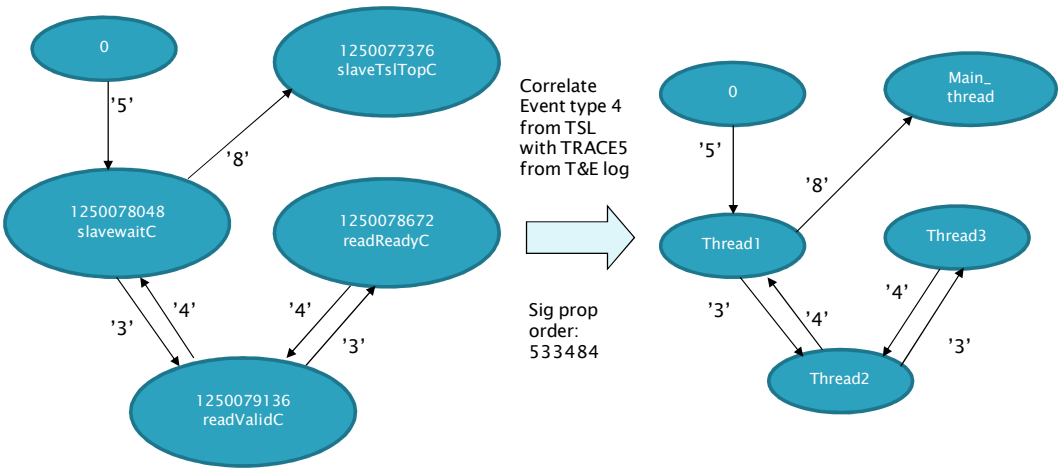



***Figure 30: TSL node graph of signal propagation tree***

Also the event data from TSL Transition Profile Events (Event Type 1 and 3) can also be represented by a node graph, in a similar way as above and as shown in *Correlation Possibilities*, section *4.3.2.3*.

### *4.4.6.4 Extended Time Chart View*

TSL log collects accumulative information for the RoseRT model so there might be hundreds of entries in the time line for each logging time. It could not help the users too

much if all the entries are shown at the same time. Some techniques are needed to filter out event data that the users might not be interested. We suggest two ways to perform this function shown in *Figure 31*. One type of the two suggested TSL log browser views (show in *Figure 27*) is also included in this figure.

As shown in *Figure 31*, one way to filter in some entries from time line is to design a filter dialogue with some main attributes listed behind checkboxes. Take a simple example, when the checkbox of the attribute *"tran cost"* was selected, only the event data related to transition cost (*Event Type* 1) will be shown in TSL time line. There might be still many entries at the same point of the time line if several controllers collect *Event Type 1*. The user might just want to show *transition cost* for a specified *controller* or even more specific like for a *capsule*. In that case, a specific filter is suggested. Sometimes the users might not remember the name of different controllers that exists in the log files, in this case, the log browser views suggested in section could be helpful. By looking at the log browser view shown to the left in *Figure 31*, it is easy for the users to get an overview about the TSL log files. It might also be possible to add a filter function to the *TSL log browser view*. By doing this, the uses can open a time chart view with only interesting entries directly from the log browser view.

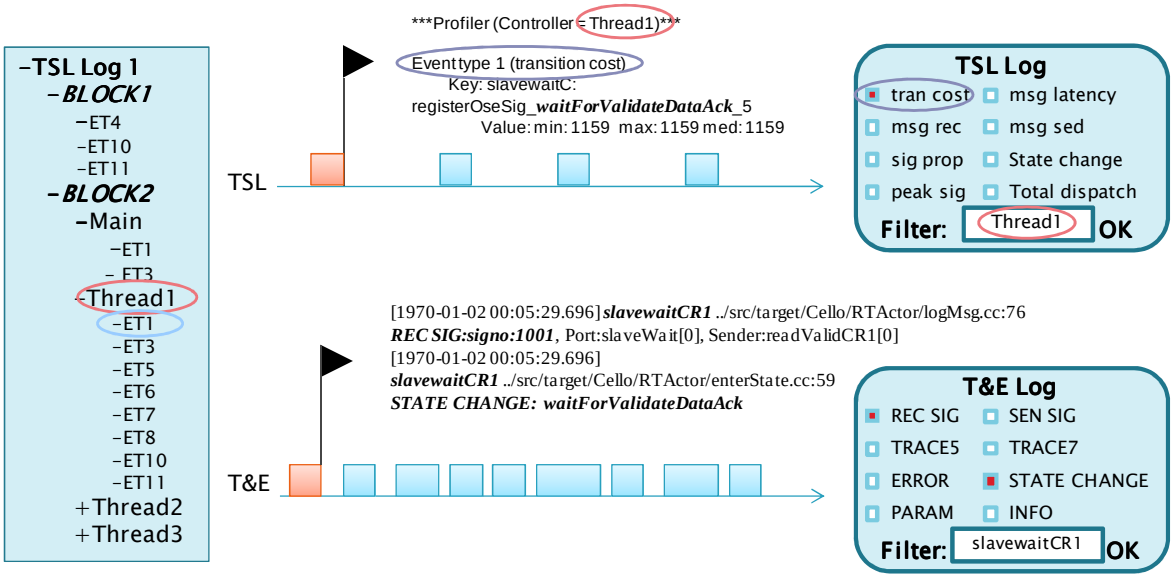



***Figure 31: TSL event data represented on a time line with suggested filters***

*Figure 32* shows a simple example about how to correlated event data from TSL log and T&E log by time line. With the help of the filter shown in *Figure 31*, three transition entries in *Thread 1* could were found out, it is also suggested to show the transition in a node graph as shown in *Figure 32*. Detailed event data of transitions could be shown in T&E time line via selecting the attributes *REC SIG* and *STATE CHANGE* in the filter dialogue and specifying a more specific name like *slavewaitCR1* (look at example in *Figure 31*) for T&E time line. *Figure 32* gives an overview about how the result will be like when correlated a transition. By doing this, it is easy to find out the time interval when the worst case took place in the time line and further analysis could be done by zooming in that time interval to check what something else happened during that period.

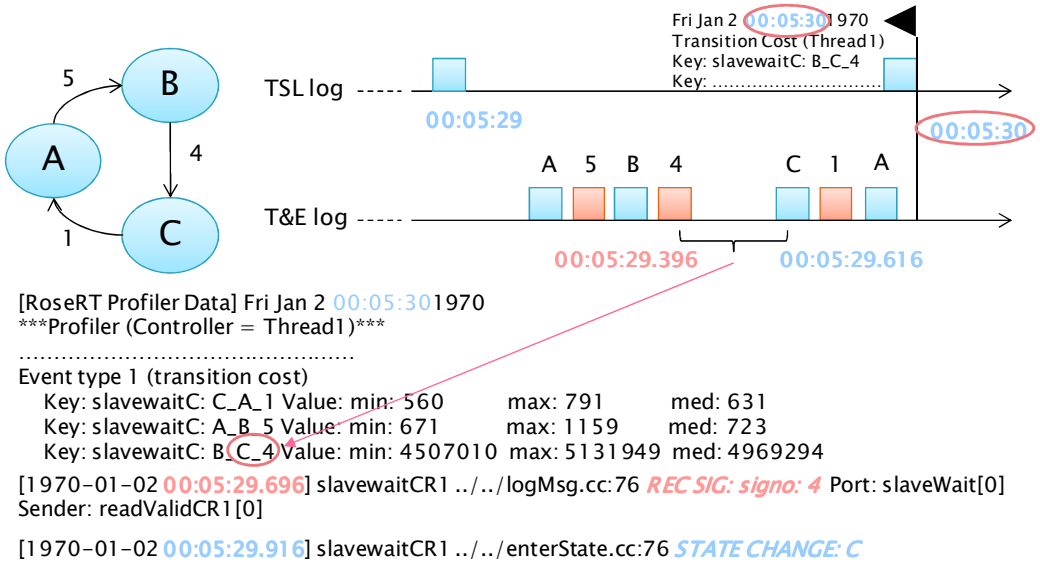



***Figure 32: Event data correlation via time lines***

The time chart view might be useful for representing the event data and it will be more powerful when complementing with log browser view, node graph and table view.

# 5 Implementation of the Correlation Tool

The Correlation Tool is implemented as an Eclipse Plug-in. It consists of parsers, data models, Graphical User Interface (GUI) and underlying calculations. By implementing this tool we will be able to confirm our previous results as well as allowing further analysis. With the GUI implementation we will also give suggestions on how the user can interact with the event data and the correlations. The aim is to facilitate in debugging and resolving problems within the system.

The *Interpreters* and *Data Model* sections will handle decisions on structuring and organizing the event data with the help of interpreters and data models. Following these sections the GUI will be presented together with the Correlation Tool functionality in the *Graphical User Interface* section. The last section of this chapter is the *Software Architecture* section where the Correlation Tool architecture is presented with further information on how the different components are connected.

## 5.1 Interpreters

To traverse and do calculations on the event data, it should first be interpreted and stored in a data model. Correlation and calculations could be done before storing the information in the data model, but at this stage the advantages was not apparent and it was decided to only do the normalization processing at this stage. Correlation pre-processing will however be discussed in the *Concluding Discussion* chapter in the end of this report based on our experience working with the event data.

From the elements presented in the *Event Data Analysis* chapter there are a few that needs to be normalized before being stored in the data model. First and foremost is the timestamp where all the sources are normalized in a uniformed format to enable correlation over time (see the *Correlation over Time* section). Further normalized data are the signal id numbers that are either stored in decimal or hexadecimal format in the different log file types.

The log files that was handled are all in raw text format and most of them are strictly structured which helps when they are to be parsed and stored in a data model. Since performance isn't an issue in this project work it was decided to use the java.util.regex library[a] for parsing the log files. The parsers have been implemented independent from the user interface, so that future parsers can be added in any way without affecting the rest of the system. All event data that is handled in the Correlation Tool will depend on the data model, and how the information gets there is of less weight.

The TSL log file differs from the other log files by being very complex and without any strict format. Regular text parsing is for this reason not a good option, and instead a suggestion of an XML structure was proposed to Ericsson. This structure can be seen in appendix section *D.4*. The structure was however never implemented (see *Complications* section in the *Concluding Discussion* chapter) and for the moment only an example parser for the structure was created. This parser handles TSL Event Type 1 and 3 and is implemented using the JDOM library[b].

## 5.2 Data Model

For storing the event data, a generic data model has been considered in comparison with the use of specific data models defined for each different type of event data. For better

[a] The Java regular expression API

[b] JDOM is a Java-based solution for accessing, manipulating, and outputting XML data from Java code (www.jdom.org)

compliance with the Correlation Tool the implementation was done as a Java Object Oriented structure. Discussion and conclusions about the data model approaches can be found in the *Specific vs. Generic Data Model* section below. Following this section the defined data models will be presented.

### 5.2.1 Specific vs. Generic Data Model

Both the specific and generic data model approaches was conducted for the data model implementation. The specific data model was our first approach, but due to bad experience using this strategy and information found on the Common Base Event we choose to change to a generic data model approach. A generic data model approach will contribute in a bigger abstraction and will facilitate when developing common views for different kind of log formats. This approach will also assist developers in adapting new log and trace data without rewriting the existing code. The decision points that were used for this conclusion can be found in appendix section *B.2*. Due to our implementation strategy of data model de-coherence it was a minor problem to also support a generic data model.

The diagram below shows a simple overview of the Correlation Tool implementation and what is needed by the user to extend the tool with a new log file type. The left side shows the extension requirements in a specific data model approach and the right side shows the requirements in a generic data model approach.

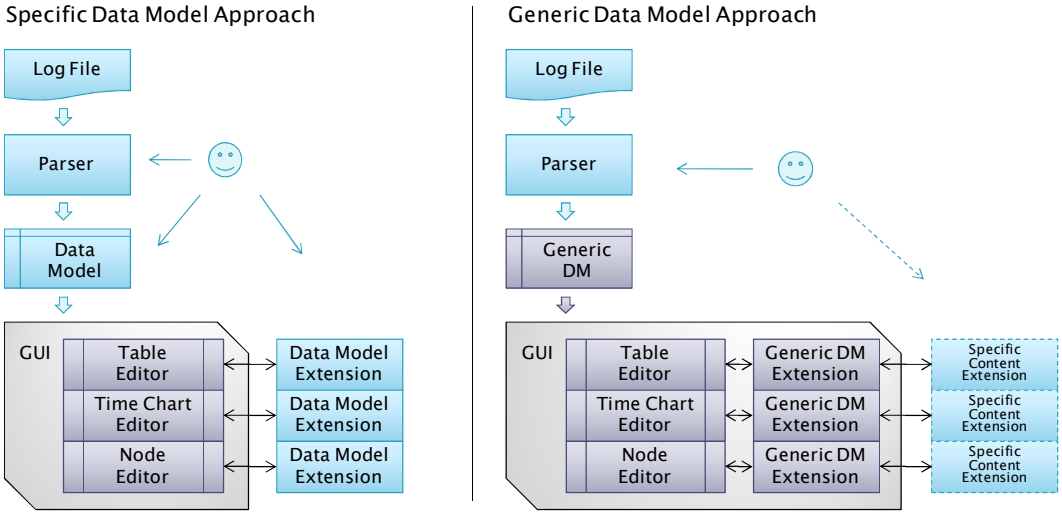



***Figure 33: Extending the application with a new log file type***

### 5.2.2 The Generic Data Model

The figure below shows the generic data model design. There was no time at this stage of the project to get familiar with the Common Base Event[a] but the design of the presented data model is still inspired by the same approach. The data model is currently defined to work for KTR, T&E, and EAP event data and the CPU Load data might also be used through the Extended Data Element. Due to the complexity of the TSL log, we chose not to use a generic approach for this log file type.

[a] See the *Common Base Event* section

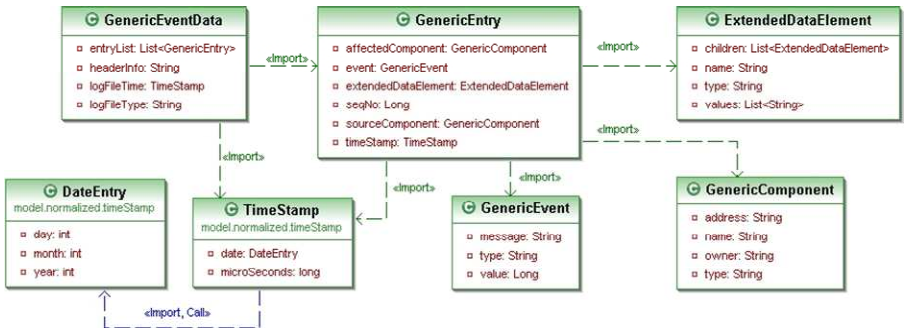



***Figure 34: The generic data model***

In the presented data model the different attributes are optional and the Correlation Tool has to adapt the output depending on what information that is presented. In the *GenericEventData* class the header info, log file time and log type can be specified. It also contains a list of generic entries where information from each entry in the log file is specified. The *GenericEntry* class stores information of sequence number, time stamp, source component, affected component, and the event. The source and affected component are of the type *GenericComponent* and here information about functions, methods or processes can be specified. The event attribute is of type *GenericEvent* and here information about event type, event value and event message is stored. In the *ExtendedDataElement* class further event data can be added if the generic structure is not sufficient. The *TimeStamp* class store dates in the *DataEntry* class and the time units from hours and below is converted to microseconds[a]. The full date and time can be collected through the *toString()* function.

### 5.2.3 The Specific Data Models

Since the specific data models that were constructed for the KTR, T&E, EAP, and CPU Load Event Data now are replaced by the generic data model, these will not be explained or shown in this section. They are however still supported in the implementation and our results concerning these can be seen in appendix section *B.1*. The TSL log is however still presented with a specific data model, which is shown in the diagram below.

[a] To normalize the time for future correlation

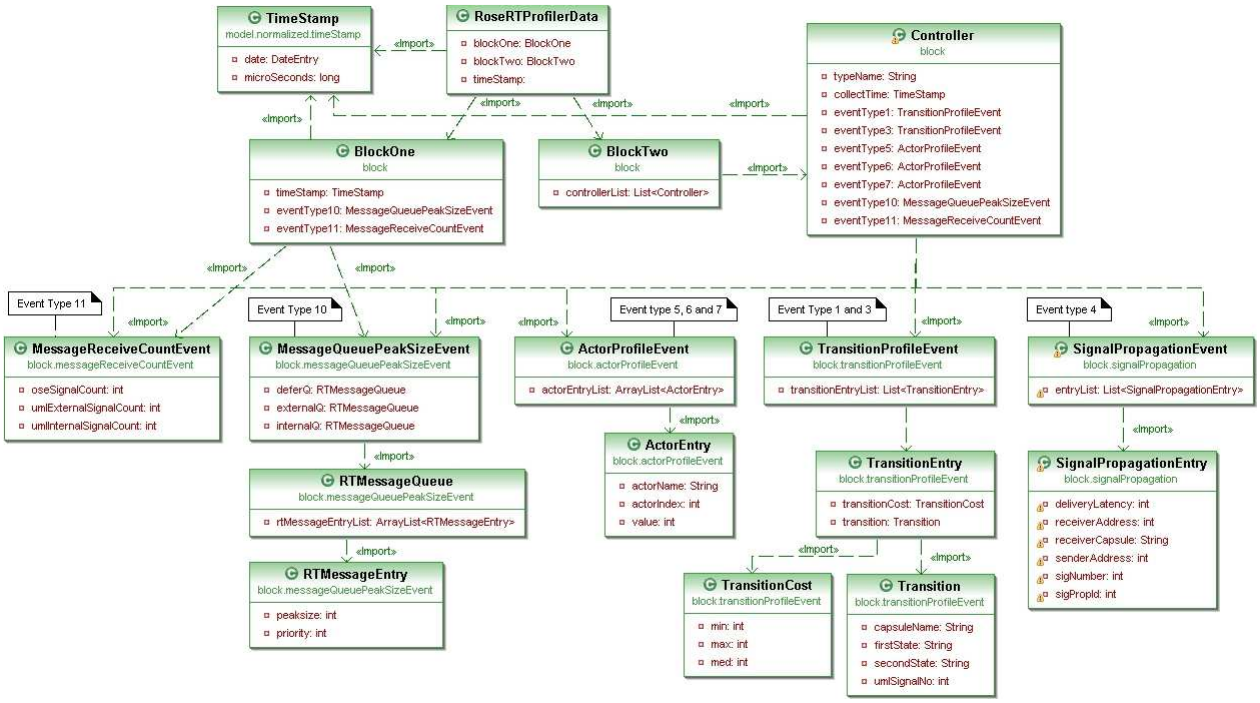



***Figure 35: The TSL specific data model***

The *RoseRTProfilerData* class is the main class. It contains the two different TSL blocks and a time stamp. Both of the block representations contain different controllers, and the controllers are in turn storing information of the different event types. The *TimeStamp* class that is used by both the main class and the *BlockOne* class, stores the time in a *DateEntry* class and in microseconds for hour units and below. The *TimeStamp* class used by this model is the same as the one used in the generic data model.

With the TSL data model we took the freedom to suggest some new names for the different TSL log file attributes. The intra and inter queue is now called internal and external queue to avoid confusion. *Event Type 1 and 3* are both referred to as *Transition Profile Events* and *Event type 5, 6 and 7* are referred to as *Actor Profile Events*. *Event type 10 and 11* are called *Message Queue Peak Size Event* and *Message Receive Count Event* respectively.

## 5.3 Graphical User Interface

In this section screenshots will be shown of the different components in the finished prototype GUI together with explanation and the thoughts behind them. All the event data representations can also interact with each other depending on what elements that are selected by the user. This will be explained further in the *Interaction between Representations* section. Features not implemented but that might be useful are presented in the *Future Work* section.

Information on how the finished interface can be used to find the reason for an error or certain behavior can be found in appendix section *C.2*.

### 5.3.1 Approach

The GUI implementation is based on the previous graphical analysis, and with the help of our Test Base Application we have constructed a few scenarios where we want our GUI to be useful.

Since the analysis part is the main focus of this project, the GUI needs to provide fast results, why the application was implemented as a prototype. In this project there were also no

specified requirements of the visual results which led us to conduct the programming iterative and evolutionary.

The project manager for this thesis has acted customer for the interface functionality and with his help we choose what functionalities that should get more or less priority throughout the iterations. Information from the meetings was also used for assumptions on which the end user might be, to aid when designing the interface.

*View* components were used for browsing the log files, and *editor* components were used for the event data views since they need to take log files as input object[a].

Using the features described in the *Graphical Representation* section as criteria, we decided to use the Zest library[b] for the Node Editor, and implement the Time Chart Editor using the SWT graphics library[c]. The reason why we didn't chose a more advanced library for the time chart was because of the lack of support for chart interactions in other examined libraries. Further information about the graphical libraries that was studied can be seen in appendix section *C.1*.

### 5.3.2 User Analysis

For the users the comprehensive goal of using the Correlation Tool is to profile and correlate log files using a graphical user interface. Scenarios where the tool might be useful follow below:

- General examination and comparison of the log files
- Finding the reason of an error or a certain behaviour by the different event data relationships
- Collecting information about a certain event or process from different log files
- Getting an overview of how the product interacts with the rest of the system through visual presentation of the propagation tree and different dependencies

The user will also like to interact with the data in a way that is not possible with simple command prompt printouts. These interactions include sorting, filtering, graphical representation, comparison and correlation between log files. He might not have knowledge about the different log structures and the log content, which is why the system should be intuitive and not include acronyms or pre-knowledge assumptions.

The user might also want to adapt the interface to work with further log files, why the programming should be conducted in a way that facilitates in future extensions.

### 5.3.3 Implemented Functionality

#### *5.3.3.1 Navigating the Log Files*

The figure below shows the Directory and Logs View. Through the Directory View the user can browse to the path where the log files are stored, and by clicking the directory of choice the contents of the directory will be shown in the Logs View. From the logs view the log files can be navigated. The TSL log file can be extended to show the different event type contents. By doing this, correlation between specific event types and other event data will be enabled. Since the TSL log is not yet supported the extending information is for the moment just dummy data. By selecting one or several log files different actions will be enabled or

[a] see *Eclipse and Eclipse Plug-ins* section for information on these components

[b] Zest is a GEF based visualization toolkit for Eclipse. The primary goal is to provide easy graph based programming (www.eclipse.org/gef/zest/zest.php).

[c] SWT is an open source widget toolkit for Java designed to provide efficient, portable access to the user-interface facilities of the operating systems on which it is implemented (www.eclipse.org/swt).

disabled depending on which files and correlations that are supported by the different actions. The actions can be executed either by clicking the icons on the toolbar or through a context menu if the right mouse button is clicked.

Currently both specific and generic data models are supported for storing the log file event data. The user can select which one of these he wants to represent the event data through the context menu and the *Switch Data Model Representation* action. The actions will do the appropriate operations depending on which representation that is currently active (implemented functionality might vary slightly).

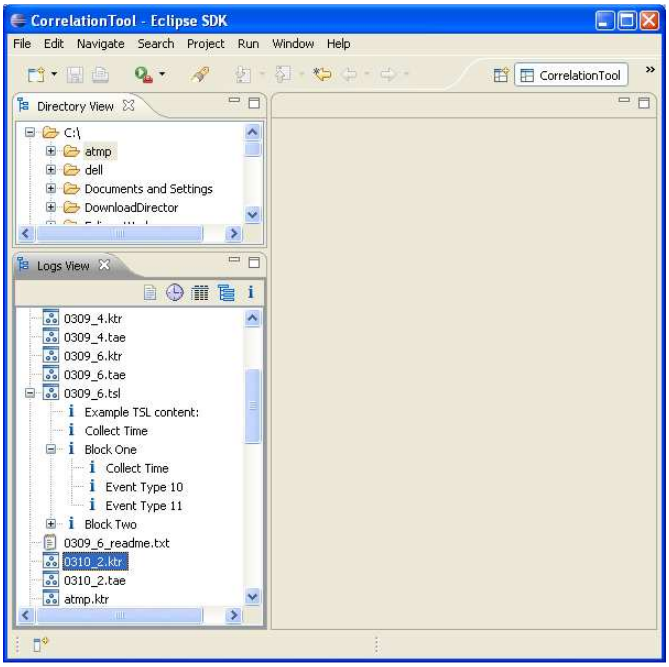



***Figure 36: The Navigation View***

### *5.3.3.2 The Table Editor*

In the Table Editor shown in Figure 37 and Figure 38 the log file entries are represented in rows and columns. In the bottom of the editor there are two tabs where the user can choose to either list all entries in the log file or list all unique component names with collected information of the components. For the moment all event data in the generic data model is supported by the entries table view, while KTR and T&E specific data models also support the component list table view. Figure 25 shows the event data of the KTR log file where each row represents one log file entry. Figure 38 contains information about the event types for which each of the component process is responsible, how many other processes the process interacts with, and the names of these processes.

Different tables can be opened for different log files so that the user can compare the different event data. Through the column that represents the normalized time stamp the event data can be synchronized to each other over time. The normalized time stamp corresponds to the time used in the Time Chart Editor[a].

The different attribute columns can be sorted depending on what the user would like to find and compare amongst the event data. If an entry is clicked, example highlighting is used for the process names to facilitate in navigating the log files. The process names in the selected entry will be highlighted in the same table and also in other opened tables of the same or of different event data types.

[a] The Time Chart Editor will be presented in the *The Time Chart* Editor section

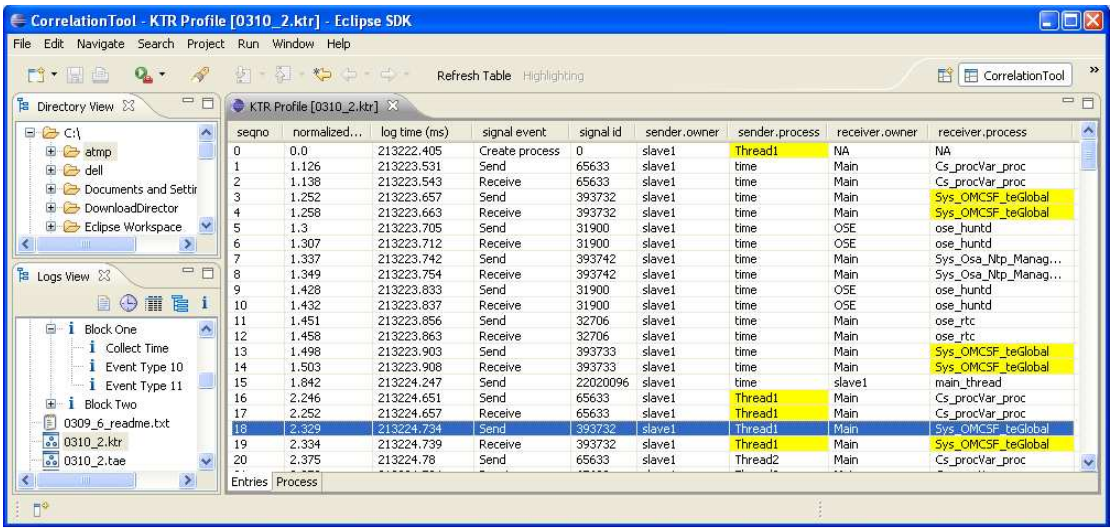



***Figure 37: The Table Editor showing the Kernel Trace event data entries***

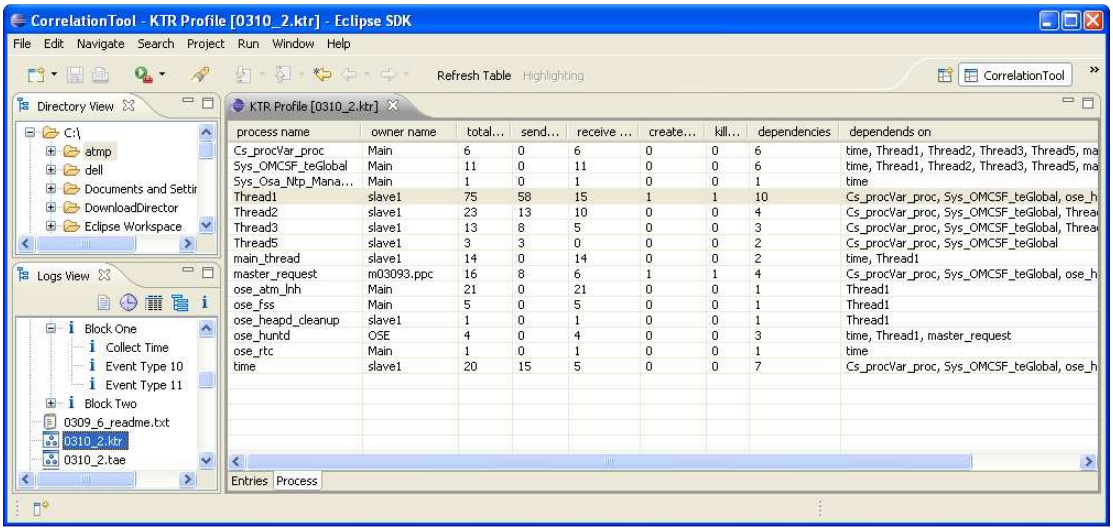



***Figure 38: The Table Editor showing the Kernel Trace event data processes***

#### *5.3.3.3 The Time Chart Editor*

In the Time Chart Editor (shown in Figure 39) the event data entries will automatically be normalized on the first event that takes place in the log file. If more than one log file is selected to be shown at the same time, it will also automatically calculate and use the biggest time span of these for the time scale. If the user needs to synchronize the different event data, he can select the log files of choice, bring up the context menu and select “Change Time Delta”. This will bring up a dialog from where the user can specify a positive or negative time delta. When the user clicks refresh from the Time Chart Refresh button, the charts will be re-drawn with respect to the new properties.

The Time Chart Editor currently supports any event data with a timestamp that is put into our generic data model. In this case it will show each of the event data entries as icons on a time line in the fashion shown in Figure 39. The different event types will be put as checkboxes to the right of the time line, from where the user can filter what information that is to be shown. The entries that contain an error will be shown in red color, and the other entries will be shown in cyan color. When the mouse pointer is moved within one of the diagrams a trace line will be shown at the position of the mouse pointer and also in the other opened diagrams. The trace line will show at what time the user currently is pointing and for all events that take place during that time, in any of the diagrams, the entry icon will show a highlighting color. If the mouse pointer is left hovering over one or multiple entries, all entries taking place at that time will be shown in a tool tip table (see Figure 39). The user can

also click on the different entry icons which will put them in an activated state (shown with golden color). If the user clicks the “open entries in external table editor” button, the activated entries will be opened in a table editor where they can be examined further.

For the CPU Load log specific features are added that allows the data to be shown using a line diagram. Since there is no parser for this log file yet, dummy data is added when the user wants to use any CPU Load log for the Time Chart Editor. Currently, only background and interact quotas will be shown, but this can easily be extended to include any kind of countable data and support for any kind of log file. An example of the line diagram together with the entries diagram is shown in Figure 40.

For both of these representations, the data that is shown in the diagram can be traversed by using the zoom function. Time is in this case specified in microseconds in the start and end text boxes followed by clicking the refresh button, which will re-draw the entries for the new specified time.

As mentioned in the *Graphical Representation* section, this event data representation might be useful when traversing and comparing events in reference to time. By finding the event that describes an occurrence in the system, other events taking place at the same time can be examined to see if one of these events might be the cause of the problem.

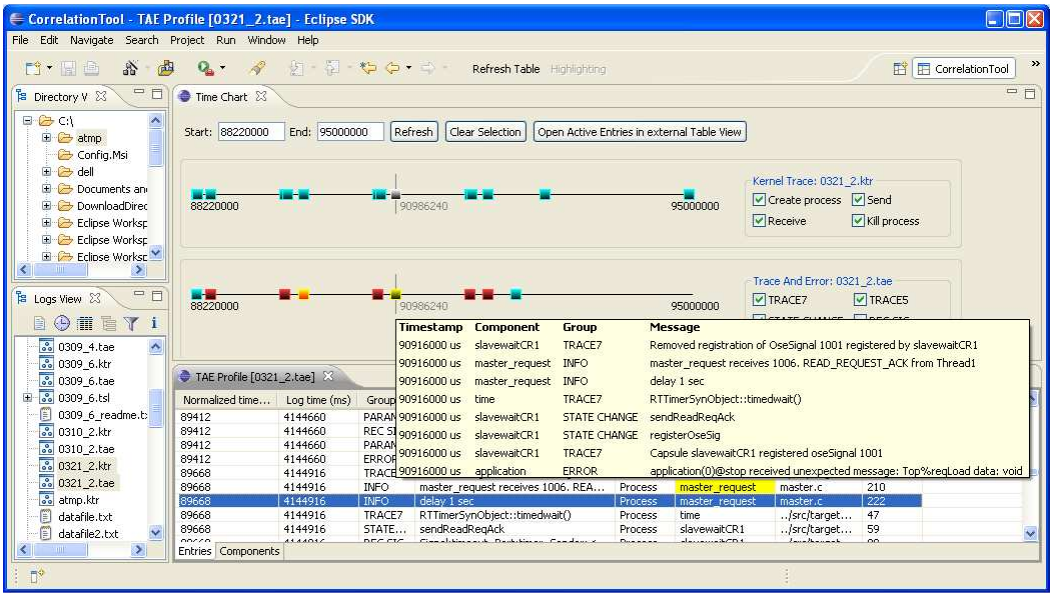



***Figure 39: The Time Chart Editor together with the Table Editor***

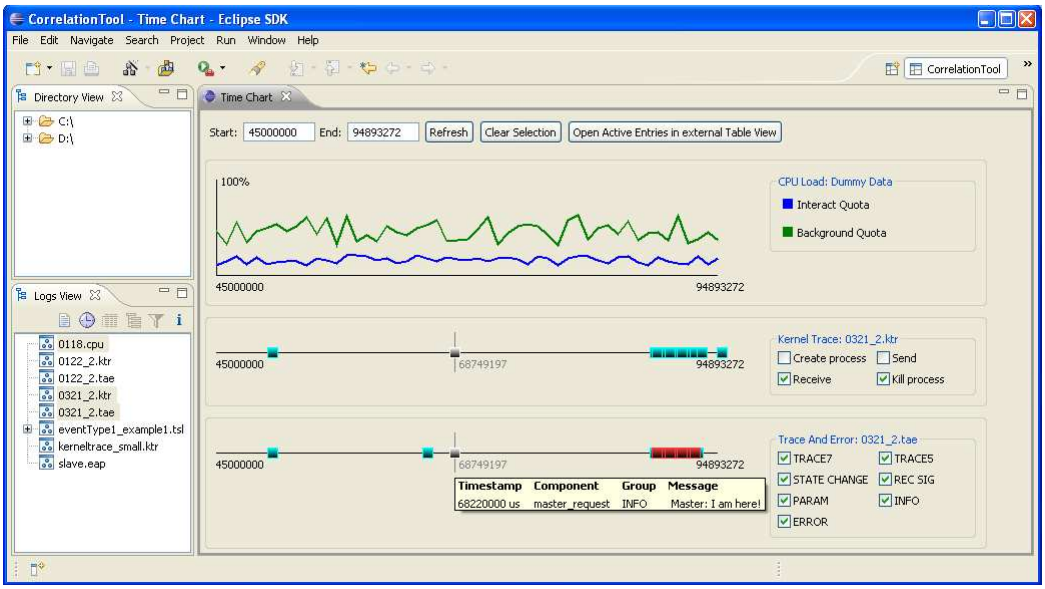



***Figure 40: The Time Chart Editor showing a line chart diagram***

#### *5.3.3.4 The Node Editor*

The Node Editor can show all event data in the generic data model that has a source and affected component (in the assigned set of log files this includes only the KTR and TSL log). All components will be drawn as node and each of the events taking place between them will be drawn as a connecting arrow as shown in Figure 41. The connecting arrows point either in one direction or in both depending on the event data, and the amount of interactions is shown in text next to the arrow. Different layouts can be selected from the Layout Algorithm drop down menu. Amongst these the *Spring* layout will try to put groups of connecting nodes separated from other groups; the *Vertical* / *Horizontal* algorithms will put the nodes in either a vertical or horizontal line; and the *Grid* layout will put the nodes with equal distance in a grid fashion.

When one or multiple components are selected these will be highlighted with the first level highlighting (yellow) and the connecting components will have a second level highlighting (orange). Also the interactions between the components can be selected. If the *Open Table Editor* button is pressed a dynamic table will be opened that filters the content depending on the node editor selection. It will show the entries that contain the selected node editor source components, together with selected component pairs if an interaction is selected.

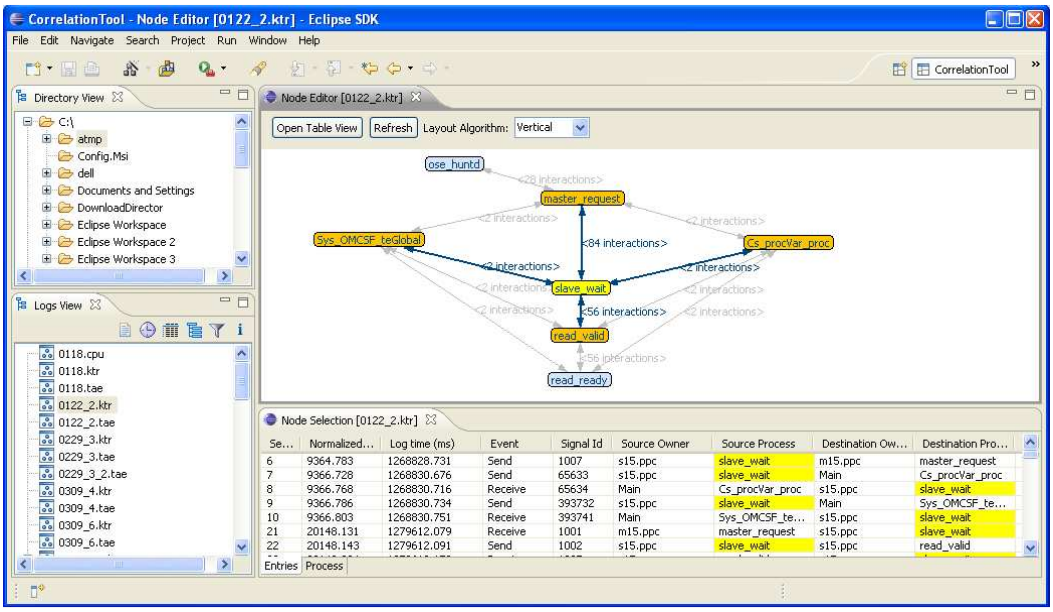



***Figure 41: The Node Editor***

#### *5.3.3.5 Interaction between Representations*

If multiple editors are opened at the same time, these will give different highlighting depending on the user selection in other editors.

When an entry is selected in any of the Table Editors the components in that entry will be highlighted in the same table and in any other table containing components with the same name. In the Time Chart Editor, the corresponding entry will be highlighted if they display event data from the same log file.

If one or multiple nodes and connections are selected in the Node Editor, the entries in the Table Editors that contain source components with the same name or corresponding interactions will be highlighted. The same entries will also automatically be activated in the Time Chart Editor.

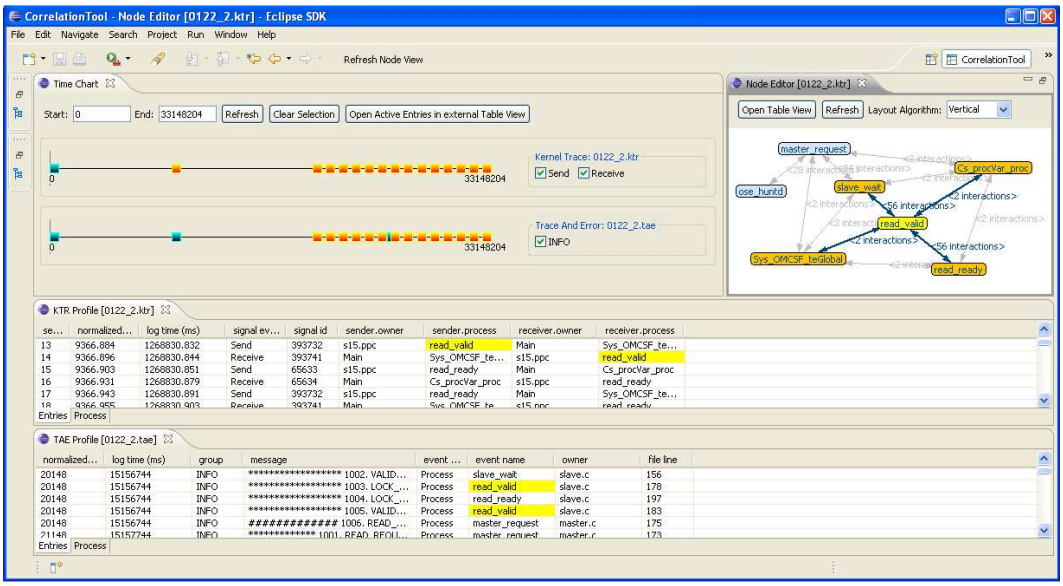



***Figure 42: Interactions between event data representations in the GUI***

#### *5.3.3.6 Other Functionalities*

Other functionalities are shown in the figure below and explained further in the following sub sections.

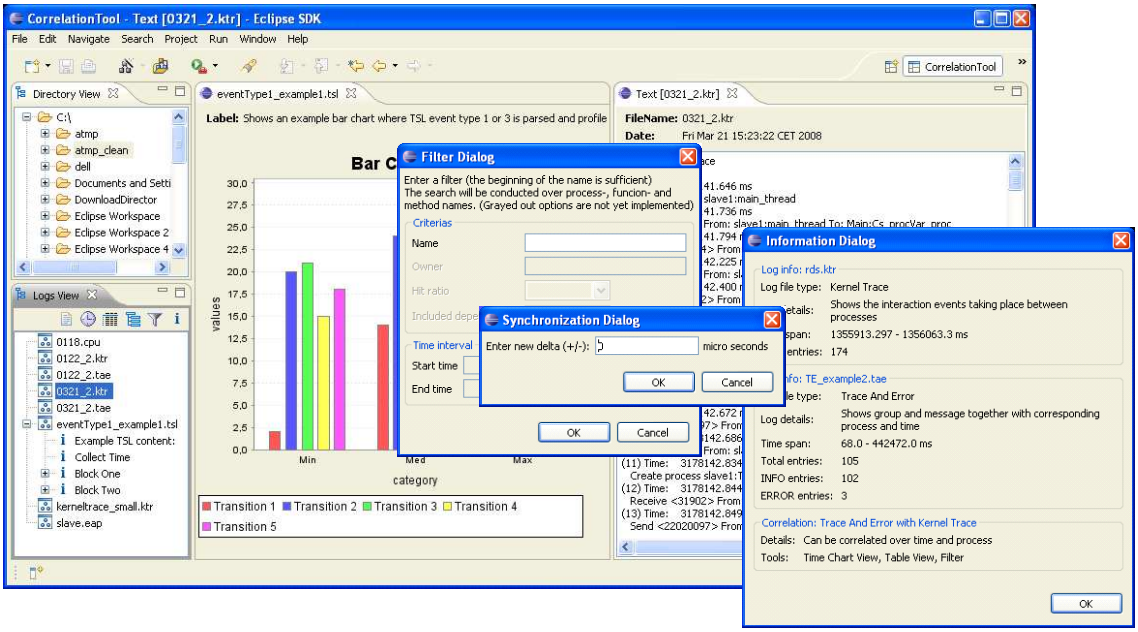



***Figure 43: Various GUI functionalities***

### *The Information Dialog*

The Information Dialog shows a short description and a summary for the user selected log files. It also gives a short description of how the different log files can be correlated and through what tools the correlation can be conducted. Typical information that might be displayed can be seen in the Information Dialog in *Figure 43*.

### *The Filter Dialog*

From the Logs View, the user can filter log file event data from a filter dialog. From here the user can specify a name pattern that will be filtered upon in the selected log files. When the user has defined the filter, the entries containing the specified criteria will be opened in separate table editors for each of the log files. The Filter Dialog also shows other useful filter criteria, but these are not yet implemented.

### *The Time Delta Dialog*

As mentioned in earlier chapters a dialog for changing log file time delta can be opened from the context menu. From here a positive or negative time delta can be set for the selected log files. The changes will be shown in Time Chart Editor and Table Editor when these are opened or refreshed. The next time the Change Time Delta dialog is opened it will display the currently stored time delta to be changed.

### *The Text Editor*

Any file in the Logs View can also be opened in a Text Editor, where the raw format of the log files or any other document can be displayed. For TSL log files with XML representation the content will be printed by the JDOM library in a structured XML format.

Through the context menu option called *Profile in Text Editor* the TSL log files with XML representation of event type 1 or 3 can be profiled in text format.

### *The TSL Bar Chart Editor*

As an early demonstration of our example TSL parser for the XML structure, a bar chart can be shown to profile the different values of TSL event type 1 and 3. The implementation is made in JFreeChart[a] and the code includes a function that converts the generated chart to an image before being shown in the editor.

[a] JFreeChart is an open-source Java chart library for creating complex charts.

## 5.4 Software Architecture

Below is a simplified diagram of the Correlation Tool that includes the GUI, the engine and external classes. In the following sections an overview of the data flow between the different groups of classes will be described.

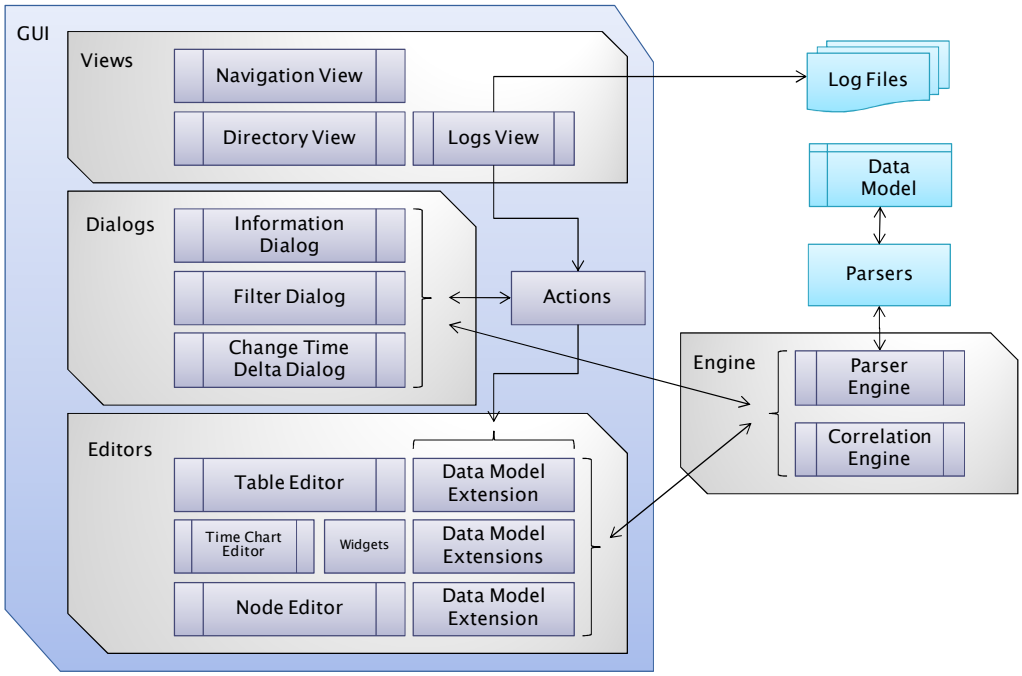



***Figure 44: Software architecture of the Correlation Tool***

Information on how the Correlation Tool can be extended with support for a new log file type can be found in appendix section *C.3*.

### 5.4.1 Views

The Navigation View shows directories and files in the system and the extending Directory View applies a filter to only show the directories.

The Logs View shows the different log files contained in the selected Directory View directory. From here different actions can be executed depending on what log files that are selected.

### 5.4.2 Actions

The existing different actions are the following:

- Change Time Delta Action
- Filter Action
- Log Information Action
- Node Action
- Switch Data Model Representation Action
- Table Action
- Time Chart Action
- TSL Bar Chart Action

When the action is executed it will send references of the input files to the specific dialog or editor class. Some of dialogs will return information that is then treated by the action. Since

the actions are abstracted from the logs view, new ones can easily be created and added by the user.

### 5.4.3 Dialogs

The different dialogs will make use of the Parser and Correlation Engine to either display or handle the event data from the input files. For the *Filter Dialog* the filter information added in the dialog will be returned to the calling action that in turn will open the corresponding Table Views to show the filtered entries. The Change Delta Dialog will return the user input and make appropriate changes to the file properties.

### 5.4.4 Editors

The Editors are created with most of the functionality independent from what data model that is used. They will however need to be extended with this information before they can show any information. These extending classes will do calls to the Parser and Correlation Engine to handle the data, and then display this information in the editor. The editors can be registered as listener and/or providers for selection changes from other editors and will change highlighting accordingly.

The Time Chart Editor consists of the Box Chart and Line Chart widgets, and in this case it is these widgets that are to be extended with data model support. Further widgets could in the future be added to show other types of time charts.

For all the Editors, the base class extensions can easily be added or removed to support other data models or other information.

### 5.4.5 Engine

The Correlation Engine handles tasks that often are used on the event data, these can be tasks such as finding entries containing certain information or comparing information from different event data.

When calling the Parser Engine the user specifies if he wants the data to be parsed to the generic or specific data model, and the engine class will depending on the input file type make a call the right parser class to handle the log file. The data model will be returned and then passed along to the class that was asking for the information.

### 5.4.6 Parsers

The parsers are independent from the rest of the system architecture. Parsers can be constructed in any way as long as it takes a file as input parameter, interprets the information and puts it in a data model that can be handled by the system.

# 6 Comparison with the TPTP Tracing and Profiling Project

## 6.1 Using the TPTP Tracing and Profiling Project

The TPTP Tracing and Profiling Project seems like a robust base from where a correlation tool can be built. It provides help classes and a generic log adapter for parsing log files to the Common Base Event model, even though this is not an all that intuitive process. When the parser is built the new log file can be used in the TPTP user interface by importing the log file to the workspace. At this moment the log file can be viewed in a table view showing the default table columns or correlated with other log files over time. The table view also includes basic functions such as filter over severity, find entries by attribute value, and sorting. If an entry is clicked further information can be shown in a property view where all the Common Base Event data is shown for that entry.

The project correlation view shows that the log entries are related to each other by drawing lines between icons that represents the entries. The entries are listed with even spaces without being related directly to the time. If the user wants to enable further correlations he has to extend the TPTP correlation engine to describe how the log file event data can be associated. From the user interface the same correlation view is used, but lines are now drawn depending on the new criteria. The user can also do rule based analysis on the log files by extending the TPTP analysis engine and specify a symptoms catalogue.

## 6.2 Advantages using the TPTP Tracing and Profiling Project

The advantage with TPTP includes a very nice integration with the Eclipse framework and the abstraction of parsers, correlations and analysis from the main code. The symptoms catalogue function is a useful tool and was never implemented in our own project. Otherwise most functions that the TPTP Tracing and Profiling Project tool are also implemented the Correlation Tool.

Another advantage of TPTP is the Generic Log Adapter which can provides two alternative ways for the programmers when they are developing parsers for their log files: one is the rule-based adapter and the other is the static adapter. With rule-based adapter, programmers do not have to do any programming work to design a parser; they just need to create an adapter configuration file and then set some rules about the structure and format of the log files need to be parsed. The rule-based adapter can usually be used for application log files that have a fixed and simple log record format. A static adapter uses a java class to parse a log file. In this way, programmers need to do some java programming to read the original log files and then store them in Common Base Event. This type of adapter can be used for some complex log files that are difficult to summarize by rules.

## 6.3 Advantages using the Correlation Tool

The Correlation Tool has many general and log specific functionality that is not included in the TPTP Tracing and Profiling Project basic functionality. These include support for opening several table views with selection listening and highlighting to facilitate comparison and manual correlation; a process summary table extension for Kernel Trace and Trace and Error Event Data; a time chart view in relation to the actual time with event type filtering; and the node view to track process dependencies.

# 7 Future Work

In this section suggestions will be provided on future work that can be done on this project. Areas that will be handled are future correlation analysis, implementation, and log file structure. For the log file structure discussion and evaluation of the log files will be provided together with suggestions on how these can be improved to better support correlation with other log files.

## 7.1 Correlation Analysis

CPP system information is recorded in many different log files from where the assigned set of log files for this project is just a few. By doing further analysis on other log files it will be possible to provide further information to complement the analysis handled in this thesis as well as providing information between the new log files. In a future implementation, support for further log files would also give more detailed information and a more complete picture of the system which would facilitate when tracing errors or system behavior.

Correlation of log files between different boards should also be analyzed to see if useful information can be extracted and how this information can be presented. This could be done by designing two simple applications that communicates with each other while running on different boards. The Test Base Application can be improved to support this function by adding code for managing the board information and identity.

Correlation beyond the observer level would benefit in automated functions to give warnings if something is out of the ordinary. One example is the use of rule sets that can define what a process should and shouldn't do, or how the events should occur in time and in relation to each other.

## 7.2 Implementation

Since the Correlation Tool provided by this project was built for getting fast results, the base code is not robust enough to be used for a finished product. The purpose for the tool should instead be to add extensions for further analysis of event data, correlations, graphical representation and user interaction. If a new product is to be developed in the future, we recommend using the TPTP trace and profile framework. This framework seems to have a robust implementation of basic functionalities that could be a good base for extending with the functionalities suggested in this project together with other Ericsson log file specific functionalities.

### 7.2.1 Extension of the Correlation Tool

#### *7.2.1.1 Parsers and Data Model*

The generic data model provided with this project work should be sufficient for simple log files and could in other cases easily be extended with further attributes; but for a generic data model that hopefully can be used for any kind of log file we recommend changing the present data model to the Common Base Event. This requires some studies in how Common Base Event works, but after that it should be simple code substitution where information is retrieved in the data model extension classes.

In order to extend the Correlation Tool with a new log file type, a new parser has to be developed. If the log file is easy to summarize with rules, an adapter can be used to aid in this process. By using the adapter the developer can define the parser by specifying rules instead of doing actual programming and as a result they do not have to understand the structure of the data model.

As mentioned earlier, the T&E log file currently stores structured messages in some of the defined trace groups. If the parsers are extended to handle this information before storing to the data model, further information and correlations can be provided by the Correlation Tool.

#### *7.2.1.2 Dynamically Adding Log Files*

Yet another advantage that we found when using a generic data model is that it would be much easier to implement functionality for dynamically adding support of new log file types. If this feature is implemented the user would be able to dynamically tell the system about a new parser and for what log file type it should be assigned, instead of having to make changes directly in the code.

#### *7.2.1.3 Support for the TSL log file*

As mentioned in the *Correlation Analysis* section, it would be very interesting to correlate the TSL log with the KTR and T&E logs, since this would bridge the gap between product debugging and system problem determination. This project presents several different ways in how this can be done, but as of now there is no parser or implementation to verify the results in a practical context. It is of course possible to create a text parser for the original TSL log file, but the structure of the log file is complex and is not designed for being parsed. If a XML representation of the log file is provided the parser could be better structured and more reliable. Since the TSL log file has a more complex structure which might be hard to represent with a generic data model we suggest that the specific data model is used also for future work.

The TSL Event Type 8 (RTMutex contention count) was never analyzed in this project, for further implementation, it is necessary to do some investigations and analysis about the RTMutex contention.

#### *7.2.1.4 The Editors*

The editors could benefit from being further integrated with the Eclipse framework. This includes integrations such as Eclipse rulers that can show where errors and warnings take place in comprehensive information and properties dialog to show further information about the log files. Each of the editors should also include more advanced features for searching and filtering. It would be a great advantage for the user if he could filter on any attribute to reduce information to contain only what is interesting. How each specific editor can be further developed follows in the sections below.

Information about graphical libraries that can be used for future work can be found in appendix section *C.1*.

***The Table Editor***

Some event data information might be too comprehensive to show in a table. An example of this is the message attribute in the T&E log that sometimes can be too long for a good overview in the table editor. A solution could be to only show basic information in the table editor and make use of a dialog or another view to show the more detailed information. The more detailed information should in this case be connected to the table in some way, for example by selecting or double clicking an entry. It would also be good if the user could export the information in the table editor as a spread sheet. This would allow the user to convert the data into any type of diagram or report.

The Table Editor could also benefit from having a fully implemented highlighting functionality. For the moment example highlighting is supported for component names, but this could be extended to any of the attributes in the generic data model. The user should also have the option to select what columns that are shown, since big log file structures otherwise wouldn't give any good overview of the data.

***The Time Chart Editor***
For better user interaction the Time Chart Editor could benefit from interactive zoom and time delta functionality. Here the user could click, drag or scroll when zooming or setting the time delta functionality. If the user still wants to include a zoom function using values, the possibility to specify time in other units than micro seconds should be included.

For the moment the Time Chart Editor is built using raw SWT graphics. If it was to use SWT components instead this would enable basic SWT functionalities such as component based tooltips, selection providers, drag and drop, etc.

Selection providers should also be implemented so that selection of the entries can be reflected in the information shown in other editors.

***The Node Editor***
The current implementation uses the default plot algorithms given by the Zest library. If there are many components to plot, the plots can sometimes be too messy for giving a good overview. To solve this, additional algorithms can be implemented to arrange the nodes in a better way. There are also other libraries that are specialized in similar areas that might be better suited.

If the TSL event data is to be supported, the Node Editor could be extended for traversing between the KTR component representations of OSE signal propagation down to the lower levels of the TSL transition representations (as suggested in the *Graphical Representation* section). In this case the parent component can be used as a filter to show a limited amount of child components. If the developer decides to do the implementation in another or additional way it is still essential that a filter is used since the information in lower levels otherwise would be too comprehensive. This includes the current implementation of KTR processes that in some cases will plot too many processes for giving a good overview.

## 7.3 Log File Evaluation and Suggestions

If an implementation is to be made in the future it would be beneficial if present log files are adapted-, and new log files are designed with this in consideration. The following sections will handle evaluation of the log files that was used in this project and suggestions for a good log file structure. References in these sections are used for the different formats, while the criteria and conclusions are based our experiences throughout this project.

### 7.3.1 Log File Structures

A few criteria for a good log file strategy are collected in the table below

| Criterion | Situation |
|---|---|
| Readability | If the log file is to be read directly without any application to first handle the data, the structure should be easy to read for the user. Since the log file also might be interesting for a user not familiar with these particular attributes it is also good if abbreviations and acronyms are avoided. Everything in the log file shouldn't always be explained since this could inflict with the readability, but in these situations explanations should instead be provided in an external document (to avoid the necessity of experts). |
| Easy to implement | The structure of the log file should not be a tedious task for the programmer to achieve. |
| Easy to parse | If the log file is to be interpreted by a parser, the log file structure should be adapted to facilitate in this task. |

***Table 4: Log file criteria***

For the set of log files that was used in this project an observation that applies for all log files is that readability is the property that is put first. The event data representations use some acronyms, but this is still acceptable since an external document with log structure explanations often can be found. Since all of the logs that were given for the project are represented by pure text output the implementation part is easy to achieve and is no issue in these cases. For the easy to parse criterion, the result varies depending on the log file. Using text output requires the use of string operations and regular expressions, but can in some cases be facilitated by making use of external log parsing applications. For most of the logs the output is strongly structured in a way that doesn't make this task too hard. The exception is the TSL log that has a large content and that wasn't designed for text parsing.

### 7.3.2 Log File Formats

Four different formats were studied to find out which one would be best suited for the different types of log files. These file formats are Text format, XML format, YAML format and Common Base Event. Example printouts using these formats are shown in appendix section *D.5*. Below is a table showing advantages and disadvantages for each of the formats.

| Format | Advantages | Disadvantages |
|---|---|---|
| Text Format | It is fast to implement and gives readable log file output. | It is not the best format for parsing, but if the log file is not to complex and is strongly formatted, a parser could be created using regular expressions |
| XML Format[a] | This format is good when being interpreted by applications or parsers, since libraries for XML operations commonly are provided for most programming languages. | It is inadequate for being read in raw output. |
| YAML[b] | Good for being interpreted with applications or parsers and at the same time it gives readable raw log file output. | Lacks in support for real time reading. If the user that is printing the log file is not familiar with the structural rules he has to first learn about these. This is not a common standard for log files, why other situations might occur where YAML is not supported. |
| Common Base Event | It is easy to migrate between applications if Common Base Event is used.<br><br>Implementation classes are provided for Java, to facilitate in the log file output, but the format can also be used in any other programming languages. | The Common Base Event is represented by XML format, but is even less readable than a user defined XML structure. |

***Table 5: Log file formats, advantages and disadvantages***

We believe that the best way for now is to use XML for log files that are not meant to be read manually, and use a strongly formatted text format where the log file needs to be read

[a] The Extensible Markup Language (XML) is a general-purpose extensible markup language
[b] YAML is a human-readable data serialization format that takes concepts languages such as XML, C, Python and Perl

from raw output. From these formats the log files can either be parsed to a user defined data model or to a Common Base Event structure when being used by the application.

### 7.3.3 Log File Content

#### *7.3.3.1 Same format of data*

If the log file attributes have the same format for similar data unnecessary calculations can be avoided. This is always convenient, but would be more valuable at real-time monitoring since calculation time is more important in this case.

#### *7.3.3.2 Acronyms, abbreviations and special definitions*

Even if the log files have a good structure and are readable for the immediate user, it might still not be intuitive for other users. This might occur when lots of acronyms, abbreviations or special definitions are used which implicates the need of experts and communication overhead when working with the log files. These should therefore only be used if it helps the readability of the log file; and when they are used, it would be good with a reference to an external document explaining the definitions.

#### *7.3.3.3 Log file specific comments*

***Kernel Trace Log***

The KTR log is a bit ambiguous with the “from:” and “to:" fields in the log events. When the event action is a send action the process specified in the “from:” clause it the source process (the process that is responsible for the action) while if the event action is a receive action the source process will be the process in the “to:” clause.

***Trace and Error Log***

The total memory that could be used for Trace and Error log is not very large, if the memory size was extended it would be possible to store and examine more entries. The accuracy of the time cannot distinguish the time when a signal was sent out and when it was received. It is better if the accuracy could achieve microseconds.

In Trace and Error log, the OSE signal number could be represented both in decimal or hexadecimal, while in KTR log, it uses decimal to represent a signal. If both of the log files represents the signals in the same format it would be easier to compare these when read from the original output.

***Execution Address Profile Log***

For the moment the EAP log contains further information of the TSL transitions, but without the possibility to correlate these two event data. This could be possible either if a naming convention is introduced containing the capsule source and destination states in the TSL log file together with actor name, or if further data is introduced in these or through other log files.

***CPU Load Log***

As mentioned in *The Assigned Set of Log Files* section, there are four types of measurements that can be shown with the CPU Load log. Here only the CPU peak load log contains timestamps, while the others contain the integration interval of how long it takes to measure the CPU utilization. For further correlation between the CPU Load log and other log files it would be valuable if timestamps were added for the start and stop time of these measurements.

The accuracy of CPU peak load is 0.01%, while for other CPU Load logs measured for user specified objects such as process name, process priority or process type the accuracy is 1%. Since many of the specified objects might have less than 2% or 1% but more than 0% CPU load, it would be interesting and useful to have a better accuracy also here.

***TSL Log***

In Block 1, Event Type 11 the present log file gives the label *OSE intra* where it should be *UML intra*. The *intra* and *inter* queues and signals can also easily be misread and we feel it would be better to refer to these as *internal* and *external*.

This log file is also very comprehensive which makes it hard to compare event types between different controllers and blocks. For this reason it would be much easier to traverse the information with the help of an external application. To do this it is however required that the log file is represented in an XML log file to facilitate in parsing.

# 8 Concluding Discussion

## 8.1 Summary of the Results

Correlation research for retrospective log analysis at the Ericsson CPP system hasn't been done before. This project is a first approach to find out what correlations that is possible and if a tool for interacting with the event data and analyzing correlations is of interest. In this project we have shown that correlations are possible, and we have analyzed the different possibilities for which of these that might give interesting and valuable information.

The correlation possibility analysis was based on log files collected from the Test Base Application. By designing this application we could collect the required log files and get a better understanding of them and their relations. During correlation analysis we found some weaknesses of the log files including time accuracy and the log structure. We have mentioned these together with some suggestions for improvements in the *Future Work* section.

Further research contributions include parsing of the log files, data model approaches for the event data, and graphical representation of event data and event data correlations. For the TSL log file we have provided a suggested XML structure and a more extensive graphical presentation analysis.

The implemented correlation tool provides data models and parsers for the different log files and a graphical user interface through where the user can profile and correlate the event data. Various concepts of presenting the event data and correlations have been verified through the interface and further suggestions have been provided for different functionalities that might be of interest. The Correlation Tool is also a finished prototype that due to de-coherence easily can be extended with further parsers, data models, editors and views.

In a late part of the thesis we found out about TPTP Trace and Profiling Framework that we consider to be very relevant to this project. This application was evaluated and compared with our own achieved results, and based on these we came up with suggestions in the *Future Work* section. In the same section we will also give suggestions on how the project can be developed further which includes a sub section where we evaluated and gave suggestions on the log file structures.

## 8.2 Complications

In this project we really wanted to do an implementation to verify the TSL correlations since these were some of our more important results. We came with the conclusion that the log file presentation wasn't suitable to be parsed in the present format why we designed and suggested an XML representation instead. When we presented these results at a technical presentation at Ericsson it was said that the representation would be implemented for us to use, why we chose to wait with this part of the project. When we realized that the representation wasn't going to be implemented in time it was too late for us to come up with any optional implementation. We do however consider the analysis results to be our main results event though they were not verified in a practical context. We also decided to compensate with further graphical analysis and provided diagrams on how an interface implementation might look like.

When we found out about the TPTP trace and profiling project and Common Base Event at a late stage in the project, we first did not know what to do with this information. It did however turn out to be a good complement to the report since it allowed us to evaluate our own results better in a comparison and also to give better suggestions in the *Future Work* section.

## 8.3 Alternative solutions

### 8.3.1 Pre-processing of Event Data

If the event data is pre-processed before the information is stored in a data model the data content can be reduced and a first detection of interesting data can be done at an early stage.

A filtering function at this stage might be useful if the data is comprehensive or if there is need to save bandwidth, to limit how much data that is stored or handled at one time. Often this is used when the purpose of the log analysis is clear, such as if intrusion attempts in a network system need to be spotted. In these situations entries that are not relevant might be filtered without affecting the analysis [27]. In our own case the purpose of our log analysis in not clear, the data that is to be used is not very comprehensive, and there is also no need to save bandwidth, why this function as far as we know is redundant.

A detection function might be good to find anomalies in the log file and mark this in the data model so that it is later easier to spot or compare later in an application. This function is more useful when there is a rule set so that one knows that something might be strange, and is probably more useful in a real time system when anomalies should be detected at an early stage. It might however be a good idea in the case of this project to set a severity value for each of the entries, so that they later can be sorted and compared over severity [24]. This can be set either if a pattern in the log entry is known, or by the log entry type. The severity value is used by the Common Base Event that was discussed earlier.

Other common pre-processing functions include grouping of events of the same type taking place at the same time [28]. This could also be done in our case, but since the data that we have been working with is not very comprehensive there wouldn't be much to gain by doing this.

### 8.3.2 Data Model vs. Data Base

We chose to store the event data in a class oriented data model since the time limit didn't allow us to get familiar with or analyze a data base approach. Advantages of using a data base could be already defined functions for comparing, filtering and retrieving information. It would also support larger data sets in comparison with the present data model and it includes predefined functionality for saving the content information.

# 9 References


[1] *Embedded System Design: A Unified Hardware/Software Introduction*
Frank Vahid, Tony Givargis; Wiley I.S.ed edition; October 2001

[2] *Designing Embedded Hardware, Second Edition*
John Catsoulis; O'Reilly Media Inc; May 2005

[3] *ThreadX product homepage*
http://www.rtos.com/page/product.php?id=1
Last viewed 28th of February 2008

[4] *LynxOS RTOS product homepage*
http://www.lynuxworks.com/rtos/rtos.php
Last viewed 10th of Mars 2008

[5] *OSE: Real-Time Operating System and Embedded Development*
Enea, Product brochure, http://www.enea.com

[6] *High Speed and Robust Event Correlation*
Yechiam Yemini, David Ohsie; IEEE Communications Magazine 34(5), pp. 82-90; May 1996

[7] *Event Pattern Detection for Embedded Systems*
Jan Carlson; Mälsardalen University Press Dissertations, No. 44; 2007

[8] *Actions and Events in Interval Temporal Logic*
James F. Allen, George Ferguson; The University of Rochester, New York, Technical Report 521; July 1994

[9] *Tools and Techniques for Event Log Analysis*
Risto Vaarandi; Tallin University of Technology, Doctoral Dissertation; June 2005

[10] *Security Warrior*
Cyrus Peikari, Anton Chuvakin; O'Reilly Media Inc; February 2004

[11] *CPP Survey*
Ericsson Internal Documentation

[12] *Remote Debug Support*
Ericsson Internal Documentation

[13] *Design Rules for Trace and Error Users*
Ericsson Internal Documentation

[14] *Profiling*
Ericsson Internal Documentation

[15] *Execution Address Profiler User Guide*
Ericsson Internal Documentation

[16] *ClearCase*
Ericsson Internal Documentation

[17] *The Eclipse home page*
http://www.eclipse.org
Last viewed 28th of Mars 2008

[18] *The Eclipse help system*
http://help.eclipse.org

Last viewed 28th of Mars 2008

[19] *ruleCore Complex Event Processing Server, product homepage*
www.rulecore.com
Last viewed 21th of April 2008

[20] *LogSurfer product homepage*
http://www.crypt.gen.nz/logsurfer/
Last viewed 23th of April 2008

[21] *SEC – Simple Event Correlator, product homepage*
http://www.estpak.ee/~risto/sec/
Last viewed 23th of April 2008

[22] *EvenLog Analyzer product homepage*
http://manageengine.adventnet.com/products/eventlog/index.html
Last viewed 19th of April 2008

[23] *TPTP – Test and Performance Tools Platform, project homepage*
http://www.eclipse.org/tptp/index.php
Last viewed 5th of May 2008

[24] *Canonical Situation Data Format: The Common Base Event V.1.0.1*
David Ogle, Heather Kreger, Abdi Salahshour, Jason Cornpropst, Eric Babadie, Mandy Chessell, Bill Horn, John Gerken, James Schoech, Mike Wamboldt; Version specifications, published 2004

[25] *Users Guide for RoseRT Target Service Libraries*
Ericsson Internal Documentation

[26] *Designing Interactive Systems*
David Benyon, Phil Turner, Susan Turner; Addison Wesley; March 2005

[27] *The Intelligent IDS: Next Generation Network Intrusion management Revealed*
Andree Yee; NFR Security Inc; July 2003

[28] *Aggregation and Correlation of Intrusion-Detection Alerts*
Hervé Debar, Andreas Wespi; Lecture Notes In Computer Science, Vol. 2212, pp. 85-103; August 2001

# 10 Terminology

## 10.1 Abbreviations

| Abbreviation | Description |
|---|---|
| API | Application Programming interface |
| AWT | Abstract Window Toolkit |
| BP | Board Processor |
| CBE | Common Base Event |
| CPP | Connectivity Packet Platform |
| DTE | Development and Troubleshooting Environment |
| EAP | Execution Address Profiler (log file type) |
| GUI | Graphical User Interface |
| KTR | Kernel Trace (log file type) |
| LM | Load Module |
| MP | Main Processor |
| MPC | Main Processor Cluster |
| MSP | Media Stream Processor |
| MVC | Model-View-Controller |
| OSE | Operating System Enea |
| RDS | Remote Debug Support |
| RoseRT | Rational Rose Real Time |
| RTOS | Real Time Operating System |
| SP | Special purpose Processor |
| T&E | Trace and Error (log file type) |
| TSL | Target Service Library (log file type) |
| SWT | Standard Widget Toolkit |
| UI | User Interface |
| UML | Unified Modeling Language |
| XML | Extensible Markup Language |

## 10.2 Definitions

| Term | Definition |
|---|---|
| Abstract Window Toolkit (AWT) | AWT is Java's original platform-independent windowing, graphics, and user-interface widget toolkit. |
| Actor | A RoseRT actor is an instance of RoseRT capsules. Multiple actors can execute at the same time. |
| Capsule | A RoseRT capsule contains states and transitions and provides coordinate behavior for the system. These can be instantiated as actors. (see *Rational Rose Real Time*, section 2.5.1) |
| Common Base Event | Common Base Event is an IBM purposed standard for events in various applications (see *Common Base Event*, section 3.2) |
| Evolutionary Development | Development where the specifications are defined throughout the development itself. It usually follows the following four steps: idea, specification, implementation, evaluation and give a new specification |
| JDOM library | JDOM is a Java-based solution for accessing, manipulating, and outputting XML data from Java code. More information about the JDOM library can be found at http://*www.jdom.org*. |

| | |
|---|---|
| Load Module | The load module corresponds to a program that is built for the OSE operating system. When loaded to the system the program will be executed as one or multiple processes. |
| MVC architecture | MVS is short for Model-View-Controller and the MVC architecture is divided into these three parts. The *Model* represents the data and information of the application and the rules used to manipulate the data; the *View* corresponds to elements of the user interface such as text, checkbox items, and so forth; and the *Controller* manages details involving the communication to the model of user actions such as keystrokes and mouse movements. |
| Remote Debug Support (RDS) | Remote Debug Support is a system level debugger for the CPP node (see *Remote Debug Support* 2.4.1). |
| Swing | Swing is a widget toolkit for Java. It provides a native look and feel that emulates the look and feel of several platforms and it also supports changing the look and feel during runtime. |
| SWT | The Standard Widget Toolkit (SWT), is a set of Java class libraries created to provide platform native user interfaces (see *Standard Widget Toolkit*, section 2.6.2) |
| XML | The Extensible Markup Language (XML) is a general-purpose extensible markup language. More extensive information can be found at *http://www.w3.org/XML/*. |

# Appendix A: Individual Thesis Contributions

Details about each of the individual contributions of this thesis can be seen in the table below. Note that the project contributions not always correspond to the contributions in the report.

| Chapter | Tobias Contributions | Xingya's Contributions |
|---|---|---|
| 1 Introduction | Report: 1.1, 1.2, 1.3, 1.4, 1.5 | Report: 1.1 |
| 2 Background | Helped in gathering information for all of the theoretical background<br>Report: 2.6 | Helped in gathering information for all of the theoretical background<br>Report: 2.1, 2.2, 2.3, 2.4, 2.3 |
| 3 Related Work | Gathered information about Common Base Event and TPTP, and helped in all of the other parts<br>Report: 3.1.5, 3.1.6; 3.2 | Helped in all the parts except for Common Base Event and TPTP<br>Report: 3.1.1, 3.1.2, 3.1.3, 3.1.4, 3.1.6 |
| 4 Event Data Analysis | - Analyzed all the different log files for data that can be extracted and correlation possibilities<br>- Analyzed graphical representation for all the log files except the TSL log<br>Report: 4.1, 4.1.1, 4.1.2; 4.3, 4.3.1, 4.3.2.1, 4.3.2.2, 4.3.2.3, 4.3.2.4, 4.3.2.6, 4.3.3; 4.4, 4.4.1, 4.4.2, 4.4.3, 4.4.4, 4.4.5 | - Analyzed all the different log files for data that can be extracted and correlation possibilities<br>- Created the Test Base Application and collected the log files with appropriate tools<br>- Analyzed graphical representation for all the log files including the TSL log<br>Report: 4.1.3, 4.1.4, 4.1.5; 4.2; 4.3.1, 4.3.2.5; 4.4.6 |
| 5 Implementation of the Correlation Tool | - Interpreters for Kernel Trace and Trace and Error log<br>- XML log representation for the TSL log<br>- Designed the generic data model<br>- Helped to designed the data model for the TSL log<br>- Analysis results regarding different graphical libraries to use with the GUI<br>- Implemented the Correlation Tool application including the graphical user interface<br>Report: 5.1, 5.2, 5.3, 5.4 | - Interpreter for the Execution Address Profiler log<br>- Helped to design the data model for the TSL log<br>- |
| 6 Comparison with the TPTP Tracing and Profiling Project | Compared the TPTP Tracing and Profiling Project with the Correlation Tool provided by this project<br>Report: 6.1, 6.2, 6.3 | Tried out the Generic Log Adapter included in the TPTP Tracing and Profiling Project<br>Report: 6.2 |
| 7 Future Work | - Wrote about suggested future work in the area of the project, including correlation analysis, implementation and log file structures<br>- Evaluated the log file structures<br>Report: 7.1, 7.2, 7.2.1.1, 7.2.1.2, 7.2.1.4 , 7.3, 7.3.1, 7.3.2, 7.3.3.1, 7.3.3.2, 7.3.3.3 | - Wrote about future work for the Test Base Application, support for the TSL log file and about extending with a generic log adapter.<br>- Evaluated the T&E, CPU Load and TSL specific log file structures<br>Report: 7.1, 7.2.1.1, 7.2.1.3, 7.3.3.3 |
| 8 Concluding Discussion | Wrote summary of the results, complications in the project and alternative solutions<br>Report: 8.1, 8.2, 8.3 | Wrote parts in the *Summary of the Results* section<br>Report: 8.1 |
| 9 References | Collected and read references from all areas in the project<br>Report: 9 | Collected and read references from all areas in the project<br>Report: 9 |
| 10 Terminology | Report: 10.1, 10.2 | Report: 10.1 |
| Appendix A: Individual Thesis Contributions | Report: Appendix A | - |
| Appendix B: Data Models | Report: B.1, B.2, B.3 | Report: B.3 |
| Appendix C: The Correlation Tool | Report: C.1, C.2**Error! Reference source not found.**, C.3 | - |
| Appendix D: Log Files | Report: D.2, D.4, D.5 | Report: D.1, D.2, D.3 |

***Table 6: Individual thesis contributions***

# Appendix B: Data Models

## B.1 Specific Data Models for Kernel Trace, Trace and Error and CPU Load Event Data

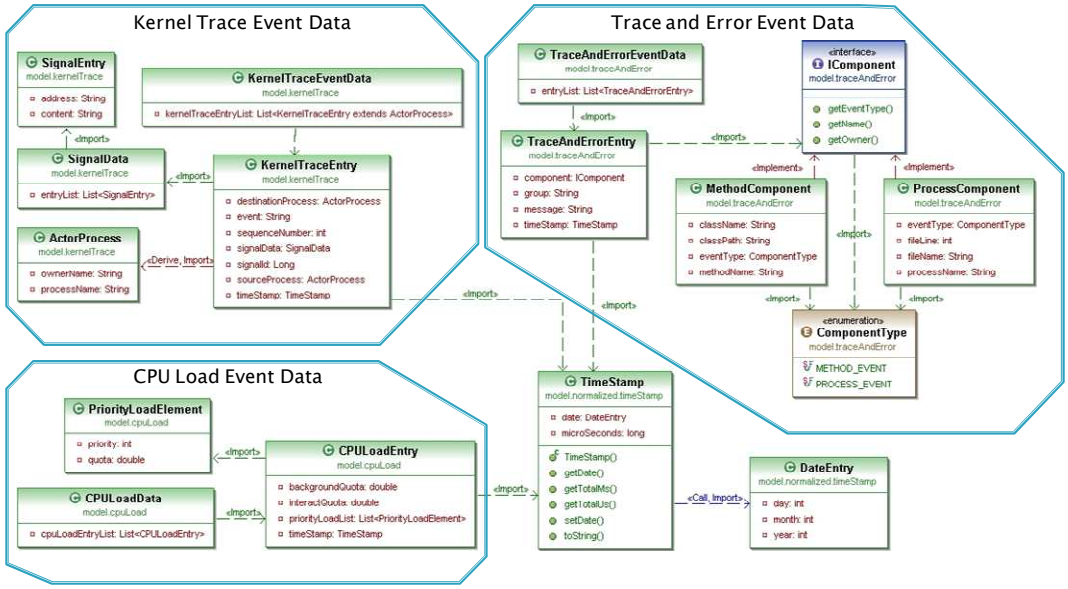



***Figure 45: Kernel Trace, Trace and Error and CPU Load data models***

The data model in Figure 45 represents the KTR, T&E and the CPU Load Event Data. All of them include a main event data class with arrays of the entries. The KTR data model represents the source and destination process with an *ActorProcess* class and the signal data is stored in the *SignalData* class. The T&E data model has two different representations for the component classes depending on if the component is a method or process. The CPU Load data model has a defined class to store the priority load element. The *TimeStamp* class that all of the separate models have in common stores the time in the *DateEntry* class and in microseconds for hour units and below. The full date and time can be collected through the *toString()* function.

## B.2 Decision points, Specific vs. Generic Data Model

| Decision points | Specific Data Models | Generic Data Model |
|---|---|---|
| Information retrieval | With this strategy it is possible to specify the exact attributes by name and where to find them in the data model. The developer doesn't have to know specifics about the log file to use the data model. | When using this approach the attribute names will have to be at a generic level. |
| Implementation difficulties | The implementation design can follow a logical pattern given by the log file structure and would be a straight forward process.<br><br>When implementing the data model in an application it has to have support for all the different specific data models.<br><br>The abstraction is not very good, and the developer has to change already written code to add support for new log data. | It could sometimes be hard to create a generic design if the log files have big variation in structure.<br><br>When developing an application the operations that can be conducted on all the different kinds of log files only have to be written once.<br><br>This approach will contribute in abstracting the code and facilitate adapting new log and trace data. |
| Extension possibilities | When adding support for further log files the user is required to program a parser, create a new data model, and extend the existing classes to support the new data model. | A new parser is all that is needed when extending with support for a new log file type with basic features.<br><br>The application still needs to be extended to use the more specific information in the log files, but this can be done in a higher abstraction layer than when using specified data models. |

***Table 7: Decision points, specific vs. generic data model***

## B.3 Common Base Event Class Hierarchy

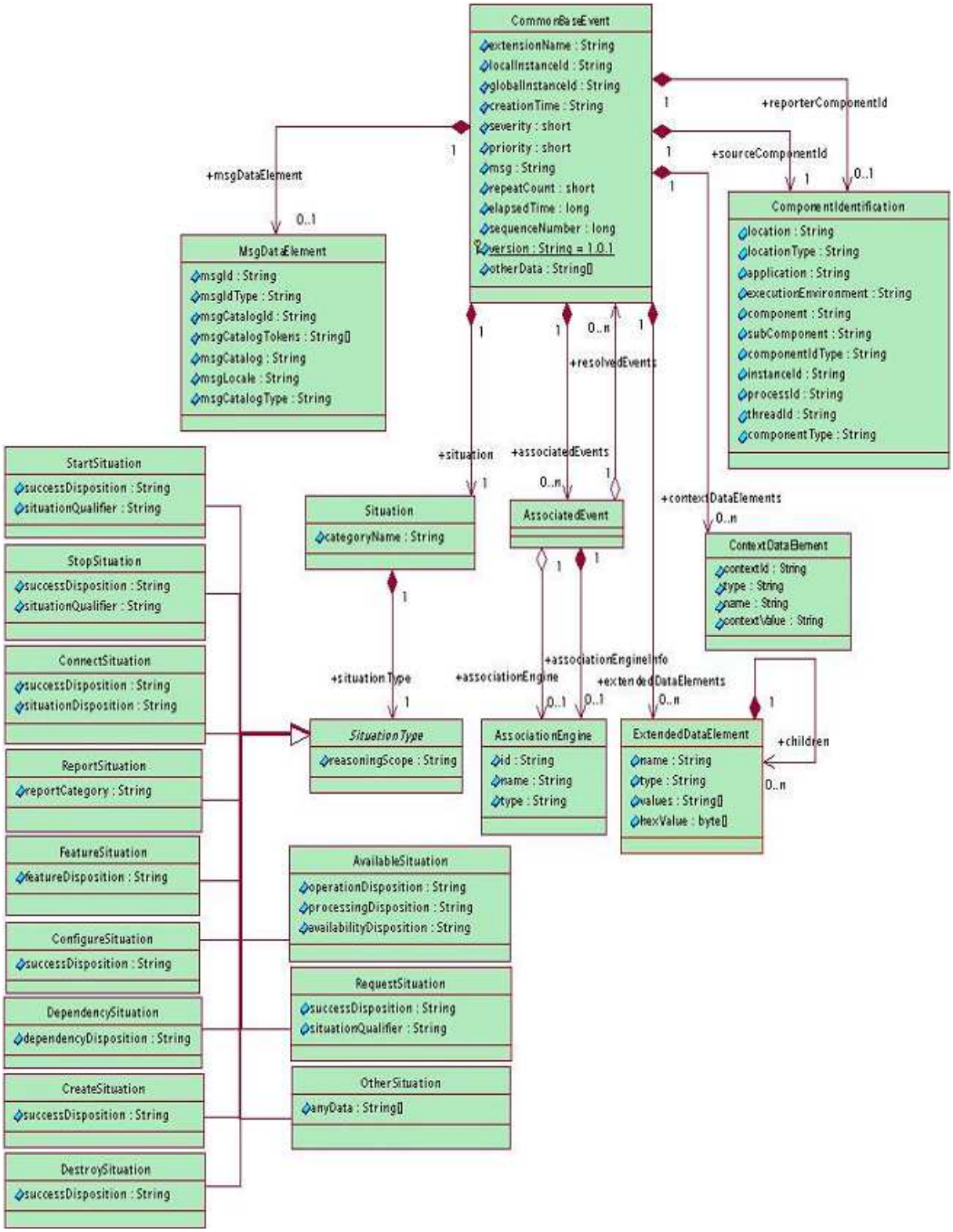



***Figure 46: The Common Base Event class hierarchy***

# Appendix C: The Correlation Tool

## C.1 Java Graphic Libraries

The *Criteria* and *Result* sections below will explain the process used to chose the graphics library to be used with this project. The *Further Information* section will give further information about graphical libraries for future work.

### C.1.1 Criteria

When we started to look for graphics libraries we knew that we wanted to be able to draw a time axis and clickable objects. We also wanted to be able to draw diagrams that could show CPU load and relate this to countable entries from the other logs, i.e. signal count, transaction costs, process hit-rate etc. The libraries should be SWT compliant since the eclipse platform is built using this library for views and graphics.

### C.1.2 Result

The last criterion came to be somewhat of an obstacle. Even though many of the libraries we looked at was said to be SWT compliant, it usually meant that the diagrams could be translated into an image before being shown in the eclipse view, which makes the operation of clicking the chart harder to achieve.

Some of the libraries that seems most promising and that we have tried out are:

- GEF (Graphical Editing Framework) (*http://www.eclipse.org/gef)*
- Zest (*http://www.eclipse.org/gef/zest)*
- JGraph (*www.jgraph.com)*
- JFreeChart (*www.jfree.org/jfreechart)*
- Actuate BIRT (*www.birt-exchange.com*)
- SWT library (*http://www.eclipse.org/swt)*

GEF and Zest are fully SWT compliant, but miss the function to draw axis's and diagrams. JGraph, JFreeChart and Actuate BIRT all seem easy to use and have nice diagram functions, but have to be translated into an image before they can be shown in an eclipse plug-in. Since no one of the above libraries (and also other libraries) seemed to be sufficient we thought about using images for showing the diagrams, and use SWT graphics to manually draw the clickable graphs. It might also be possible to create nice axis's with one of the above libraries as an image, and then draw clickable SWT graphics on top of the image.

For graphically represent relations between states and processes (e.g. transitions between states and signals between processes) Zest could be a good option. Here you can add nodes and connections between them and make both the nodes and connection clickable. The same operations can also be done in GEF and JGraph, but Zest seems easier to work with.

### C.1.3 Further Information

We also found other graphical libraries later in the project that seems promising, but that we never analyzed further. These include:

- aiSee *(http://www.aisee.com)*
- EMF *(http://www.eclipse.org/modeling/emf/)*

The Birt-Exchange community site *(www.birt-exchange.com)* was created late in our project, why we didn't further explore this place. It was created due to the demand for supported products and services based on Eclipse BIRT and it could be used to further examine BIRT support for spreadsheets and diagrams in Java.

The Zest library has released a new version with new features since the one we used in this project.

## C.2 Using the Correlation Tool

The user should collect the log files that might be relevant for tracing the behavior (amongst those that is currently supported). For the application to recognize the different log files these has to be named with the file endings *ktr*, *tae*, *eap*, *tsl* and *cpu* for the Kernel Trace, Trace and Error, Execution Address Profiler, TSL and CPU Load log respectively. When the logs are collected Eclipse should be opened including the Correlation Tool plug-in. The Correlation Tool perspective can then be opened in the "*Window/Open Perspective/Other"* menu option and the user can navigate to where the log files are stored through the Navigation View. From here the user can examine the event data in two different ways. Either by comparing events taking place at the same time as the system behavior, or by finding process dependencies.

For comparing events taking place at the same time the user can select the log files containing time stamps and select "*Open with time chart editor"* from the context menu. If there is need of time synchronization the user can change the time delta for the log files by selecting respective log file and choose "*Change Time Delta"* from the context menu, enter the time difference, close the dialog, and finally update the Time Chart. By finding the event that describes the occurrence of the system behavior, the event data taking place at the same time can be examined to see if one of these events can be the cause of the problem. The user can observe the entry information by hover the mouse pointer over the relevant entry icon, or by selecting the relevant entries and press the "*Open with table editor"* button which will show more extensive information in a table editor.

For finding process dependencies the user can select one or multiple Kernel Trace log files and choose "*Open with table editor"*. By clicking the process tab in bottom left corner or the table editor, all unique processes are listed together with the processes they interact with. The same information can also be seen graphically by instead selecting *"Open with node editor"* from the context menu. If the user knows at what process a certain system behavior takes place, he now knows what other processes that interacts with this processes which might be possible reasons for this behavior. Further information about these processes can be found by selecting the log files containing information about the processes and selecting *"Profile using filter"*. Here the user can enter the name of the processes and press the okay button, which will open table editors from all the selected log files containing information about the process.

For further functionalities that can be used with the Correlation Tool see the *Implemented Functionality* section (section 5.3.3).

## C.3 Adding support for a new log file type

Adding support for a new log file type can be done in the following two steps:

1. A parser has to be created and registered in the Parser Engine. The Parser has to take a file as input, interpret the log file and put the information in the data model.
2. The new log file type should then be set as enabled for the Logs View actions of interest in the Logs View class.

If the functionality given by the Correlation Tool is not sufficient, the Logs View action can make a call to a class extending one of the present editors. Correlation or profiling calculations can be added to the Correlation Engine while the editor is responsible for showing this information.

# Appendix D: Log Files

## D.1 Example Log File Outputs

### D.1.1 Trace and Error log, example output

```
…
[1970-01-13 01:07:35.248] Thread1 ../../errorHandler.cc:712 TRACE7:
          regController: ctr = 55D30FF8, ctrListHead = 55D2AA40, pid = F047B
[1970-01-13 01:07:35.248] Thread2 ../../errorHandler.cc:712 TRACE7:
          regController: ctr = 55D30EA8, ctrListHead = 55D2AC10, pid = E0475
[1970-01-13 01:07:35.248] Thread3 ../../errorHandler.cc:712 TRACE7:
          regController: ctr = 55D318F8, ctrListHead = 55D2AC60, pid = 8047C
…
[1970-01-13 01:08:25.316] application ../../slaveTslTopC.cpp:289 INFO:
          incarnating all the actors of application

[1970-01-13 01:08:25.316] slaveTslDummyCR1 ../../initializeAll.cc:182 TRACE5:
          [RTProfiler EVENT_TYPE_9 - Actor Information]
          Actor: slaveTslDummyCR1[0]
          Actor adress: 1439870592
          Physical thread (controller name): Thread5
          OSE Process: 525437

[1970-01-13 01:08:25.316] slaveTslDummyCR1 ../enterState.cc:59 STATE CHANGE:
          waitForDummySig

[1970-01-13 01:08:25.316] slavewaitCR1 ../../initializeAll.cc:182 TRACE5:
          [RTProfiler EVENT_TYPE_9 - Actor Information]
          Actor: slavewaitCR1[0]
          Actor adress: 1439870304
          Physical thread (controller name): Thread1
          OSE Process: 984187
…
[1970-01-13 01:09:03.468] slavewaitCR1 ../../logMsg.cc:76 REC SIG:
          Signal:READ_REQUEST, Port:slaveWait[0], Sender:readValidCR1[0]
[1970-01-13 01:09:03.468] slavewaitCR1 ../../RTActor/logMsg.cc:130 PARAM:
          Signal:READ_REQUEST, Data:osesig: 1001
[1970-01-13 01:09:03.468] slavewaitCR1 ../../RTActor/enterState.cc:59 STATE CHANGE:
          waitForValidateDataAck
[1970-01-13 01:09:03.468] readValidCR1 ../../logMsg.cc:76 REC SIG:
          Signal:VALIDATE_DATA, Port:readValid_to_slaveWait[0], Sender:slavewaitCR1[0]
[1970-01-13 01:09:03.468] readValidCR1 ../../logMsg.cc:130 PARAM:
          Signal:VALIDATE_DATA, Data:void:
[1970-01-13 01:09:03.468] readValidCR1 ../../enterState.cc:59 STATE CHANGE:
          waitForLockDataAck
…
[1970-01-13 01:09:07.412] application ../src/target/Cello/RTActor/badMessage.cc:196 ERROR:
          application(0)@stop received unexpected message: Top%reqLoad data: void
…
```

### D.1.2 Kernel Trace log, example output

```
…
(7) Time:    3178142.655 ms
   Create process slave1:Thread1
…
(11) Time:    3178142.834 ms
   Create process slave1:Thread2
…
(15) Time:    3178143.041 ms
   Create process slave1:Thread3
…
(111) Time:    3246360.358 ms
   Create process m0226.ppc:master_request
…
(120) Time:    3266362.117 ms
   Send <1001> From: m0226.ppc:master_request To: slave1:Thread1
(121) Time:    3266362.252 ms
   Receive <1001> From: m0226.ppc:master_request To: slave1:Thread1
(122) Time:    3266362.440 ms
   Send <22020096> From: slave1:Thread1 To: slave1:Thread2
(123) Time:    3266362.509 ms
   Receive <22020096> From: slave1:Thread1 To: slave1:Thread2
(124) Time:    3266362.555 ms
   Send <22020096> From: slave1:Thread2 To: slave1:Thread3
(125) Time:    3266362.604 ms
   Receive <22020096> From: slave1:Thread2 To: slave1:Thread3
(126) Time:    3266362.641 ms
   Send <22020096> From: slave1:Thread3 To: slave1:Thread2
(127) Time:    3266362.666 ms
   Receive <22020096> From: slave1:Thread3 To: slave1:Thread2
(128) Time:    3266362.690 ms
…
(240) Time:    3271080.383 ms
   Kill process m0226.ppc:master_request
(241) Time:    3273034.918 ms
   Kill process slave1:Thread1
…
```

### D.1.3 CPU Load log, example outputs

The CPU Load log will contain different information depending on the parameters used to obtain it. Some example outputs of the CPU Load log are shown below.

```
$ capi peak
Top hundred peak load measurement
---------------------------------------------------------
Log entry =  1: Thu Jan 1 23:59:03 1970
irq  0.10
prio 00-07     0.31  0.11  0.68  0.07  0.03  0.05  0.00  0.00
prio 08-15     0.00  0.00  69.79 0.00  0.00  0.00  0.10  0.00
prio 16-23     0.05  0.00  0.00  0.00  0.04  0.00  0.01  0.00
prio 24-31     1.96  0.76  0.00  0.00  0.00  0.00  0.00  0.02
bg  15.91

$ capi type pri 23
CPU load report
---------------
Integration interval: 100000 microseconds
Process name         pid     type    %  %  %  %  %  %  %  %  %  %
Thread3              2046c   pri23   0  0  0  0  0  0  0  0  0  0
main_thread          30469   pri23   0  0  0  0  0  0  0  0  0  0

$ capi name master_request Thread1
CPU load report
---------------
Integration interval: 152000 microseconds
Process name         pid     type    %  %  %  %  %  %  %  %  %  %
master_request       a0481   pri16   0  0  0  0  0  0  0  0  0  0
Thread1              e047a   pri23   0  0  0  0  0  1  0  0  0  0

$ capi prio
CPU load report
---------------
Integration interval: 100000 microseconds
Processes     %    %    %    %    %    %    %    %    %    %
Int           1    1    1    1    0    0    0    0    1    1
bg           96   97   97   98   99   99  100  100   96   96
pri0          0    0    0    0    0    0    0    0    0    0
…
pri31         0    0    0    0    0    0    0    0    0    0
Total        99  100  100  100   99  100  100  100   99  100
```

### D.1.4 TSL log, example output

```
[RoseRT Profiler Data] Sun Feb 8 06:02:45 1970

        [Block 1] *****PROFILER Data for all controllers*****
Profile Total Collect Time      Seconds: 32    nanoSeconds: 657549000

 Peak Signal Intra (InternalQs) Size per Priority
         Priority Level :0       Peak :0
         …
         Priority Level :6       Peak :0
 Peak Signal Inter (IncomingQs) Size per Priority
         Priority Level :0       Peak :0
         …
         Priority Level :6       Peak :0
 Peak Signal Defer (DeferQs) Size:     0

 Total OSE signal dispatch count          1
 Total UML inter signal dispatch count 7
 Total OSE intra signal dispatch count  1

 Global RTLayerConnector mutex contention count 0

 ***Signal Propagation Tree***
 Signal propagation ID: 1 Signal: 5 sent by: 0 received by: slavewaitC.1439346016
delivery latency: 1006
…

        [Block 2] *****PROFILER Data for each controller*****
***Profiler (Controller = main)***
 Profile Collect Time    Seconds: 32    nanoSeconds: 657657000

 Event type 1 (transition cost)
   Key: slaveTslTopC: sendDummySig_stop_8    Value: min: 69  max: 69  med: 69

Event type 3 (msg latency)
  Key: slaveTslTopC: initial_sendDummySig_6 Value: min: 1912 max: 1912 med: 1912

Event type 5 (msg receive)
        Key: application[0]     Value: 2

Event type 7 (state change)
        Key: application[0]     Value: 2

Event type 10 (Peak Signal Inter (IncomingQs) Size per Priority)
         Priority Level :4       Peak :1

Event type 11 (total dispatch count)
         OSE signals: 0
         UML signals inter process: 2
         UML signals intra process: 0

***Profiler (Controller = Thread2)***
…
```

### D.1.5 Execution Address Profiler log, example outputs

***Output type one***

```
#
# Output from:
#   ./6 -f /home/uabafln/exjobb/Correlation of data/TslSlave.ppc.elf
# Generated:
#   Tue Mar 18 12:12:25 2008
#
0xffffffff        1        slave1
0x00000000        100      _Z20new_readReadyC_ActorP12RTControllerP10RTActorRef
0x00000064        92       _ZN16readReadyC_ActorC2EP12RTControllerP10RTActorRef
0x000000c0        92       _ZN16readReadyC_ActorC1EP12RTControllerP10RTActorRef
0x0000011c        96       _ZN16readReadyC_ActorD2Ev
0x0000017c        96       _ZN16readReadyC_ActorD1Ev
0x000001dc        104      _ZN16readReadyC_ActorD0Ev
0x00000408        92       _ZN16readReadyC_Actor9chain3_t1Ev
0x00000464        324      _ZN16readReadyC_Actor11rtsBehaviorEii
0x00000c50        36       _ZNK16readValidC_Actor12getActorDataEv
…
0x000045c0        7700     _vsOutFmt
0x000063d4        224      snprintfOutFoo
0x000064b4        88       vsnprintf
0x00006828        152      vfprintf
0x000068c0        116      fprintf
…
```

***Output type two***

```
# Command Line: execprof -p 25 -f /c/usr/slave1.reg
slave1          :_Z20new_readReadyC_ActorP12RTControllerP10RTActorRef      0   0.00%
slave1          :_ZN16readReadyC_ActorC2EP12RTControllerP10RTActorRef      0   0.00%
slave1          :_ZN16readReadyC_ActorC1EP12RTControllerP10RTActorRef      0   0.00%
slave1          :_ZN16readReadyC_ActorD2Ev                                 0   0.00%
slave1          :_ZN16readValidC_Actor22chain2_gotValidateDataEv           1   0.00%
slave1          :_ZN16slavewaitC_Actor25chain3_gotValidateDataAckEv        1   0.00%
…
slave1          :_vsOutFmt                                                 90  0.02%
slave1          :snprintfOutFoo                                            0   0.00%
slave1          :vsnprintf                                                 0   0.00%
slave1          :efs_outfmt_put                                            69  0.02%
slave1          :efs_stdout                                                0   0.00%
…
```

## D.2 Data that can be extracted

The data that can be extracted from the different log files are shown in the tables below. Since the TSL log is slightly more complex this one is put in a separate table.

| KTR log | TAE log | CPU Load log | EAP log |
|---|---|---|---|
| - Sequence number<br>- Timestamp<br>- OSE event type<br>- Signal id<br>- Sender process<br>- Sender owner (load module)<br>- Receiver process<br>- Receiver owner (load module)<br>- Process dependencies<br>- Delay time between sent messages | - Timestamp<br>- Component name<br>- File or object name<br>- File line number<br>- Trace group<br>- Message<br><br>From trace group STATE CHANGE:<br>- New state<br><br>From trace group SEND SIG and REC SIG:<br>- Signal name<br>- Signal number<br>- Port<br>- Sender process<br>- Receiver process<br><br>From trace group TRACE5:<br>- Actor name<br>- Actor address<br>- Physical thread<br>- OSE process id number | - Log entry no<br>- Timestamp<br>- Interact quota<br>- Highest priorities quota<br>- Total quota at a certain time | - Load module name<br>- Log generation time<br>- Address<br>- Symbol name (including load module, process and function)<br>- Hit rate per symbol component |

***Table 8: Data that can be extracted from the KTR, T&E, CPU Load and EAP logs***

| TSL section | Section info | Data can be extracted |
|---|---|---|
| Header | Header of the TSL log file | Time stamp, controller name , Profile Total Collect Time |
| Event type 1 | Transition cost | Event name, capsule name, capsule state, UML signal number, transition cost (max, med, min) |
| Event type 3 | Message latency | Event name, capsule name, capsule state, UML signal number, message latency (max, med, min) |
| Event type 4 | Signal propagation Tree | Event name , Signal propagation ID, UML signal number, sender address, receiver capsule name, receiver address,<br>delivery latency: 1006 |
| Event type 5 | Total uml msg receive | Event name, actor name, actor index, UML message received count |
| Event type 6 | Total uml msg sent | Event name, actor name, actor index, UML message sent count |
| Event type 7 | Total state change | Event name, actor name, actor index, state changed count |
| Event type 10 | Peak Signal Size per Priority | Event name, event queue type, priority level, peak size |
| Event type 11 | Total dispatch count | Event name, signal type, signal dispatch count |

***Table 9: Data that can be extracted from the TSL Log***

## D.3 Common Event Data between the TSL and Other Log Files

The common event data between the TSL log and other logs is shown in *Table 10* below. In this table, *yes* means that the common event data can be abstracted from the related log files directly; *indirectly* means that the common event data could not be abstracted without any extra information or dealing methods. For example, capsule name could be found both in TSL log and Execution address profiling log, but in execution address profiling log, it is represented together with some redundant information which needs to be filtered away.

| **TSL** | | **T&E** | **KTR** | **EAP** | **CPU Load** |
|---|---|---|---|---|---|
| Time stamp | | yes | yes | yes | yes |
| Controller name | | yes | | yes | yes |
| Total Collect Time | | | | | |
| ET 1, 3 | capsule name | yes | | indirectly | |
| | capsule state | yes | | yes | |
| | UML signal No. | indirectly | indirectly | indirectly | |
| | transition cost/ message latency | | | | |
| ET 5, 6, 7 | Event name | | | | |
| | actor name | yes | | indirectly | |
| | actor index | yes | | indirectly | |
| | UML message rec count/ UML message sent count/ state changed count | | | | |
| | delivery latency | | | | |
| | receiver address | yes | | | |
| | sender address | yes | | | |
| | receiver capsule name | | | indirectly | |
| | Signal propagation ID | yes | | | |
| | UML signal number | indirectly | indirectly | | |
| ET 10 | Event name | | | | |
| | event queue type | | | | |
| | priority level | | | | |
| | peak size | | | | |
| ET 11 | Event name | | | | |
| | signal type | indirectly | | | |
| | signal dispatch count | | | | |

***Table 10: Common event data between the TSL and other log files***

## D.4 Suggested XML format for the TSL log

The XML structure below describes one of our test cases. Where multiple entries can be added to the structure three dots (…) will be shown.

```
<roseRtProfilerData>
        <generationDate value="Thu Jan 15 11:35:24 1970"/>
        <block1>
            <totalCollectTime seconds="53" nanoSeconds="552316000"/>
            <eventType10>
                <internalQueue>
                    <peakSizeEvent pri="0" peak="0"/>
                    <peakSizeEvent pri="1" peak="0"/>
                    <peakSizeEvent pri="2" peak="0"/>
                    <peakSizeEvent pri="3" peak="0"/>
                    <peakSizeEvent pri="4" peak="1"/>
                    <peakSizeEvent pri="5" peak="0"/>
                    <peakSizeEvent pri="6" peak="0"/>
                </internalQueue>
                <externalQueue>
                    <peakSizeEvent pri="0" peak="0"/>
                    <peakSizeEvent pri="1" peak="1"/>
                    <peakSizeEvent pri="2" peak="0"/>
                    <peakSizeEvent pri="3" peak="0"/>
                    <peakSizeEvent pri="4" peak="1"/>
                    <peakSizeEvent pri="5" peak="0"/>
                    <peakSizeEvent pri="6" peak="0"/>
                </externalQueue>
                <deferQueue>
                    <peakSizeEvent pri="0" peak="0"/>
                </deferQueue>
            </eventType10>
            <eventType11>
                <oseSignalCount value="1"/>
                <umlExternalSignalCount value="7"/>
                <umlInternalSignalCount value="1"/>
            </eventType11>
            <eventType4>
                <signalPropagationEntry id="1">
                    <oseSignalNo value="5"/>
                    <sendingActor actorAddress="0"/>
                    <receiveingActor capsuleName="slavewaitC" actorAddress="1439346016"/>
                    <umlMessageDeliveryLatency value="1731"/>
                </signalPropagationEntry>
                <signalPropagationEntry id="1">
                    <oseSignalNo value="3"/>
                    <sendingActor actorAddress="1439346016"/>
                    <receiveingActor capsuleName="readValidC" actorAddress="1439347104"/>
                    <umlMessageDeliveryLatency value="2568"/>
                </signalPropagationEntry>
                <signalPropagationEntry id="1">
                    <oseSignalNo value="3"/>
                    <sendingActor actorAddress="1439347104"/>
                    <receiveingActor capsuleName="readReadyC" actorAddress="1439346640"/>
                    <umlMessageDeliveryLatency value="1780"/>
                </signalPropagationEntry>
                <signalPropagationEntry id="1">
                    <oseSignalNo value="4"/>
                    <sendingActor actorAddress="1439346640"/>
                    <receiveingActor capsuleName="readValidC" actorAddress="1098920712"/>
                    <umlMessageDeliveryLatency value="1030"/>
                </signalPropagationEntry>
                <signalPropagationEntry id="1">
                    <oseSignalNo value="8"/>
                    <sendingActor actorAddress="1439346016"/>
                    <receiveingActor capsuleName="slaveTslTopC" actorAddress="1439345344"/>
```

```
                <umlMessageDeliveryLatency value="4636"/>
            </signalPropagationEntry>
            ...

        </eventType4>
    </block1>
    <block2>
        <collectTime seconds="53" nanoSeconds="552430000"/>
        <rtController type="Main">
            <eventType1>
                <transitionEntry>
                    <transition capsuleName="slaveTslTopC"
                                firstState="sendDummySig"
                                secondState="stop"
                                umlSignalNo="8"/>
                    <transitionCost min="130" max="130" med="130"/>
                </transitionEntry>
                <transitionEntry>
                    <transition capsuleName="slaveTslTopC"
                                firstState="initial"
                                secondState="sendDummySig"
                                umlSignalNo="6"/>
                    <transitionCost min="1211" max="1211" med="1211"/>

                </transitionEntry>
                ...

            </eventType1>
            <eventType3>
                <transitionEntry>
                    <transition capsuleName="slaveTslTopC"
                                firstState="initial"
                                secondState="sendDummySig"
                                umlSignalNo="1"/>
                    <transitionCost min="3689" max="3689" med="3689"/>

                </transitionEntry>
                <transitionEntry>
                    <transition capsuleName="slaveTslTopC"
                                firstState="sendDummySig"
                                secondState="stop"
                                umlSignalNo="8"/>
                    <transitionCost min="4636" max="4636" med="4636"/>
                </transitionEntry>
                ...

            </eventType3>
            <eventType5>
                <actorEntry>
                    <actor name="application" index="0" />
                    <count value="2" />
                </actorEntry>
                ...

            </eventType5>
            <eventType6>
                <actorEntry>
                    <actor name="readReadyCR1" index="0" />
                    <count value="1" />
                </actorEntry>
                ...

            </eventType6>
            <eventType7>
                <actorEntry>
                    <actor name="application" index="1" />
                    <count value="2" />
                </actorEntry>
```

```
                    ...

                </eventType7>
                <eventType10>
                    <internalQueue>
                        <peakSizeEvent pri="4" peak="1"/>
                        ...

                    </internalQueue>
                    <externalQueue>
                        <peakSizeEvent pri="2" peak="1"/>
                        ...

                    </externalQueue>
                </eventType10>
                <eventType11>
                    <oseSignalCount value="0"/>
                    <umlExternalSignalCount value="2"/>
                    <umlInternalSignalCount value="0"/>
                </eventType11>
            </rtController>
            <rtController type="ClientPT">
            ...

            </rtController>
            ...

        </block2>
</roseRtProfilerData>
```

## D.5 Log File Formats

This section shows a few example printouts of the different log file formats.

```
---
Time: 2001-11-23 15:01:42 -5
User: ed
Warning:
  This is an error message
  for the log file
---
Time: 2001-11-23 15:02:31 -5
User: ed
Warning:
  A slightly different error
  message.
---
Date: 2001-11-23 15:03:17 -5
User: ed
Fatal:
  Unknown variable "bar"
Stack:
  - file: TopClass.py
    line: 23
    code: |
      x = MoreObject("345\n")
  - file: MoreClass.py
    line: 58
    code: |-
      foo = bar
```

***Figure 47: YAML log structure***

```
<block2>
    <collectTime seconds="53" nanoSeconds="552430000"/>
    <rtController type="Main">
          <eventType1>
                <transitionEntry>
                      <transition capsuleName="slaveTslTopC"
                                  firstState="sendDummySig"
                                  secondState="stop"
                                  umlSignalNo="8"/>
                      <transitionCost min="130" max="130" med="130"/>
                </transitionEntry>
                <transitionEntry>
                      <transition capsuleName="slaveTslTopC"
                                  firstState="initial"
                                  secondState="sendDummySig"
                                  umlSignalNo="6"/>
                      <transitionCost min="1211" max="1211" med="1211"/>

                </transitionEntry>
          </eventType1>
</eventType3>
```

***Figure 48: Log file structure showing TSL Block 2, Event Type 1***

```
             [Block 2] *****PROFILER Data for each controller*****

***Profiler (Controller = main)***
 Profile Collect Time    Seconds: 53    nanoSeconds: 552430000

 Event type 1 (transition cost)
        Key: slaveTslTopC: sendDummySig_stop_8  Value: min: 130 max: 130
med: 130
        Key: slaveTslTopC: initial_sendDummySig_6       Value: min: 1211
max: 1211       med: 1211
```

***Figure 49: Text log file structure showing TSL Block 2, Event Type 1***