Log Event Filtering Using Clustering Techniques

by

Ahmed Wasfy

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2009

© Ahmed Wasfy 2009

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Large software systems are composed of various different run-time components, partner applications and, processes. When such systems operate they are monitored so that audits can be performed once a failure occurs or when maintenance operations are performed. However, log files are usually sizeable, and require filtering and reduction to be processed efficiently. Furthermore, there is no apparent correspondence of how logged events relate to particular use cases the system may be performing. In this thesis, we have developed a framework that is based on heuristic clustering algorithms to achieve log filtering, log reduction and, log interpretation. More specifically we define the concept of the Event Dependency Graph, and we present event filtering and use case identification techniques, that are based on event clustering. The clustering process groups together all events that relate to a collection of initial significant events that relate to a use case. We refer to these significant events as *beacon* events. Beacon events can be identified automatically or semi-automatically by examining log event types or event names against event types or event names in the corresponding specification of a use case being considered (e.g. events in sequence diagrams). Furthermore, the user can select other or additional initial clustering conditions based on his or her domain knowledge of the system. The clustering technique can be used in two possible ways. The first is for large logs to be reduced or sliced, with respect to a particular use case so that, operators can better focus their attention to specific events that relate to specific operations. The second is for the determination of active use cases where operators select particular seed events of interest and then examine the resulting reduced logs against events or event types stemming from different alternative known use cases being considered, in order to identify the best match and consequently provide insights on which of these alternative use cases may be running at any given time. The approach has shown very promising results towards the identification of executing use cases among various alternative ones in various runs of the Session Initiation Protocol.

Acknowledgements

First of all, I would like to thank God for granting me the great opportunity to come to a fantastic place like Waterloo to do my Masters, for giving me the patience, strength and knowledge to make this work possible, for blessing me with the chance to work with the best supervisor in the world Dr. Kostas, and for everything else I got.. really no matter what I say, it will never be enough.

Next, I am very thankful for Dr. Kostas, the best supervisor in the world! He has been a great mentor throughout this work, I learned from him a lot. He was also always very very patient with me, especially in accommodating my very frequent annoying long emails!

I would also like to thank Drs. Krzysztof Czarnecki and Dr. Ladan Tahvildari for their time and effort to read this work, and for their valuable feedback.

Finally, I would like to thank my parents, family and friends (in Waterloo and Dubai) for supporting me throughout this period, and for making this an unforgettable experience for me.

Dedication

To my parents and brothers

Table of Contents

List of Figures viii
List of Tables ix
Chapter 1 1
Introduction 1
1.1 Problem Description 1
1.2 Contributions 3
1.3 Outline 4
Chapter 2 5
Related Work 5
2.1 Dynamic Program Analysis 5
2.2 Clustering Techniques 10
2.3 Complex Event Processing 13
2.4 Monitoring Frameworks 15
Chapter 3 18
Event Processing 18
3.1 Event Dependency Graph 19
3.1.1 Event Schema 19
3.1.2 Event Dependency Relations 23
3.2 Event Filtering 28
3.2.1 Process Outline 28
3.2.2 Specification Elements 31
3.2.3 Log Filtering Using Clustering 32
3.3 Summary 40
Chapter 4 41
Use Case Identification 41
4.1 Process Outline 42
4.1.1 Simple Use Cases 42
4.2 Complex Use Case 46
4.3 Proof 47
4.4 Summary 54
Chapter 5 55

Experiments 55
5.1 Event Dependency Graph Creation and Clustering 56
5.2 Log Filtering 58
5.3 Use Case Determination 65
5.4 Stability Analysis 67
5.5 Summary 72
Chapter 6 73
Conclusions 73
6.1 Contributions 73
6.2 Concluding Remarks 74
6.3 Future Work 75
References 77

List of Figures

Figure 1 - Complete Event Schema Class Diagram 22
Figure 2 - Log Filtering and Reductino Process for a Selected Use Case 28
Figure 3 - Clustering Configuration Specification 30
Figure 4 - SIP RFC Registration 35
Figure 5 - Summary of the 1-cluster algorithm 36
Figure 6 - SIP RFC Registration 38
Figure 7 - Summary of the n-cluster algorithm 39
Figure 8 - Summary of Use Case Identification Algorithm 45
Figure 9 - EDG Incremental Creation Time 56
Figure 10 - Clustering Time 57
Figure 11 - Total Time (Creation + Clustering) 58
Figure 12 – SIP RFC Registration 60
Figure 13 - Simplified Actual Registration Seq. Diagram 60
Figure 14 - SIP RFC Call Establishment 61
Figure 15 - Simplified Actual Call Establishment Seq. Diagram 62
Figure 16 - Recall/Precision Using the One-Cluster Technique 64
Figure 17 - Recall/Precision Using the N-Clusters Technique 65
Figure 18 – Precision/Recall using stability technique 1 68
Figure 19 – Precision/Recall using stability technique 1 68
Figure 20 - Precision/Recall using stability technique 2 70
Figure 21 - Precision/Recall using stabillity technique 2 70

List of Tables

Table 1 - Summary of CBE attributes 21
Table 2 - Summary of scenarios and corresponding beacon events 59
Table 3 - Average Clustering Time per Scenario 62
Table 4 - Use Case Determination Results 66

Chapter 1

Introduction

1.1 Problem Description

Large software systems are composed of a number of different run-time components, partner applications and processes. In many situations, we need to audit and analyze the log files emitted by these different run-time components, partner applications and processes so that, we can perform root cause analysis, diagnostics, or simply to obtain a view of which use cases may be running at any given point for maintenance, planning, or evolution purposes. However, the analysis of events in log files is a computationally expensive and complex process, especially when many different components and software monitors are involved. Techniques that are being used to analyze log files that are emitted by different sources and in different formats, fall into two main categories. The first category is based on statistical analysis that aims to correlate events using data mining, advanced event correlation techniques and complex event processing techniques. The motivation behind these approaches is for the operator to be able to identify events that exhibit a high degree of co-occurrence and may also associate with a high degree of probability to a particular error or cause of system failure. In this category of approaches the monitoring system must have access to a large number of past cases so that statistically significant correlations can be established first.

The second category is based on pattern matching and on customized diagnostic rules that aim to associate structural patterns of these events to system failures, intrusions, deviations from the expected behavior, or other important system events that require the operator's attention. Approaches in this category suffer from the issue of rule and pattern completeness, in the sense that very detailed rules or insufficient patterns may affect recall while loose patterns may affect precision.

In this work, we take a different approach towards event filtering that can be used not only for log reduction but also for root cause analysis and system understanding. For example, in many situations operators need to know which use cases are running at any given time so that load balancing, resource allocation, and threat determination can be performed.

The premise of the proposed approach is that events in a system, relate both to the particular active use cases involved and to the structural and deployment properties of the system. In this respect, we propose a collection of event dependence relations that require limited knowledge of the inner workings of the system, and can be easily extracted using simple monitoring techniques yet, they provide valuable information on the structure of events in large log files. Once such dependence relations are extracted and an Event Dependency Graph is created, we then propose the use of a clustering technique that groups together all events that relate to a collection of initial significant events that relate to a use case and we refer to as beacon events. The clustering technique is based on an hierarchical agglomerative clustering algorithm with initial conditions. Beacon events can be identified automatically or semi-automatically by examining log event types or event names against event types or event names in the corresponding specification of a use case being considered (e.g. events in sequence diagrams). Furthermore, the user can select other or additional initial clustering conditions based on his or her domain knowledge of the system. The clustering technique can be used in two possible ways. The first is for large logs to be reduced or sliced with respect to a particular use case, so that the operators can better focus their attention to specific events that relate to specific

operations. The second is for the determination of active use cases where operators select particular seed events of interest and then examine the resulting reduced logs against events or event types stemming from different alternative known use cases being considered, in order to identify the best match and consequently provide insights of which of these alternative use cases may be running at any given time. The approach has shown very promising results towards the identification of executing use cases among various alternative ones.

1.2 Contributions

The main contribution of this work is to address the problem of analyzing large volumes of dynamic system information, namely log files. This process can be very computationally expensive and in some cases intractable for practical purposes. One of the possible solutions, that we have adopted to address this problem, is to develop techniques to filter the log events so that logs can be reduced in size and simplified in complexity to allow for easier analysis. We define the concept of the Event Dependency Graph, and apply event filtering and use case identification techniques based on clustering. In this context, the major contributions of the proposed solution are as follows:

- It defines the concept of the Event Dependency Graph that is formed by a collection of relations that aim to denote structural and behavioral associations between events in one or more log files.
- It introduces a novel technique to filter or slice logs, using a heuristic clustering algorithm, with respect to a particular use case. This enables system operators to better focus their attention to specific events that relate to specific operations.
- It proposes an approach for the determination of active use cases running on the system with a small number of initial seed events.

- The techniques presented in this work can be utilized to aid root cause analysis and system understanding.

1.3 Outline

The rest of this thesis is organized as follows:

Chapter 2 provides a literature review of related work in the field. It covers four main subtopics, namely *Dynamic Program Analysis, Clustering Techniques*, *Complex Event Processing* and *Monitoring Framework.*

Chapter 3 introduces the concept of the Event Dependency Graph, and formally defines the relations that constitute the model. The event filtering process is then outlined in more detail along with a description of the specification elements. Finally, algorithms summarizing the two techniques to perform log filtering using clustering are presented.

Chapter 4 builds on the techniques presented in chapter 3, with the aim of determining active use cases running on a system. An algorithm is presented to outline the approach. Sequence diagram variations are also explained, and we outline how our algorithm can still be applicable with all of them.

Chapter 5 shows the results obtained from applying the proposed techniques on two separate sets of data. The first set is a collection of logs obtained from running a collection of predetermined use cases on a NIST implementation of the Session Initiation Protocol. The second set of logs was obtained by simulating a complex system.

Finally, Chapter 6 concludes the thesis by presenting the contributions of this work and discussing some of the future research directions.

Chapter 2

Related Work

This chapter provides an overview of the related work in the field. Four main areas will be discussed, namely *Dynamic Program Analysis, Clustering Techniques, Complex Event Processing* and *Monitoring Frameworks*. Dynamic Program Analysis deals with obtaining data from a running software system to verify certain properties of the system. The Clustering Techniques section discusses general clustering techniques with a special focus given to the Bunch clustering tool used in this work.The Complex Event Processing section discusses techniques for processing multiple events from diverse sources to achieve a certain objective. Finally, the Monitoring Frameworks section elaborates on some of the existing monitoring frameworks that enable software developers and tester to profile and monitor their applications.

2.1 Dynamic Program Analysis

Dynamic program analysis has been extensively used to understand the behavior of software systems. A number of different analysis approaches have been presented in literature. Bruegge et al. [4] designed a framework to support source code instrumentation of systems written in C/C++. K. Koskimies et al. [38] presented another tool, SCED, for source code instrumentation, with the

limitation of being able to monitor independent applications only. Yet both tools assume that access to the source code is available, which might not always be the case. Similarly, there are tools that use compiled-code instrumentation. An example of this is the Java bytecode instrumentation tool, BIT, by H. Lee et al. [88]. Profiling and debugging is another technique used in dynamic program analysis. This technique utilizes interfaces provided by modern development environments to facilitate runtime data collection. Examples of this include the JVMDI [59] and JVMTI [61], which replaces the earlier experimental JVMPI [60], for Java (the Eclipse Test & Performance Tools Platform, discussed in the next subsection, is based on JVMTI). Microsoft.Net framework also has a similar interface, the Common Language Runtime (CLR) Profiler [58]. M. Salah et al. [54] propose an approach, combining dynamic and static analysis, to map use-cases to specific sections of the source code. However this approach could result in limitations such as performance degradation with large systems, and it only works with programs executing within the same process space. The technique proposed in this paper is shown to be more scalable due to the fact that we use selective monitoring depending on a specific use case. Also, by using TPTP our technique works even with applications running on multiple hosts.

H. Safyallah et al. [75] present a technique to perform dynamic analysis of software systems, based on frequent trace patterns, to identify software features in the source code. This is done by instrumenting the code to produce function entry/exit listings. Again, access to source code is assumed here. A. Kinneer et al. [3] discuss an infrastructure, SOFYA, for providing dynamic analysis. The framework uses bytecode instrumentation to capture events and offers a feature to help developers specify program observation without the need for manual modification of the source code. S. Neginhal et al. [43] propose a technique, based on dynamic analysis, which visualizes the relationships between program elements graphically to aid program comprehension. They also developed a tool, CVision, allowing users to select specific parts of the code that are relevant to a

given concern. However the tool only works on programs written in C, and it assumes that the user doesn't only have domain knowledge of the system, but also an understanding of the source code to be able to select the relevant portions of the program. The technique we propose could apply to any system as long as the event logs exist, and no access to or knowledge of the source code is necessary. G. Antoniol et al. [68] present an approach that collects system data and generates a probabilistic model of the system. The dynamic collection of program information utilizes web services as part of their proposed architecture, enabling them to support collection of program information even on distributed clients. In order to save space and improve efficiency, the proposed model collects only summary information, instead of detailed ones. Data is later compressed and encoded following a xml schema and sent to the main server for processing. An interesting variation was presented by G. A. Di Lucca et al. [69] where dynamic analysis was used to collect traces from web applications. The web applications analyzed were all dynamically generated based on a set of initial options specified by the user. In their work, they used the WANDA [23] tool for instrumenting web applications. WANDA aims at recovering the architecture of web applications, and represents it by generating the UML documentation of the system.

A. Zaidman [87] suggested using dynamic analysis to aid program comprehension, with the goal of achieving that in a faster manner. Two techniques were discussed, one based on the frequency of execution, stemming from the observation that program traces will consist mainly of repetitive calls to a small number of methods. The second technique, based on runtime coupling, helps developers know program dependencies at runtime. Similarly, T. Systa [81] presented an environment that uses dynamic program analysis to aid software comprehension. However this work was based on dynamically analyzing java byte code. A prototype environment, SCED, was developed where scenario diagrams and state charts were generated. The tool also provides developers with the option of specifying what classes and methods to be traced. Since this step requires knowledge of class

interactions, a static analyzer, Rigi, is embedded to perform this. Rigi [26] is a reverse engineering tool that can identify all software artifacts in a system and the relations between them. It also supports running queries on the dependency graph so that unnecessary nodes/relations can be filtered out. The tool also applies string matching algorithms to find the required patterns within the event traces. This helps in raising the level of abstraction and decreasing the overall trace size. A similar tool to dynamically analyze Java programs was presented by J. Gargiulo et al. [55]. The tool, Gadget, uses profiling, filtering and clustering techniques to extract dynamic program structures, with the objective of making it easier to understand. This is done by first building a dynamic dependency graph of the classes and calling relationships, and then clustering that graph.

O. Greevy et al. [18] present a very interesting technique to help software comprehension. They use dynamic analysis to achieve an explicit mapping between features and classes. In order to achieve this, they define what is known as feature traces as event traces collected by running a specific set of features in the program. By collecting a large number of feature traces, classes responsible for specific features can be identified. It's worth noting that this approach is complementary, so features that require services from specific classes are also reported. T. Richner et al. [19] propose combining static and dynamic program analysis to support the creation of different views of object-oriented systems. Program traces obtained are stored in a logic database, allowing users to issue queries and obtain system information. In order to filter the large amount of program traces collected, they used their technique iteratively to refine the final view. So the results of the first view are used to filter the tracing options for the second iteration, and so on until a satisfiable view is reached. Along the same line of handling large execution traces, A. Hamou-Lhadj et al. [1] present a way to automatically achieve this. In their work, they try to filter out those traces that are related to utility classes from the ones that implement high-level concepts. The algorithm is based on fan-in analysis. A. Hamou-Lhadj [28] also presented a similar technique called trace summarization. The

technique involves taking a trace as input, and returning a summary of the main events involved as output. To perform this summary, similar techniques to those used in natural language processing are applied, such as extracting events based on naming conventions. This technique was later also semi-automated [48] to allow faster trace summarization.

B. Dufour et al. [6] introduced a framework to help developers understand the dynamic traces generated by their systems. They also introduced a set of metrics that are robust and architecture-independent to help achieve this. These metrics can then be used to aid program comprehension, as they cover different aspects of the code including memory usage and data structures. A comparable tool was also developed at the University of Ottawa [2] to aid program comprehension by collecting execution traces. The Software Exploration and Analysis Tool (STEP), incorporates various filtering techniques to analyze the large volume of traces collected. Traces can be also visualized from within the Eclipse IDE, however they focus only on method calls. An interesting view on the subject was presented by T. Gschwind et al. [82], where runtime data is collected in order to analyze the dynamic behavior of software systems. The developed tool, A Reverse Engineering (ARE) tool, collected parameter and object values to enable developers perform reflective analysis on dynamic method invocations. Other researchers have looked into ways to help software maintainers through dynamic analysis. The Daikon [52] project for example, aims at discovering program invariants by analyzing the execution traces. This would help developers identify what sections of the code need to be preserved when performing code modifications.

Dynamic program analysis was also used to identify design patterns in code. In their work, L. Wendehals et al. [67] compare the collected traces against a behavioral model, sequence diagram in their case. The sequence diagram is converted into deterministic finite automata, then the method call sequence is tested to make sure it conforms to the automata. Their approach also incorporates data collected by a static analyzer to perform the pattern identification. Furthermore, literature has given

some attention to filtering/storing data generated by dynamic program analyzers. R. Brown et al. [70] introduce STEP as a framework for storing program trace data. STEP tries to standardize the way developers handle their trace data. The system provides methods to allow the encoding of trace information in a compact flexible format. The system includes a trace data definition language as well to simplify the encoding of data. In [40] , an event-processing language is presented that is based on regular expressions. EventScript's main goal is to provide real-time response to incoming events.

2.2 Clustering Techniques

Clustering aims at combining observations into clusters/groups, based on a common characteristic that they all share [79]. This helps in achieving a better understanding of the underlying observations. The research community presented a number of different clustering techniques. One of the earliest attempts was presented by L. Belady et al. [20] where they presented an automatic approach to software clustering. Their goal was to reduce the complexity of software systems, by providing a measure of the complexity based on information obtained from the system's specifications. R. W. Schwanke [78] introduced a tool called Arch that offers a semi-automatic clustering approach with the aim of providing developers with modularization advice to help them improve existing code. Arch tries to enforce a good software engineering practice by minimize the coupling between procedures in different modules, and maximize the cohesion of procedures within the same module. Later on Schwanke et al. also explored the use of neural networks to cluster software [29].

The Rigi tool presented by Muller et al. [26] employs a number of clustering heuristics to measure the strength of interfaces across the different subsystems. In their work, they also used the

module names as part of the clustering criteria. S. Choi et al. [76] present a fully-automatic clustering algorithm that is based in a directed resource flow graph. A resource flow graph represents modules as nodes, and arcs denote that the two connected modules provide resources to each other. Similar to Schwanke's work, their work also focuses on maximizing the cohesion of modules. C. Lindig et al. [80] developed a modularization technique that is based on mathematical concept analysis. K. Sartipi and K. Kontogiannis et al. [42] [39] presented a clustering framework with the goal of recovering the architecture of a software system. In their work, they used data mining techniques to extract associations, data and control flows, among components. These associations are then annotated on a graph, and this information is used to apply the clustering.

V. Tzerpos and R. C. Holt [31] presented a clustering algorithm, ACDC, which clusters software systems to help program comprehension. This is done based on a set of subsystem patterns that have shown good program comprehension properties. They also presented a heuristic algorithm to help compute a software clustering metric evaluating the similarities of two decompositions [32]. Along the same lines, they also formally defined the stability of software clustering algorithms and evaluate the stability of different clustering algorithms presented in literature [33].

J. M. Neighbors [65] presented a technique to manually identify software subsystems to extract reusable components. To achieve this, interconnections between components, compile-time and link-time, were examined. A. Lakhotia [46], in an attempt to unify clustering techniques, designed a framework defining a set of symbols and terminologies to describe any clustering approach, including its inputs, outputs and processing. One of the advantages of this work is that it makes comparing different clustering techniques easier, and hence their effectiveness can be evaluated. N. Anquetil and T. Lethbridge [47] presented a clustering technique that uses naming conventions as its clustering criteria, showing some promising results.

Clustering algorithms fall into two main categories namely hierarchical and partitioning algorithms. Hierarchical algorithms find successive clusters using previously established clusters and can be further subdivided to agglomerative and divisive. Agglomerative clustering is done in a bottom-up fashion, where items are iteratively put in the cluster with the most level of similarity. Divisive clustering is top-down, so items are all together at the beginning, and are then iteratively split to form the clusters. The partitioning clustering algorithms on the other hand typically determine all clusters at once [83].

S. Mancoridis et al. [16] [73] [7] [74] treat software clustering as a search problem, and apply search heuristics to solve it. At first, their clustering technique assigns entities randomly to different clusters. Then the search heuristics are applied to move the entities around, and create new clusters if necessary, until better clusters are achieved. The search heuristics are based on hill climbing and genetic algorithms (to overcome the local optima problem of hill climbing algorithms). They have developed a tool, Bunch [63], which incorporates their clustering techniques.

Bunch was the main tool we used to perform clustering during our work. It was designed to be flexible, portable (students and researchers can easily install and use the tool) and fast (execution speed should be fast to allow clustering of large systems). The objective function employed in the tool aims at maximizing cohesion and minimizing coupling across the software modules involved. However, and perhaps this is one of the best features of Bunch, its design allows researchers to develop their own objective functions and clustering algorithms, and incorporate them into the tool. The tool also supports the creation of abstractions of source code by producing a high-level view of the system structure. The main goal behind this was to aid software developers and maintainers understand the structure of large and complex systems. We have used Bunch to cluster our Event Dependency Graph (discussed in the next chapter) to generate events that are highly relevant to a particular use case.

2.3 Complex Event Processing

Complex Event Processing (CEP) is an event processing concept that deals with techniques for processing multiple events from many diverse sources with the goal of identifying the meaningful events within large data sets of collected events. CEP utilizes a variety of techniques such as detection of complex patterns, event correlation and abstraction, use of event type hierarchies, as well as relationships between events such as causality, membership, and timing. In [41] and in [50] the challenges and the themes of CEP as these are applied in large software systems are presented. The research community has also developed several prototype approaches such as the Aurora [15] and Stream [25] projects.

S. K. Chen et al. [72] present a set of adaptive algorithms which help convert structural events into simple name-value pair events that can be later fed into legacy rule-based event correlation engines for Business Performance Management (BPM). Complex events are presented in xml, and then mapped into a smaller set of name-value pairs. In their implemented BPM infrastructure, the Enterprise Service Bus that is used to send real time events to the system does not only get data from external sensors, but also designed to provide a feedback for itself making the engine both an event consumer and a producer. The events are generated based on predefined aggregation and filtering rules. D. C. Luckham [51] introduced the RAPIDE system architecture as an event pattern language and a rule engine based on the collected events. A similar approach was presented by Y. Magid et al. [86] where a partially implemented tool for Complex Event Processing in real time applications was presented. The tool, given a set of rules, generates code for a CEP application. It extends IBM's Active Middleware Technology (a rule-based CEP engine for non-real time applications) to the real-time domain. However, it introduces some restrictions on the IBM Active Middleware Technology to allow it to handle real-time applications. The authors also discuss

different SOA applications that have real-time requirements where this tool may be helpful. A neat feature of this tool is that after a set of rules is given, the tool can calculate the time required to execute the code it generates, and so determine if it'll still meet the application's real-time deadlines.

L. Brenna et al. [44] introduce Cayuga, an event processing engine developed at Cornell University. Some advantages of the system include its ability to scale with the arrival rate of events. The system is designed in such a way that each event has its own relational schema, allowing users to execute queries using a SQL like language. The engine also has a trace visualize displaying how events are matched to each other. Borealis [13] is among the other Complex Event Processing frameworks presented in literature. Extending Aurora's [15] core functionality as an event stream processing engine, Borealis is intended to be a second generation stream processing engine providing capabilities such as dynamic revision of query results, dynamic query modifications and highly-scalable optimizations. A comparable technique was presented in [84], where the authors defined continuous queries, the concept of evaluating queries on streams of data. They also introduced an architecture for handling continuous queries, taking into considerations issues that deal with semantics and efficiency.

A. S. J. Schiefer [77] proposed a new event processing infrastructure to handle real-time Business Intelligence called the Sense And Response Infrastructure (SARI). The main goal behind it is to enable the support of real-time business processes over three types of data, namely past, present and future –oriented. Past-oriented data refers to the original documentation of the system, its business processes and its history. Present-oriented data deals with the ability of the system to respond quickly to varying requirements and handling risks. Finally, future-oriented data represents how the proposed infrastructure detects trends and cycles. This work was also related to the previous work done by A. S. J. Schiefer and C. McGregor in [56]. In their work, they introduced an

architecture to that enables the correlation of events with respect to a business process. The architecture also allows users to apply their own defined functionality to the events.

2.4 Monitoring Frameworks

In the area of monitoring frameworks, the Eclipse Test & Performance Tools Platform (TPTP) Project [22] is an open platform providing software developers and testers with robust tools enabling them to address the entire test and performance life cycle. It is based on the Java Virtual Machine Tool Interface (JVMTI) [62]. It supports a wide range of features from early software testing to production monitoring, including test editing and execution, monitoring, tracing and profiling, and log analysis capabilities. It is tightly integrated with Eclipse, which allows for the profiling of applications from within the Eclipse IDE. In addition to its ability to profile local applications and complex applications running on multiple hosts on different platforms, it also supports embedded systems. We have used TPTP in our work to monitor system events and generate them in a Common Base Event (CBE) [34] log format for processing.

In addition to TPTP, a number of monitoring frameworks exist for almost all major programming languages. The Java PathExplorer [71] (JPAX) is one of the tools used to monitor java applications. The tool automatically instruments Java bytecode, and sends out events to the monitoring module. The monitoring module can then be used to test the incoming events against the system's high level requirement specifications and against lower level error detection procedures. The formal high level requirement specification can be provided in many ways such as temporal logic formulae. Low-level error detection typically tries to find concurrency related errors in the code, or point out their potential existence. These include errors such as race conditions and deadlocks.

Another Java monitoring framework is the one presented by M. Kim et al. in [53]. The Monitoring and Checking (MaC) framework provides a way to verify the correctness of Java programs during run-time. This process consists of 3 stages: 1) Program variables and function call data are extracted from the code. 2) The extracted data is then correlated to the requirements specification of the system. 3) Finally, the occurrence of these events is verified during run-time to ensure that the system behaves according to the specifications. A similar framework was also presented by Y. Cheon [11] named Runtime Assertion Checker (RAC), to enable checking Java programs at runtime. Developers annotate their code using the Java Modeling Language [45] (JML), to add their specifications. This is then translated into Java bytecode, and the specifications are transparently checked during runtime.

Java with assertions [14] (Jass) is another framework that allows developers to test if their systems comply with the specifications. The way the Jass tool works is very similar to RAC. A compiler is used to translate annotated code into Java, and then specifications are checked dynamically during runtime. An additional feature of Jass is that it checks trace assertions, ensuring that methods were invoked in the right order and time. A. K. Mok and G. Liu [49] presented the Java Runtime Timing-constraint Monitor (JRTM) tool, which allows for the monitoring of timing constraints in real-time systems. Developers specify timing constraints using a Real Time Logic (RLC) based language along with the events of interest. Then a monitor tracks the occurrence of each event by storing its name and time, so that synchronization can be enforced.

Many monitoring frameworks also exist for systems written in C/C++. One of these frameworks is BEE++ [5], which was designed to allow monitoring of distributed applications written in C/C++. The framework also provides visualization and debugging classes. Developers manually instrument their code by adding *sensors*. When a sensor is encountered during program execution, data related to the fired event is sent to all the analysis tools (observers) that bound themselves to that

sensor. This takes the load of the executing nodes in the distributed environment, as the event processing is moved to a separate node. Sentry [24] is another C monitoring framework. It is designed to run as a separate process in parallel with a running C program. Sentry observes the execution of the C program, and ensures that it conforms to its specified behavior. Detected errors are reported back to the running application. C. L. Jeffery et al. [10] presented Alamo, a dynamic monitoring framework for C applications. The way it works is conceptually very similar to Sentry. The monitor, called Execution Monitor, executes the target program, and when the execution is over gets back a report of all the events that occurred. Monitoring points are identified from the parse trees, using the C Instrumentation tool CCI [36]. More general monitoring frameworks also exist, like Temporal Rover [17] for example. It enables developers to annotate their Java, C/C++ and Verilog codes using properties specified in Linear Time Temporal and Metric Temporal Logics. The tool's parser then converts annotated programs based on their original language, and the program is validated during execution using the newly generated code.

Chapter 3

Event Processing

This chapter describes how our framework processes events. In Section 3.1, we will introduce the *Event Dependency Graph* (EDG) that is used to represent relations between events. We present the event schema first, and then a total of eight relations will be presented and formally defined. The next step after the creation of the Event Dependency Graph is event filtering. This process will be described in more detail throughout Section 3.2. An overview of the process will be provided in section 3.2.1. Section 3.2.2 describes some of the different specification elements used in the field, and explains how we are using sequence diagrams to filter events. Section 3.2.3 describes how clustering is used to filter events, and presents the 2 algorithms that we have designed to achieve this. Finally, a short summary of the chapter is presented in Section 3.2.4.

3.1 Event Dependency Graph

3.1.1 Event Schema

In order to be consistent and to comply with standards, we have decided to adopt IBM's Common Base Event (CBE) [34] format for encoding events in our log files. Eclipse's TPTP also provides a feature allowing developers to monitor applications and log events in CBE format. Logs are all generated in XML format, following the CBE schema.

The CBE model [9] has a 3-tuple structure allowing it to convey information about the module reporting a particular situation, the module affected by the situation and data about the situation itself. Due to the fact that the reporting and the affected modules are often the same, the CBE schema forces only having the information relating to the reporting module. The third section in the structure, data about the situation, is mandatory.

Each entry (event) in the CBE log file has a number of attributes that containing important information needed for our analysis. These attributes are all summarized in Table 1. However, since this information is not enough for us to build our Event Dependency Graph, we had to add extended elements for each event in the log file, a feature that is supported by the CBE schema. We introduced 5 extended elements as follows:

`<sessionID>` – contains the current session ID that the event belongs to, if any.

`<features>` – contains a collection of features that the event has. These features are typically user defined, as will become clearer in the next section.

`<tasks>` – contains a collection of tasks indicating that the event is originating or is affecting a particular task in a workflow or a business process.

<logicalResource> - contains a collection of resources indicating that the event is originating or is affecting a particular resource related to the logical point of view of the system architecture.

<infrastructuralResource> - contains a collection of resources indicating that the event is originating or is affecting a two (or more) resources that are topologically or infrastructurally related to each other.

<deployedResource> - contains a collection of resources indicating that the event is originating or is affecting the same deployed resource.

<data> - contains a collection of data elements indicating that the event relate to a particular data element. This is typically a persistent storage repository.

These extended attributes could be manually or automatically added to the existing log file, using information from varying sources such as the system specifications or the Configuration Management Database (CMDB). Some of these attributes can also be inferred from the module reporting the situations, for example events that originate from JDBC modules would have a data element. It is also worth noting that all of these extended elements are optional depending on the scenario, since the user could specify what relations to include before processing and clustering the Event Dependency Graph.

Table 1 - Summary of CBE attributes

Attribute	Type	Description
creationTime	String	Event date/time stamp
globalInstanceID	String	A value that uniquely identifies an event
Msg	String	A human readable text providing info about the event
elapsedTime	Long	Time interval between identical event instances
Priority	Integer	A number from 0-100 indicating an event's priority
repeatCount	Integer	Count of identical events
Severity	Integer	A number from 0-10 indicating an event's severity
Application	String	Business name of a component
Component	String	Component/Module generating the event
Location	String	Physical address of the generating component/module
processed	String	Process ID of the current process generating the event
threaded	String	Thread ID of the current thread generating the event

The complete event schema can be summarized as a class diagram as shown in Figure 1. This is a modified version of the CBE schema presented in [9]. The diagram shows all of the attributes we have used in our approach, as well as the relations among different entities. The class 'Event' represents the root element in the CBE schema, containing basic information about an event such as its creation time and global instance ID. Each event would then have a source component that it affects, and optionally a reporting component. Since often both these components are the same, the multiplicity of the reporting component is 0..1 to indicate that it is optional. The extended element class represents all the extended elements that an event could have. Each extended element can also have a number of extended elements for itself, to follow the CBE schema. Finally, the situation element represents the mandatory CBE element containing information about the reporting situation. In our work, however, situation information is not used.

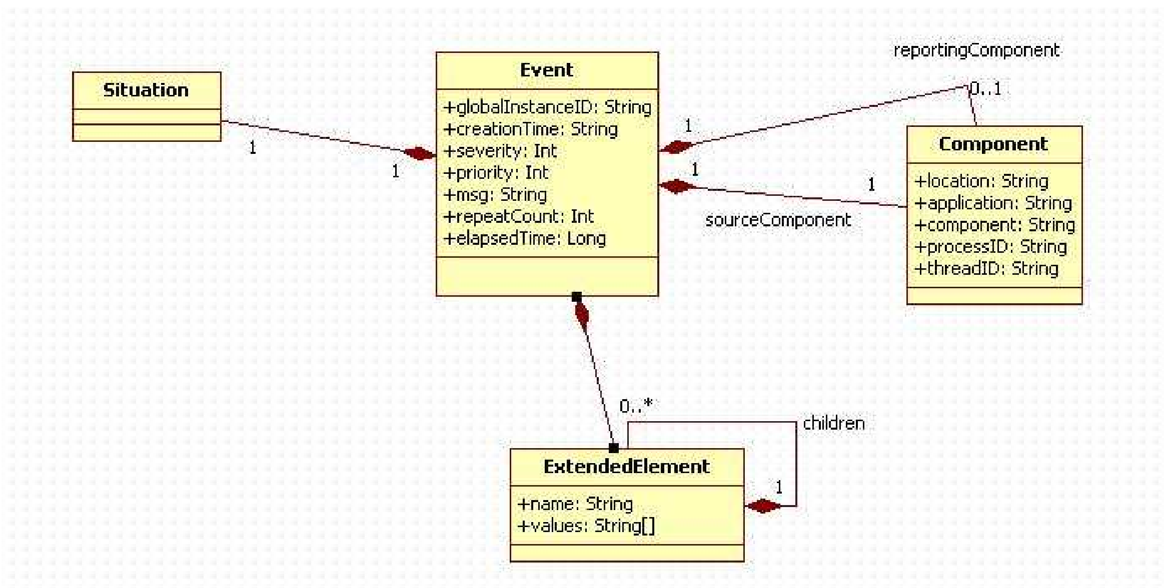

Figure 1 - Complete Event Schema Class Diagram

3.1.2 Event Dependency Relations

The cornerstone of the proposed log filtering and reduction technique is the *Event Dependency Graph (EDG)* that is formed by a collection of relations that aim to denote structural and behavioral associations between events in one or more log files. We define eight event dependency relations that are presented in more detail below.

Coincidental Dependency – Events 1 and 2 are said to be coincidentally dependant if they share a collection of features that the user defines. Formally, this is defined as:

$$COD\ (e1, e2) = \{\ (e1, e2) | \exists\ e1\ \in E1\ \wedge\ \exists\ e2\ \in E2 \\ \wedge\ \forall f\ \in F\ \wedge \left(\ Has(e1, f) \wedge Has(e2, f)\right)\}$$

where:

e1,e2 are single events

E1, E2 are log files

f is a a single feature

F is a collection of user defined event features

Has(x,y) is a predicate with the interpretation "event x has feature y"

Logical Dependency – this type is sub-divided into:

a) **Workflow Dependence –** Events 1 and 2 are said to have workflow dependence if they are produced or consumed by the same task on a workflow or a business process. Formally, this is defined as:

$$WFD\ (e1, e2) = \{\ (e1, e2) | \exists\ e1\ \in E1\ \wedge\ \exists\ e2\ \in E2$$

$$\wedge\ \exists\ t \in T\ \wedge\ \left(Op(e1, t) \wedge\ Op(e2, t)\right)\}$$

where:

t is a single task in a workflow,

T is a collection of tasks in a workflow of business process,

O(x,y) is a predicate with the interpretation of "event x is produced or consumed from task y".

b) Architectural Dependence – Events 1 and 2 are said to have architectural dependence if they originate or consumed by the same resource as this is seen from the logical point of view of the system architecture. Formally, this is defined as:

$$ALD(e1, e2) = \{ (e1, e2) | \exists\, e1 \in E1 \wedge \exists\, e2 \in E2 \\ \wedge \exists\, r \in R \wedge \big(U(e1, r) \wedge U(e2, r)\big)\}$$

where:

r is a component of the logical view of the system's architecture,

R is the collection of components in the logical view of the system's architecture,

U(x,y) is a predicate with the interpretation of "event x is produced or consumed by resource y".

Topological Dependency – Events 1 and 2 are said to be topologically dependent if they originate or consumed by two different resources that belong to the same infrastructure component (e.g. a bean container). Formally, this is defined as:

$$ATD\ (e1, e2) = \{ (e1, e2) | \exists\, e1 \in E1 \wedge \exists\, e2 \in E2 \wedge \exists\, r1 \in R \wedge \exists\, r2 \\ \in R \wedge (U(e1, r1) \wedge U(e2, r2))\}$$

where:

r1, r2 are components that are topologically or infra-structurally related in the logical view of the system's architecture,

R is the collection of components in the logical view of the system's architecture,

U(x,y) is a predicate with the interpretation of "event x is produced or consumed by resource y".

Temporal Dependency – this type is sub-divided into:

a) **Exact Dependence –** Events 1 and 2 are said to have exact dependence if they have the same timestamp. The timestamp can be one of two types, logical timestamps and physical timestamps. Formally, this is defined as:

$$TED(e1, e2) = \{(e1, e2) | \exists\ e1 \in E1\ \wedge\ \exists\ e2 \in E2 \\ \wedge\ EQUAL(ts(e1), ts(e2))\}$$

where:

ts(x) is a function symbol that indicates the logical or physical timestamp of event x.

b) **Range Dependence –** Events 1 and 2 are said to have range dependence if they occur within a specified time frame [t1, t2]. Formally this is defined as:

$$TRD(e1, e2) = \{\ (e1, e2) | \exists e1 \in E1\ \wedge\ \exists\ e2 \in E2 \\ \wedge\ t1 \leq ts(e1) \leq ts(e2) \leq t2\}$$

c) **Approximation Dependence –** Events 1 and 2 are said to have time approximation dependence if they *approximately* have the same timestamp. The range of approximation can be redefined for every pair of events as required. Formally this is defined as:

$$TSD(e1, e2) = \{\ (e1, e2) | \exists\ e1 \in E1\ \wedge\ \exists\ e2\ \in E2 \\ \wedge\ ts(e1) \cong ts(e2)\}$$

Procedural Dependency – this type is sub-divided into:

a) **Process Dependence –** Events 1 and 2 have process dependence if they originate or consumed by the same process. Formally, this is defined as:

$$PID(e1, e2) = \{\ (e1, e2) | \exists\ e1 \in E1\ \wedge\ \exists\ e2 \in E2$$

$$\wedge\ \exists\ pid\ \in P, (B(e1, pid) \wedge B(e2, pid))\}$$

where:

pid is a process of the system with unique process identifier *id,*

P is the collection of processes,

B(x,y) is a predicate with the interpretation of "event x is produced or consumed by process y".

b) **Container Dependence –** Events 1 and 2 have container dependence if they are produced or consumed from two processes operating within the same pool of processes. Formally, this is defined as:

$$PCD(e1, e2) = \{ (e1, e2) | \exists\, e1 \in E1 \ \wedge \ \exists\, e2 \in E2 \ \wedge \ \exists\, pid1 \in Co \ \wedge \ \exists\, pid2 \in Co \ \wedge (B(e1, pid1) \wedge B(e2, pid2))\}$$

where:

pid is a process of the system with unique identifier *id,*

Co is the collection of processes in a container

B(x,y) is a predicate with the interpretation of "event x is produced or consumed by process y".

Transactional Dependency – Events 1 and 2 are said to be transactionally dependent if they relate to the same process ID and same session ID. Formally, this is defined as:

$$PTD(e1, e2) = \{ (e1, e2) | \exists\, e1 \in E1 \ \wedge \ \exists\, e2 \in E2 \ \wedge \ \exists\, tid \in Tr \ \wedge \ (Bs(e1, tid, sid) \wedge B(e2, tid, sid))\}$$

where:
tid is an observed transaction with unique identifier *id*

Tr is the collection of observed system transactions

Bs(x,y, z) is a predicate with the interpretation of "event x belongs to transaction y and relates to session z of the transaction".

Communicational Dependency – and this is sub-divided into:

a) Data Dependence – Events 1 and 2 have data dependence if they relate to the same data. Formally, this is defined as:

$$DAD(e1, e2) = \{(e1, e2) | \exists\ e1 \in E1\ \wedge\ \exists\ \mathrm{e2}\ \in \mathrm{E2} \\ \wedge (\mathrm{A(e1, d)} \wedge \mathrm{A(e2, d)})\}$$

where:

A(x,y) is a predicate with the interpretation of “event x relates to operations on the data element d” (e.g. CRUD operations).

b) Resource Deployment Dependence – Events 1 and 2 have resource dependence if they operate or affect the same deployed resource. Formally this is defined as:

$$DRD(e1, e2) = \{(e1, e2) | \exists\ e1\ \in E1\ \wedge\ \exists\ \mathrm{e2} \in \mathrm{E2}$$

$$\wedge\ \exists\ rd \in Rd\ \wedge \big(\mathrm{U}(\mathrm{e1}, rd) \wedge \mathrm{U}(\mathrm{e2}, rd)\big)\}$$

where:

rd is a component of the logical view of the system’s architecture,

R_d is the collection of deployed run-time components,

U(x,y) is a predicate with the interpretation of “event x is produced or consumed by deployed resource y”.

Correlational Dependency – Events 1 and 2 are said to be correlationally dependent if in the observed history of the system these events occur together within a certain probability or frequency. This definition can also be extended to patterns of events that is a pattern of events P_1e_i, can exhibit correlational dependency with pattern of events P_2e_j. Formally this is defined as:

$$CRD(e1, e2) = \{(e1, e2) | \exists\ e1 \in E1\ \wedge\ \exists\ \mathrm{e2} \in \mathrm{E2} \\ \wedge\ \mathrm{P}co(e1, e2) \geq p\}$$

where:

e1, *e2* are events or patterns of events and,

Pco is the probability or frequency that *e1* and *e2* co-occur in a series of past observations.

3.2 Event Filtering

3.2.1 Process Outline

The proposed log filtering and reduction process is composed of two main phases. A block diagram illustrating the outline of the log filtering and reduction process is depicted in Figure 2.

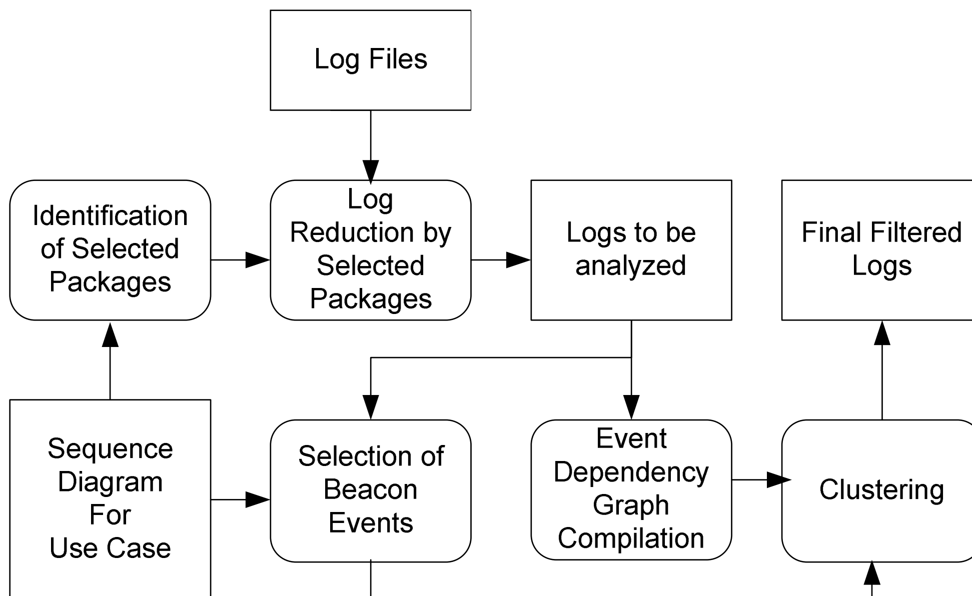

Figure 2 - Log Filtering and Reductino Process for a Selected Use Case

In the first phase the logs are reduced by selecting events that pertain to packages, components, applications and resources of interest as these are implied by the sequence diagram of the specific use case being considered. This part of the process is not restricting the user to consider any other additional package, component, application or resource he or she considers to be of interest. The result of this phase is a collection of log events to be analyzed for the use case being considered and

the creation of an Event Dependency Graph. A configuration file is providing the relationships that the user wants to consider for any given analysis. An excerpt of the clustering configuration file is illustrated in Figure 3. As shown in the excerpt below, the configuration file allows the user to specify 3 main configuration elements, namely the components/software packages of interest, a time frame, the approximation factor (for the approximate temporal dependency) and the PIDs of the processes being monitored. EDG relations matching one or more of the configuration criteria would have higher weights accordingly, and as a result are more likely to be clustered into our output cluster.

```
<?xml version="1.0" encoding="UTF-8"?>
<FilteringCriteriaConfig>

  <Components>
     <Component>net.java.sip.communicator</Component>
     <Component>net.java.sip.communicator.sip</Component>
     <Component>net.java.sip.communicator.sip.security</Component>
  </Components>

  <TimeWindow>
     <startTime>2009-03-23T22:18:15.768Z</startTime>
     <endTime>2009-03-23T22:19:17.253Z</endTime>
     <approximationFactor>0</approximationFactor>
  </TimeWindow>

  <Processes>
     <Process>1</Process>
     <Process>4</Process>
  </Processes>
</FilteringCriteriaConfig>
```

Figure 3 - Clustering Configuration Specification

In the second phase of the process, a number of *beacon* events are selected manually or automatically from the logs to be analyzed based on lexicographical similarity with events appearing in the sequence diagram of the use case. It is noted that one event in the specification may relate to one or more beacon events or event types appearing in the log files to be analyzed. The user may also

decide to consider any other beacon event he or she considers to be of interest. Finally, the event dependency graph along with initial cluster conditions stemming from the beacon events is clustered to produce the final filtered logs.

3.2.2 Specification Elements

Over the years, the software engineering community has proposed a number of different software specification models. One of the most useful specification models for transactional systems are sequence diagrams. Even though these models have been mostly used for requirements specifications, they also denote important relationships between processes and events which are consumed, produced or affect these processes. In this work, we consider that log files generated by various components of a system can become very complex and may include a substantial amount of noisy events that are generated by either the infrastructure or by other applications that are serving many concurrent and in many cases unrelated users and use cases. In order to better filter and isolate the events that may be related to a particular use case or scenario, we consider sequence diagrams as the primary source of information to initiate the filtering process. This information takes the form of sequence diagram events. As sequence diagram events may be represented at a higher level of abstraction than the actual implementation we need to associate a sequence diagram event with one or more events observed in the event log files of the system being monitored. This association is not always straightforward. However, in the research literature there have been a number of techniques that have been proposed to associate model elements that conform to different schemas and domains. One technique is based on fuzzy association rules [85] that is used to associate intrusion models to audit data that is events obtained from the system being monitored. Another technique is based on lexicographical and linguistic similarities between the elements of the models being compared [37].

Yet another technique, is based on similarities between features of the model elements being compared. Such techniques can be used to extract associations between model elements to identify difference between model elements [64], [21].

For our work, we consider a linguistic similarity approach [57] combined with feature vectors to associate high level of abstraction events specified in the sequence diagrams of the system's use cases and method names or event types obtained from the monitoring framework used. The linguistic similarity is used to identify logged events that have a lexicographical similarity with the specified events in the sequence diagrams and the feature vectors are used to limit the types of associate events according to the package or process where they emanate or are received. In this respect, the association technique is aiming to map an event specified in a sequence diagram to one or more events in the log file that are used as beacons for the filtering process. Initial results obtained by the analysis of various scenarios in an implementation of the Session Initiation Protocol indicate a high level of recall and precision in this type of analysis.

3.2.3 Log Filtering Using Clustering

Clustering techniques have been extensively used by the software engineering community to perform software architecture recovery [35] and for mining software repositories [30]. In this work, we utilize a collection of event dependency relations that are used to generate an *Event Dependency Graph (EDG)*. The event dependency graph denotes events and relations between these events as they are collected from the system's monitoring infrastructure. We have utilized the Eclipse Test and Performance Toolkit Platform (TPTP) to collect events at the JVM layer of Java based applications, but the proposed approach can be utilized with any other event extraction and monitoring framework.

For consistency and compliance with standards, we also utilized the Common Base Event format for encoding events obtained from the monitoring modules. The event dependency graph is then clustered utilizing an hierarchical agglomerative clustering algorithm that is discussed in more detail below.

In this work, we experiment with two types of clustering techniques. In the first type, we initiate the clustering algorithm with the condition that all beacon events should appear in one cluster. The events that concentrate in this cluster form the final result. In the second type, we initiate clustering with the condition that each beacon event should appear in a separate cluster. At the end of the process, these clusters are then merged to form the final result. These two different approaches are discussed in more detail below.

3.2.3.1 One-Cluster Initial Condition

In this approach the motivation is to identify the set of events that collectively exhibit the *strongest* relation with the set of selected beacon events. This type of clustering aims to increase precision of the obtained results. In a nutshell, this approach is based on the clustering of the event dependency graph by considering an initial condition where all the log file beacon events are forced to be on the same cluster. The clustering process then not only identifies several other clusters but also identifies additional events that are merged to the initial cluster that contains all the beacon events. Upon termination of the clustering process, the final result is obtained by examining the contents of this initial, and by now extended cluster. The clustering algorithm utilizes the Module Quality metric to identify the clusters with the maximal number of intra-relations (relations in elements within the cluster) and minimum number of inter-relationships (relations of elements across

clusters). The following explains the algorithm more formally and is then followed by an example showing how the algorithm is applied on the *SIP Registration* scenario:

Algorithm: Log-Filtering-One-Cluster

Input: We have our Event Dependency Graph $\mathcal{G}$ that was created to model the relationships between different entries in the log file.

We then have a Set of Beacon events $\mathcal{B} = \{b_1 \ldots b_n\}$ pertaining to the scenario of interest.

These events are typically selected by an expert user.

Output: A cluster of events containing the beacon events selected initially, in addition to other events that relate to the scenario of interest.

Process:

S1. We start by forcing all of our beacon events to be in the same cluster. Let c be an initial cluster $c = \{b_1 \ldots b_n\}$ containing the beacon events $\{b_1 \ldots b_n\}$

S2. The rest of EDG is then divided into a number of clusters. Let $\mathcal{C}$ be the set of all sub-clusters, and initially = $\{ c \}$

S3. We start the clustering process which tries to assign events to clusters using an iterative heuristic approach. For each clustering iteration *i* and

while clustering is not done

S3.1 (Update c with new events)

or,

S3.2 (Create newly formed cluster c_i and set

$\mathcal{C} = \{ c, c_1, c_2, \ldots c_i\}$)

or,

S3.3. Update existing clusters c_1, c_2, $\ldots c_{i-1}$

S4. return (c)

<u>**Example – SIP Registration**</u>

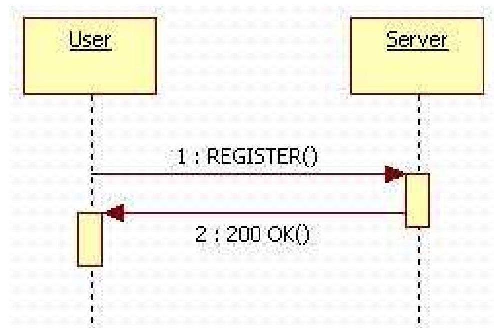

Figure 4 - SIP RFC Registration

<u>Input:</u> We have the created EDG, and we select the following beacon events: Register and OK.

S1. c = {Register, OK}

S2.-4. c = {**Register**, **OK**, *addCommunicationsListener, cancelPendingRegistrations, fireUnregistering, getFromHeader, getLocalViaHeaders, getMaxForwardsHeader, init, initProperties, scheduleReRegistration*}

<u>Output:</u> The c cluster above with the events shown.

The full algorithm is also summarized in Figure 5 below.

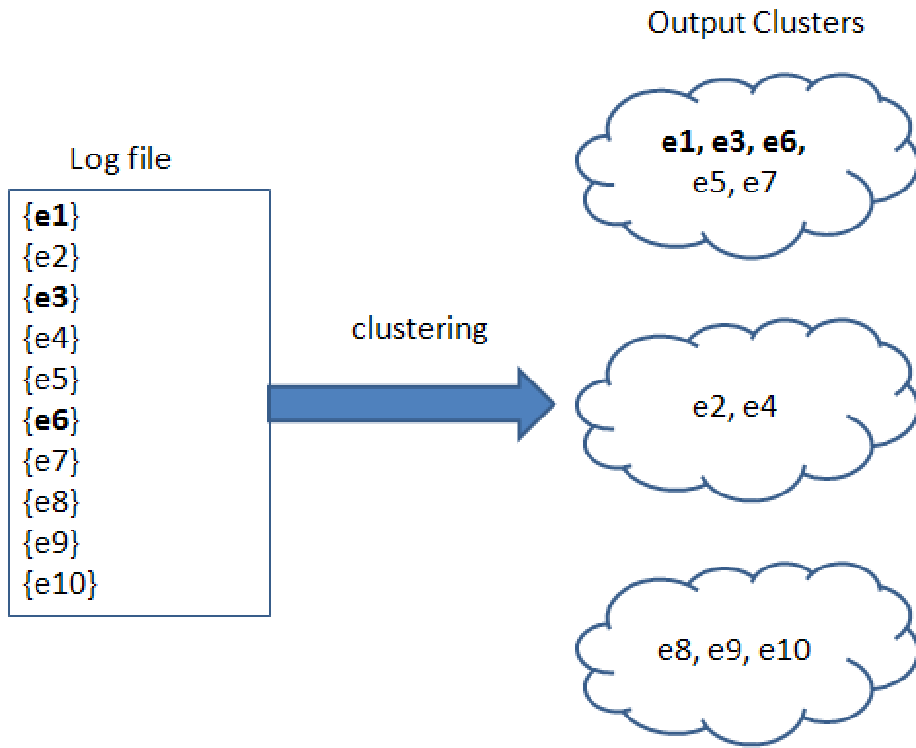

Figure 5 - Summary of the 1-cluster algorithm

3.2.3.2 N-Clusters

In this approach the motivation is to identify the *maximal* set of events that individually exhibit a relation with one or more of beacon events. This type of clustering aims to increase recall of the obtained results. In a nutshell, this approach is based on the clustering of the event dependency graph by considering an initial condition where each beacon event should appear in a separate cluster. In this respect, if we identify `n` beacon events, the initial condition of the clustering process will consider `n` different initial clusters with one beacon element each. All other events in the event

dependency graph will be either placed in one or more of these initial clusters or will form new clusters. In this respect, the clustering process not only identifies new clusters but also, and most importantly, identifies events that are finally placed in the clusters formed by the initial condition. The final result is obtained by considering the all the events that appear in the union of all initial condition clusters. The following explains the algorithm more formally and is then followed by an example showing how the algorithm is applied on the *SIP Registration* scenario:

Algorithm: Log-Filtering-N-Clusters

Input: We have our Event Dependency Graph $\mathcal{G}$ that was created to model the relationships between different entries in a log file.

We then have a set of Beacon events $\mathcal{B} = \{b_1 \ldots b_n\}$ pertaining to the scenario of interest. These events are typically selected by an expert user.

Output: A cluster of events containing the beacon events selected initially, in addition to other events that relate to the scenario of interest.

Process:

S1. We start by forcing each beacon event to be clustered separately in a new cluster. Let $\mathcal{I}$ be a set of Initial Clusters $\mathcal{I} = \{ c_1, c_2 .. c_n \}$ where each cluster c_k contains the beacon event b_k

S2. The rest of EDG is then divided into a number of clusters. Let $\mathcal{C}$ be the set of all sub-clusters, and initially = $\mathcal{I}$

S3. We start the clustering process which tries to assign events to clusters using an iterative heuristic approach. For each clustering iteration *i* and

while clustering is not done

S3.1 (Update c_1 or c_2 or .. c_n with a new event)

or,

S3.2 (Create newly formed cluster c_{n+1} and let

$\mathcal{C} = \{c_1, c_2, \ldots c_n, c_{n+1}\}$)

or,

S3.3. Update existing clusters $c_1, c_2, \ldots c_k$ for $k>n$

S4. At the end of the clustering process, the clusters containing beacon events are all unioned together into one cluster and the algorithm returns ($\bigcup_{i=1..n}(c_1, c_2, \ldots c_n)$)

Example – SIP Registration

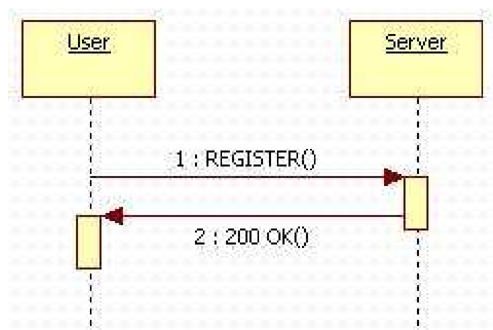

Figure 6 - SIP RFC Registration

Input: We have the created EDG, and we select the following beacon events: Register and OK.

S1. c_1 ={Register} and c_2 = {OK}

S2.-4. c_1 ={**Register**, *addCommunicationsListener, cacheCredentials, cancelPendingRegistrations, checkIfStarted, endAllCalls, fireRegistered, fireRegistering, fireUnregistered, fireUnregistering, getContactHeader*}

and c_2 = {**OK**, *getFromHeader, getLocalHostAddress, getLocalViaHeaders, getMaxForwardsHeader, init, initProperties, processResponse, scheduleReRegistration, start, startRegisterProcess*}

U_i = {**Register**, *addCommunicationsListener, cacheCredentials, cancelPendingRegistrations, checkIfStarted, endAllCalls, fireRegistered, fireRegistering, fireUnregistered, fireUnregistering, getContactHeader,* **OK**, *getFromHeader, getLocalHostAddress, getLocalViaHeaders, getMaxForwardsHeader, init, initProperties, processResponse, scheduleReRegistration, start, startRegisterProcess*}

Output: The U_i cluster shown above.

The full algorithm is also summarized in Figure 7 below.

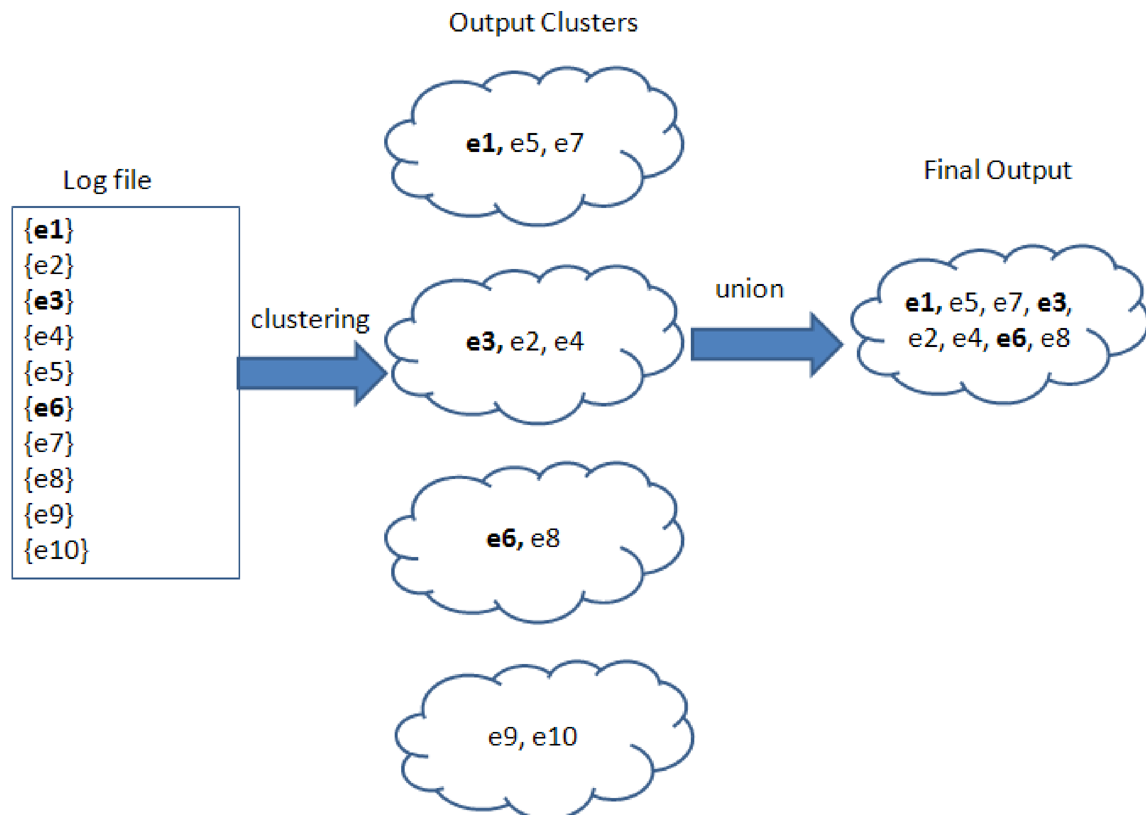

Figure 7 - Summary of the n-cluster algorithm

3.3 Summary

In this chapter, we have introduced our Event Dependency Graph, as a way to model behavioral and structural relations among events. The event schema was presented and the EDG was formally defined with a total of eight relations. We then described our methodology for performing event filtering. The process was clearly outlined, indicating how we used sequence diagrams as a means of specification elements to help us understand how different components and events react and relate to each other. The use of a clustering algorithm to perform event filtering was explained. We also presented 2 algorithms that we have experimented with, namely the one-cluster initial condition and the n-cluster initial condition algorithms.

Chapter 4

Use Case Identification

In this chapter, we will describe how our framework can be used for use case identification. This basically allows a system administrator to scan a system and identify what use cases are running on a system. This can be helpful in a number of situations, for example in order to perform load balancing, resource allocation and threat determination. We outline our approach by presenting an algorithm to achieve our goals. Finally we present a formal proof of the validity of the proposed approach.

4.1 Process Outline

4.1.1 Simple Use Cases

The Event Dependency Graph provides not only a robust model for denoting relationships between events produced by a software system for assisting on log filtering for a selected use case but also, a way of identifying active use cases as a system operates. We start by describing our approach to identify simple use cases, i.e. use cases that can be represented as a simple sequential sequence diagram, without any alternatives, options or loops.

The process is composed of four main steps. The first step is an off-line step aiming to compile a set of significant events per use case that can be used as a golden comparison standard. The second step of the process is for the operator or the monitoring process to select significant *seed* events from the log files of the system. S*eed* events are log file events for which there are reasons to raise interest to either an operator or to an automated monitoring process. The third step of the process is to perform clustering and identify in the log files all other events that are highly related to these initial seed events. The fourth step of the process is to compare precision and recall values of the obtained cluster that contains all seed events, against the collection of beacon events per use case selected in the first step of the process. As the beacon events per use case can be identified off-line and may contain events in a significant level of detail, they are considered as the golden standard to compare the clustering results against. The use cases that correspond to the collection of beacon events with which the obtained clustering results have the highest precision and recall are considered as the possible active use cases in the system. Since the approach is based on identification of a small number of seed events and clustering can be performed with higher computational efficiency than complex event processing and pattern matching, the proposed approach has a benefit over the

traditional rule based or pattern based approaches. The outline of the algorithm for the use case determination is provided below.

<u>Algorithm: Use-Case Determination</u>

<u>Input:</u> - We have our Event Dependency Graph $\mathcal{G}$ that was created to model the relationships between different entries in a log file.

- We have a set of collections of use case beacon events $\mathcal{B} = \{B_1 \ldots B_n\}$ for each possible use case $U_1, U_2, \ldots U_n$. that could run on the system. These beacon events are typically selected by an expert user. This process is only done once and is then stored in a data repository on the system.

- A collection of log file seed events $\mathfrak{S} = \{s_1, s_2, \ldots s_k\}$ pertaining to the log we are currently interested in.

- R a set of tuples $<p_i, r_i, OB_i>$, where p_i is computed precision value of the events of an obtained cluster against the set B_i, r_i is the corresponding recall value , and OB_i is the set of beacon events observed in the obtained cluster. The R tuples are computed for each one of the use cases stored on our system.

<u>Output:</u> A collection of potentially active uses cases U_i where $1 \leq i \leq n$

<u>Process:</u>

1. Let c be the cluster $c = \{e_1 \ldots e_n, s_1, s_2, \ldots s_k\}$ that is one of the clusters obtained by utilizing the 1-Cluster or N-Cluster log filtering process as discussed in Chapter 3, and $s_1, s_2, \ldots s_k$ are the seed events.

2. Let R = empty

3. For each collection B_1 of use case beacon events

3.1 Compute Precision and Recall values $<p_i,r_i>$

between the events of the sets, c and B_1 .

3.2 Update set *R* with <p_i,r_i>

4. We then compute a rank for each R tuple in order to identify the running use case. The rank formula aims at normalizing the precision/recall values obtained for each use case by multiplying the precision by the ratio of beacon events to observed beacon events. Beacon events refers to the number of beacon events that were initially selected for each use case. Observed beacon events refers to the number of beacon events observed in the log file being examined. These values might not be the same if , for example, the current running use case does not run to completion or there's a composition of 2 or more use cases. $U = \text{Rank}(R)) = \{U_i \mid r_i+p_i(|B_i| / |OB_i|) \geq r_j+p_j(|Bj| / |OB_j|), \text{ for all } j, 1 \leq j \leq n\}$

5. At the end, the tuple with the highest rank identifies the running use case. The algorithm returns the rank (*U*) for each tuple. The full algorithm can be summarized as shown in **Error! Reference source not found.**.

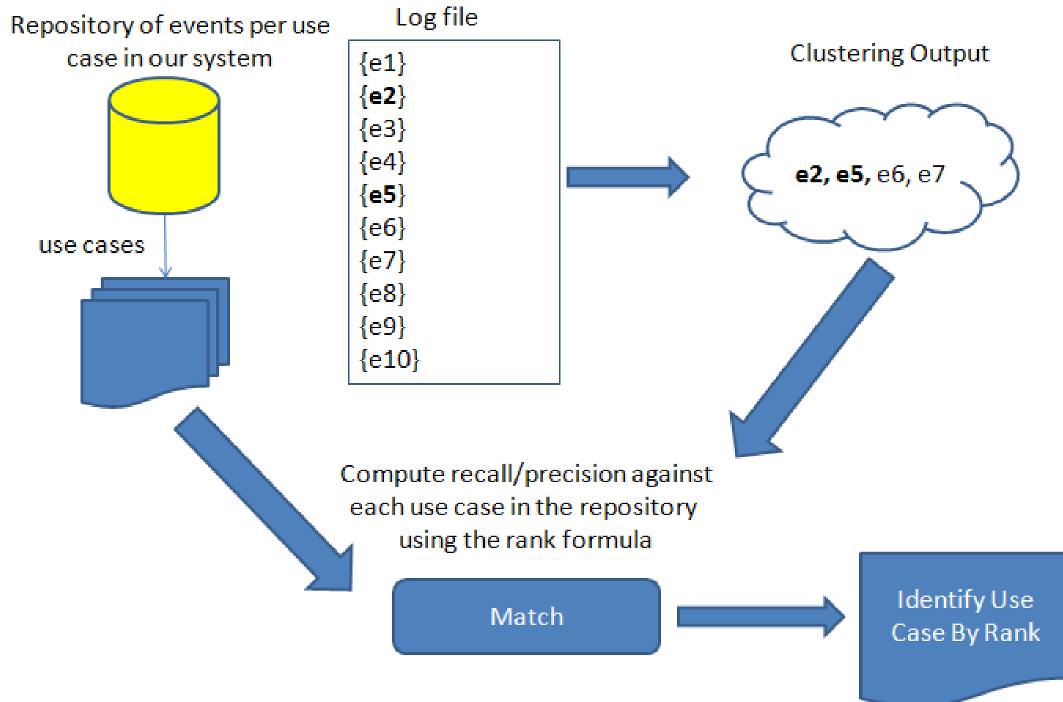

Figure 8 - Summary of Use Case Identification Algorithm

The algorithm above is customized in two ways. The first way is on the selection of the clustering process that is used for computing the set c. For this work we have experimented with both clustering processes namely 1-Cluster and N-Cluster approaches. The results are reported in the experiments section. The second way to customize the above algorithm is the selection of the use case identification process that is utilized by the *Rank* function in step 4 of the algorithm. For this work we select the use cases that correspond to beacon events Bi for which a) the set c. has the highest value of the summation of the recall value among all the B_1, B_2, ..B_n. plus the precision value normalized by the ratio of the number of beacon events by the number of observed beacon events.

The approach has been evaluated with a number of different scenarios or combinations of scenarios stemming from the SIP protocol with high rate of success on the identification of the system's active use cases. The results are presented in Chapter 5.

4.2 Complex Use Case

So far we have presented our use case identification algorithm for simple use cases, i.e. use cases that can be represented as a simple sequential sequence diagram, without any alternatives, options or loops. In this section we prove that more complex use cases can be handled using the same algorithm as well. Complex use cases, as we define them, fall into two categories as follows:

1. **Alternatives and Options –** Alternatives are used to indicate a mutually exclusive choice between two or more sequences [8]. Options are used to indicate a sequence that will occur only if a certain condition is satisfied [8].

In both cases, we end up having a simple sequence of events occurring, irrespective of which paths are taken. This means that we get a sequential composition of methods, just like the simple use case, and hence our algorithm would still work.

2. **Loops –** Loops are used to represent a repetitive sequence of events.

Loops in a sequence diagram can also be broken down into a sequential composition of methods, irrespective of the number of loop iterations. Again, this can be treated just like the simple use case, and hence our algorithm would still work.

4.3 Proof

Intra-connectivity:

Intra-connectivity measures the degree of connectivity among nodes within the same cluster. A higher intra-connectivity value is desirable as it indicates that the nodes that share similar properties are clustered together. Formally, intra-connectivity is presented as follows:

$$A_i = \frac{\mu_i}{N_i^2}$$

where:

μ_i*: the number of intra-edge relations within the cluster*

N_i *: the total number of nodes within the cluster*

Inter-connectivity:

Inter-connectivity measures the degree of connectivity among nodes in different clusters, i.e. cross-cluster connectivity. A lower inter-connectivity value is preferable as it indicates that clusters are independent and more complete. Formally, inter-connectivity is presented as follows:

$$\boldsymbol{E_{i,j} = \frac{\varepsilon_{i,j}}{2N_iN_j}}$$

where:

$\boldsymbol{\varepsilon_{i,j}}$*: the number of inter-edge relations between clusters i and j*

MQ:

MQ measures the overall modular quality of the system and is represented as follows:

$$\boldsymbol{MQ = \frac{1}{k}\sum_{i=1}^{k} A_i - \frac{1}{\frac{k(k-1)}{2}}\sum_{i,j=1}^{k} E_{i,j}}$$

Proof for the 1-Cluster Technique:

Let b_i^1 be a beacon event for use case 1, and let b_j^2 be a beacon event for use case 2. We define B^1 and B^2 , the sets of all beacon events for use cases 1 & 2, as follows:

$$B^1 = \{b_1^1, b_2^1, \dots, b_i^1\}$$

$$B^2 = \{b_1^2, b_2^2, \dots, b_j^2\}$$

Let e_k^1 be all events that can be clustered around B^1 , and let e_m^2 be all events that can be clustered around B^2 . We define E^1 and E^2, the sets of all events that can be clustered around our initial beacon events, as follows:

$$E^1 = \{e_1^1, e_2^1, \dots, e_k^1\}$$

$$E^2 = \{e_1^2, e_2^2, \dots, e_m^2\}$$

If we cluster around beacon events of use case 1, we know that we'll get all the events in the set E^1 , and similarly for use case 2. The question here is what happens if we cluster a compound case consisting of use cases 1 & 2? (i.e. a sequential composition of use cases 1 & 2). In that case, we can have 4 sets of results, as follows:

Case 0: The output cluster contains all events in the sets E^1 and E^2

Case 1: The output cluster contains all events in the set E^1 and a subset of the events in the set E^2

Case 2: The output cluster contains a subset of the events in the set E^1 and all events in the set E^2

Case 3: The output cluster contains subsets of both sets E^1 and E^2

We claim that the MQ in case 0 would be greater than that in cases 1, 2 and 3. And next we are going to prove that and state the conditions where this claim holds. We start by proving that MQ(case 0) is

greater than MQ(case 1), and therefore case 2 as well since it's just the dual of case 1. Then we will prove that MQ(case 0) is greater than MQ(case 3).

1. Proof that MQ(case 0) is greater than MQ(case 1 or case 2):

$$MQ(case\ 0) = \frac{1}{2}\left(\frac{x}{(n1+n2)^2} + \frac{y}{z^2}\right) - \frac{w}{2(n1+n2)z}$$

where:

n1: The sum of the cardinalities of E^1 *and* B^1

n2: The sum of the cardinalities of E^2 *and* B^2

x: The number of intra-edges within the output cluster

y: The number of intra-edges in the rest of the system

z: The total number of nodes in the rest of the system

w: The number of inter-edges in the system

$$MQ(case\ 1) = \frac{1}{2}\left(\frac{x'}{(n1+n2')^2} + \frac{y'}{z'^2}\right) - \frac{w'}{2(n1+n2')z'}$$

where:

n1: The sum of the cardinalities of E^1 *and* B^1

n2': The sum of the cardinalities of B^2 *and a subset of* E^2

x': The number of intra-edges within the output cluster

y': The number of intra-edges in the rest of the system

z': The total number of nodes in the rest of the system

w': The number of inter-edges in the system

Due to the migration of nodes from the output cluster to the rest of the system and vice-versa, the following inequalities hold:

$$z' \geq z$$

$$w' \leq w$$

$$x' \leq x$$

$$y' \geq y$$

Now our goal is to prove that MQ(case 0) > MQ(case 1), or:

$$\frac{1}{2}\left(\frac{x}{(n1+n2)^2}+\frac{y}{z^2}\right)-\frac{w}{2(n1+n2)z} > \frac{1}{2}\left(\frac{x'}{(n1+n2')^2}+\frac{y'}{z'^2}\right)-\frac{w'}{2(n1+n2')z'}$$

which is equal to:

$$\left(\frac{x}{(n1+n2)^2}+\frac{y}{z^2}\right)-\frac{w}{2(n1+n2)z} > \left(\frac{x'}{(n1+n2')^2}+\frac{y'}{z'^2}\right)-\frac{w'}{2(n1+n2')z'}$$

then by moving the expressions around:

$$\left(\frac{x}{(n1+n2)^2}+\frac{y}{z^2}\right)-\left(\frac{x'}{(n1+n2')^2}+\frac{y'}{z'^2}\right) > \frac{w}{2(n1+n2)z}-\frac{w'}{2(n1+n2')z'}$$

for RHS, the following holds:

$$\frac{w}{2(n1+n2)z}-\frac{w'}{2(n1+n2')z'} > \frac{w}{2(n1+n2)z'}-\frac{w'}{2(n1+n2)z'}$$

we know from the above inequalities that z' and n2 are greater than z and n2', and therefore the above inequality is true. As a result, now we need to prove the following:

$$\left(\frac{x}{(n1+n2)^2}+\frac{y}{z^2}\right)-\left(\frac{x'}{(n1+n2')^2}+\frac{y'}{z'^2}\right) > \frac{w}{2(n1+n2)z'}-\frac{w'}{2(n1+n2)z'}$$

or:

$$\frac{x}{(n1+n2)^2} - \frac{x'}{(n1+n2')^2} + \frac{y}{z^2} - \frac{y'}{z'^2} > \frac{w}{2(n1+n2)z'} - \frac{w'}{2(n1+n2\quad)z'}$$

similarly, we can replace n2 and z by n2' and z' as follows:

$$\frac{x}{(n1+n2')^2} - \frac{x'}{(n1+n2')^2} + \frac{y}{z'^2} - \frac{y'}{z'^2} > \frac{w}{2(n1+n2)z'} - \frac{w'}{2(n1+n2\quad)z'}$$

now we can group the terms as follows:

$$\frac{x-x'}{(n1+n2')^2} + \frac{y-y'}{z'^2} > \frac{w-w'}{2(n1+n2)z'}$$

As a result we can conclude that the MQ(case 0) will always be greater than MQ(case 1 or case 2) as long as the sum of the reduction on intra-edges over the output cluster and the rest of the system is greater than the sum of the reduction of inter-edges over the output cluster and the rest of the system.

2. Proof that MQ(case 0) is greater than MQ(case 3):

$$MQ(case\ 0) = \frac{1}{2}\left(\frac{x}{(n1+n2)^2} + \frac{y}{z^2}\right) - \frac{w}{2(n1+n2)z}$$

where:

n1: The sum of the cardinalities of E^1 *and* B^1

n2: The sum of the cardinalities of E^2 *and* B^2

x: The number of intra-edges within the output cluster

y: The number of intra-edges in the rest of the system

z: The total number of nodes in the rest of the system

w: The number of inter-edges in the system

$$MQ(case\ 1) = \frac{1}{2}\left(\frac{x'}{(n1' + n2')^2} + \frac{y'}{z'^2}\right) - \frac{w'}{2(n1 + n2')z'}$$

where:

n1: The sum of the cardinalities of E^1 *and* B^1

n2': The sum of the cardinalities of B^2 *and a subset of* E^2

x': The number of intra-edges within the output cluster

y': The number of intra-edges in the rest of the system

z': The total number of nodes in the rest of the system

w': The number of inter-edges in the system

Now our goal is to prove that MQ(case 0) > MQ(case 3), or:

$$\frac{1}{2}\left(\frac{x}{(n1 + n2)^2} + \frac{y}{z^2}\right) - \frac{w}{2(n1 + n2)z} > \frac{1}{2}\left(\frac{x'}{(n1' + n2')^2} + \frac{y'}{z'^2}\right) - \frac{w'}{2(n1' + n2')z'}$$

which is equal to:

$$\left(\frac{x}{(n1 + n2)^2} + \frac{y}{z^2}\right) - \frac{w}{2(n1 + n2)z} > \left(\frac{x'}{(n1' + n2')^2} + \frac{y'}{z'^2}\right) - \frac{w'}{2(n1' + n2')z'}$$

then by moving the expressions around:

$$\left(\frac{x}{(n1 + n2)^2} + \frac{y}{z^2}\right) - \left(\frac{x'}{(n1' + n2')^2} + \frac{y'}{z'^2}\right) > \frac{w}{2(n1 + n2)z} - \frac{w'}{2(n1' + n2')z'}$$

for RHS, the following holds:

$$\frac{w}{2(n1 + n2)z} - \frac{w'}{2(n1' + n2')z'} > \frac{w}{2(n1 + n2)z'} - \frac{w'}{2(n1 + n2\ \ )z'}$$

we know from the above inequalities that z' , n1 and n2 are greater than z, n1 and n2', and therefore the above inequality is true. As a result, now we need to prove the following:

$$\left(\frac{x}{(n1+n2)^2}+\frac{y}{z^2}\right)-\left(\frac{x'}{(n1'+n2')^2}+\frac{y'}{z'^2}\right)>\frac{w}{2(n1+n2)z'} - \frac{w'}{2\big(n1+n2\quad\big)z'}$$

or:

$$\frac{x}{(n1+n2)^2}-\frac{x'}{(n1'+n2')^2}+\frac{y}{z^2}-\frac{y'}{z'^2}>\frac{w}{2(n1+n2)z'} - \frac{w'}{2\big(n1+n2\quad\big)z'}$$

similarly, we can replace n1, n2 and z by n1', n2' and z' as follows:

$$\frac{x}{(n1'+n2')^2}-\frac{x'}{(n1'+n2')^2}+\frac{y}{z'^2}-\frac{y'}{z'^2}>\frac{w}{2(n1+n2)z'} - \frac{w'}{2\big(n1+n2\quad\big)z'}$$

now we can group the terms as follows:

$$\frac{x-x'}{(n1'+n2')^2}+\frac{y-y'}{z'^2}>\frac{w-w'}{2(n1+n2)z'}$$

Again, we can conclude that the MQ(case 0) will always be greater than MQ(case 3) as long as the sum of the reduction on intra-edges over the output cluster and the rest of the system is greater than the sum of the reduction of inter-edges over the output cluster and the rest of the system.

Therefore, our algorithm would still work as it will always produce a higher MQ value given the above conditions. Chapter 5 further supports this by presenting experimental evidence.

Proof for the n-Cluster Technique:

Since the n-Cluster algorithm performs a UNION operation at the end to join all output clusters together, having a simple use case or a complex (sequential composition of two or more cases) one will result in no change in the output. Therefore we can conclude right away that the n-Cluster technique will always be valid for both categories of use cases.

4.4 Summary

In this chapter, we have presented how our framework could be used to identify running use cases on a system. This is of vital importance to system administrators as it enables them to perform threat determination, resource allocation and load balancing tasks. We have formally presented and proved our algorithm and also described how it would still work with different variations of sequence diagrams, including more complex ones with alternatives, options and loops. Experimental results on use case determination are presented in Chapter 5, where we were able to achieve a 100% accuracy in identifying active use cases on a system.

Chapter 5

Experiments

In this chapter, we validate our framework by presenting the experimental results that we have collected. We have conducted experiments on two sets of data. The first set belongs to logs collected by running real life scenarios of a NIST implementation of the Session Initiation Protocol (SIP), called the SIP Communicator. Logs were collected using Eclipse's TPTP. The second set of data is composed of a set of simulated log files that were automatically generated. In the first section of this chapter presents results related to the creation of the Event Dependency Graph (EDG) and clustering. Section 2 focuses on log filtering by presenting the recall and precision results obtained after filtering logs relating to different scenarios. In the third section, we show the results of using our framework for use case determination. Finally, we present a stability analysis to validate our approach.

5.1 Event Dependency Graph Creation and Clustering

The Event Dependency Graph is created incrementally with linear complexity on the number of nodes of the dependency graph. Assuming that the number of relations is orders of magnitude smaller than the number of nodes in the graph the creation of the graph can be done very efficiently as the system operates. In this work the EDG creation process is based on a single threaded sequential traversal of nodes and the establishment of elations of existing nodes of the graph with the newly added node. However, this process can be greatly optimized in a production environment by hashing and partitioning the graph with the purpose of excluding nodes that definitely do not relate with the newly added node, and by performing bitwise operations and masks for feature matching between the newly added node and other nodes in the graph.

Experimental results related to the time required to compile an event dependency graph are illustrated in Figure 9, where the time to compile incrementally an EDG is exhibiting linear behavior. The graph is compiled in steps of 10 events and for a total of 10,000 events (i.e. EDG nodes).

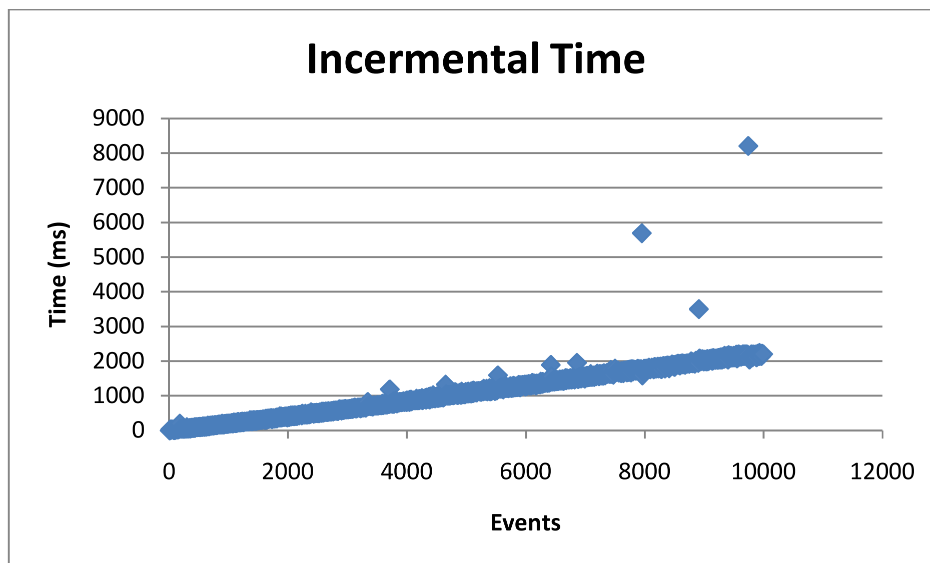

Figure 9 - EDG Incremental Creation Time

The second step after the creation of the Event Dependency Graph is to cluster this graph. The time required to perform clustering varies depending on the number of relations that exist in the EDG and/or the number of relations that the user wants to include during the clustering process. This depends on the scenario or the goal behind which the user applies clustering. The total time required for clustering, up to 500 events, is shown in Figure 10. We observe that the time demonstrates an exponential behavior. A similar conclusion can also be made regarding Figure 11, which shows the total time required to perform both EDG creation and clustering with up to 10,000 events. This behavior is expected due to the time required to perform clustering. We will elaborate more on other ways to reduce the clustering time in Chapter 6, as part of the future work section.

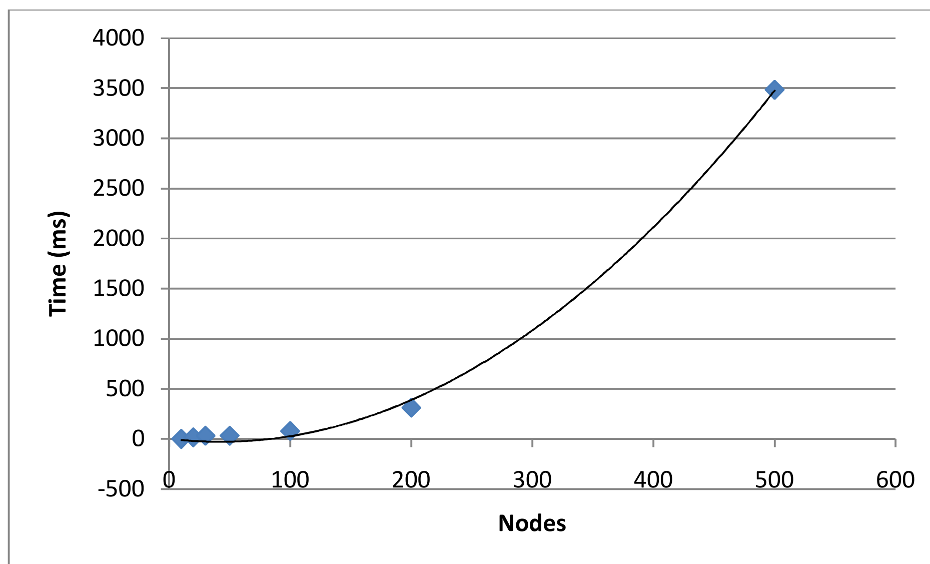

Figure 10 - Clustering Time

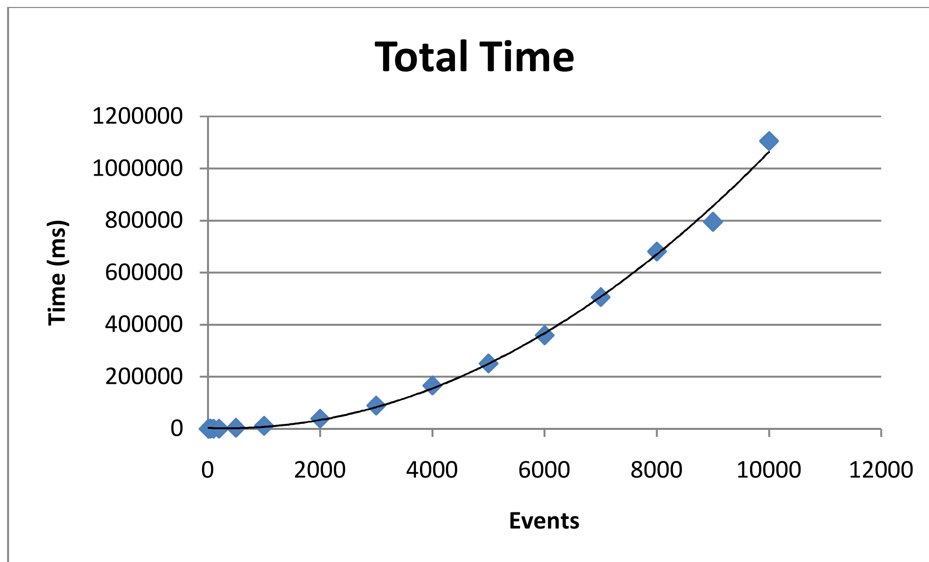

Figure 11 - Total Time (Creation + Clustering)

5.2 Log Filtering

In order to validate the proposed log filtering and reduction approach, we have applied it to the SIP Communicator system [66], which is part of the implementation of the Session Initiation Protocol (SIP) [27] developed by the National Institute of Standards and Technology (NIST). SIP is an application layer signaling protocol that is used to start, modify and terminate various types of sessions such as Internet Telephony calls. We used TPTP [22] to monitor and log all the events in the system.

A total of eight different scenarios (summarized in Table 2) were studied, half of which were basic scenarios and the other half were logical combinations of those basic scenarios. Each scenario generates more than forty thousand events even for the simplest case. For this experiment only related to the scenario source code packages were monitored in order to reduce the number of events per log file. Packages were selected according to how relevant they are with the respect to a given use case,

which was represented in the form of a sequence diagram. An example of this is illustrated in Figure 12 and Figure 13. Figure 12 illustrates the SIP RFC standard for the registration scenario, while Figure 13 illustrates the corresponding modified sequence diagram for that scenario based on the SIP Communicator implementation. It is worth mentioning that the process of selecting only specific packages for monitoring by TPTP resulted in a considerable reduction in the amount of events logged. The next step of the process is to identify beacon events in the system that relate to our specific scenario. In our registration example, those events should correspond to the REGISTER message and the 200 OK response code according to SIP's RFC.

Table 2 - Summary of scenarios and corresponding beacon events

Scenario	Number of Beacon Events
Registration	2
Call Establishment	3
Call Failure (No Answer)	5
Call Termination	2
Reg. + Call Est.	5
Reg. + Call Fail.	7
Reg. + Call Est. + Call Term.	7
Call Est. + Call Term.	5

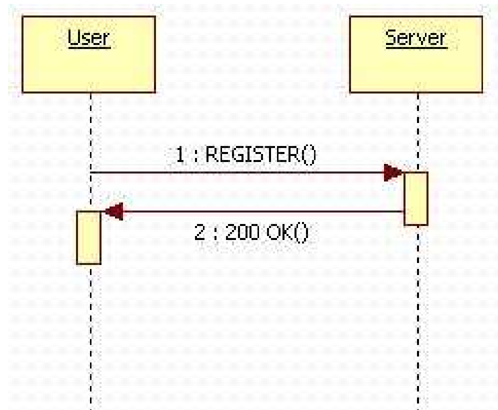

Figure 12 – SIP RFC Registration

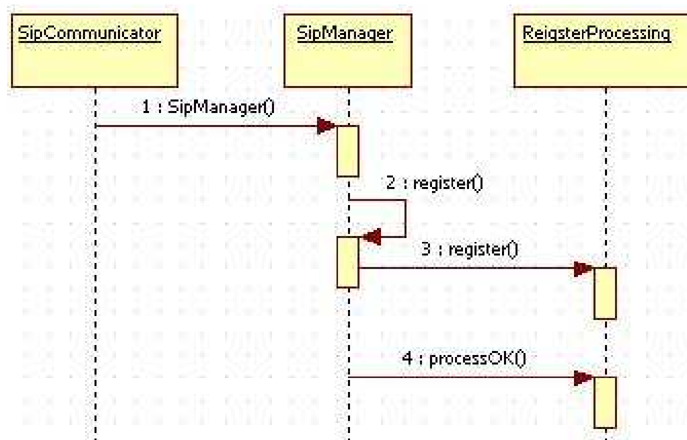

Figure 13 - Simplified Actual Registration Seq. Diagram

From the sequence diagram we can clearly identify that this corresponds to the register() and processOK() methods, respectively. Those beacon events will be our initial criteria for performing the clustering for this use case. The number of beacon events varies by scenario, as shown in Table 2. The Event Dependency Graph (EDG) is then built from the log file, as described in Chapter 3, using our Java-based tool. Our tool uses JGraphT [12], which is an open-source Java graph library, to

construct the EDG. The final step is to perform the clustering using the Bunch tool by utilizing the hill climbing heuristic optimization option in Bunch, as empirical results showed that it outperforms the genetic algorithms approach [63].

Similarly, sequence diagrams relating to the call establishment scenario are shown in Figure 14 and Figure 15. One can also clearly observe the semantic similarity between the RFC protocol and the methods, as shown for example in the INVITE signal and the invite() method.

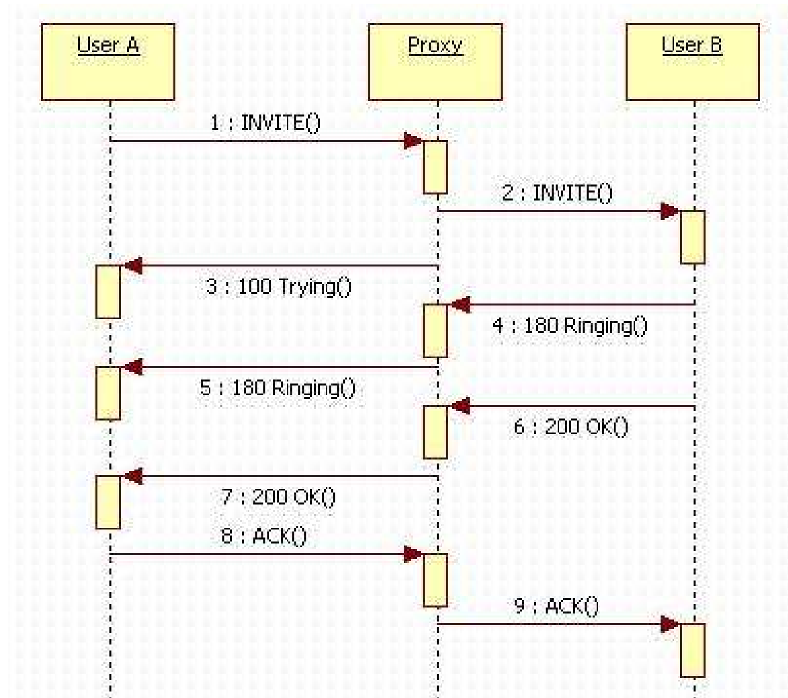

Figure 14 - SIP RFC Call Establishment

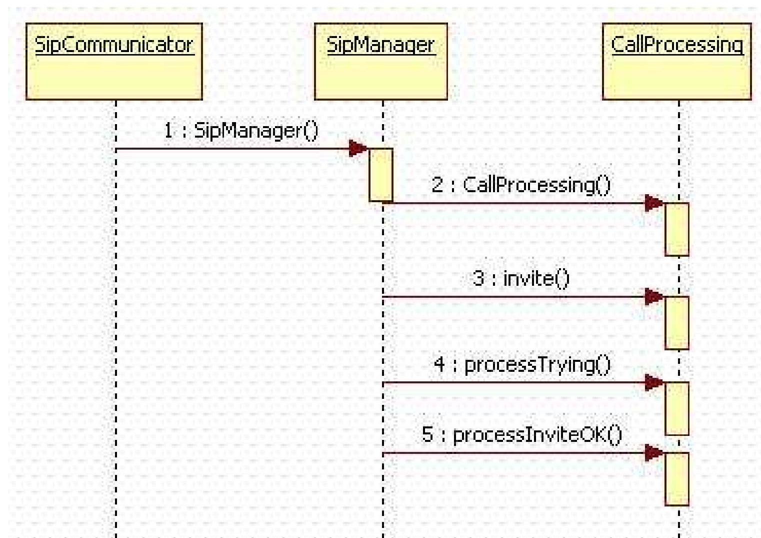

Figure 15 - Simplified Actual Call Establishment Seq. Diagram

For these experiments, we have used both the 1-Cluster and the N-Cluster initial condition techniques. The data illustrating the average number of events per cluster and the time to filter the events for the different use cases is summarized in Table 3.

Table 3 - Average Clustering Time per Scenario

Scenario	Average Cluster Size	Time (s)	Number of Events
Registration	12	1	176
Call Establishment	30	5	377
Call Failure (No Answer)	23	5	322
Call Termination	23	5	382
Reg. + Call Est.	32	6	377
Reg. + Call Fail.	35	4	322

Reg. + Call Est. + Call Term.	34	7	382
Call Est. + Call Term.	19	4	382

From the above results we observe that the time required to perform the clustering operation increases in a scalable manner as compared to the number of events. Furthermore, in order to evaluate whether the proposed clustering technique actually filtered events relevant to each scenario, we computed a "golden-standard" to compare against. For this work we considered this golden standard to be the logs that can be obtained by running the scenarios in debug mode. This was possible as we had full access to the source code and we could identify the methods that had to be monitored.
By doing so, the system registered method entry/exit details that could then be compared against the results obtained by the proposed clustering process. More specifically, the clustering results were compared against the golden-standard, and we computed precision and recall values for each scenario using both of the proposed clustering techniques. Figure 16 and Figure 17 illustrate the precision & recall results for techniques 1 and 2 respectively.

From the obtained results we also observe that both techniques were able to achieve high recall values for most of the scenarios. More specifically, in the one-cluster techniques we have obtained high precision values at a slight cost of recall in some cases (see Registration and Call Termination scenarios). This reduced recall may be due to the fact that these scenarios involve a small number of sequence diagram and beacon events, and the clustering few initial beacon events to start with.

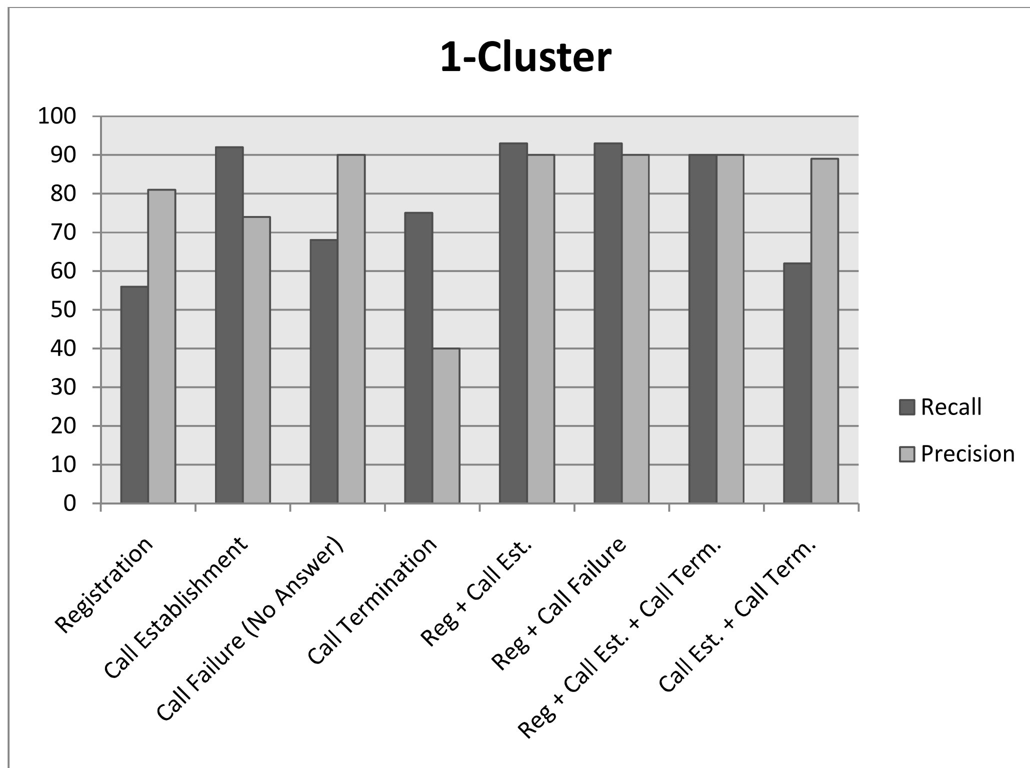

Figure 16 - Recall/Precision Using the One-Cluster Technique

Nonetheless, the one-cluster technique was able to achieve more than 90% recall in three of the scenarios with fairly high precision rates. The n-clusters (union) technique achieved a 100% recall in three of the scenarios, however now at the cost of precision (see Call Termination, Registration and Call Failure scenarios). This is an expected result as the union of `n`-clusters may introduce noisy events that may have been clustered as part of the process in each of the `n` clusters.

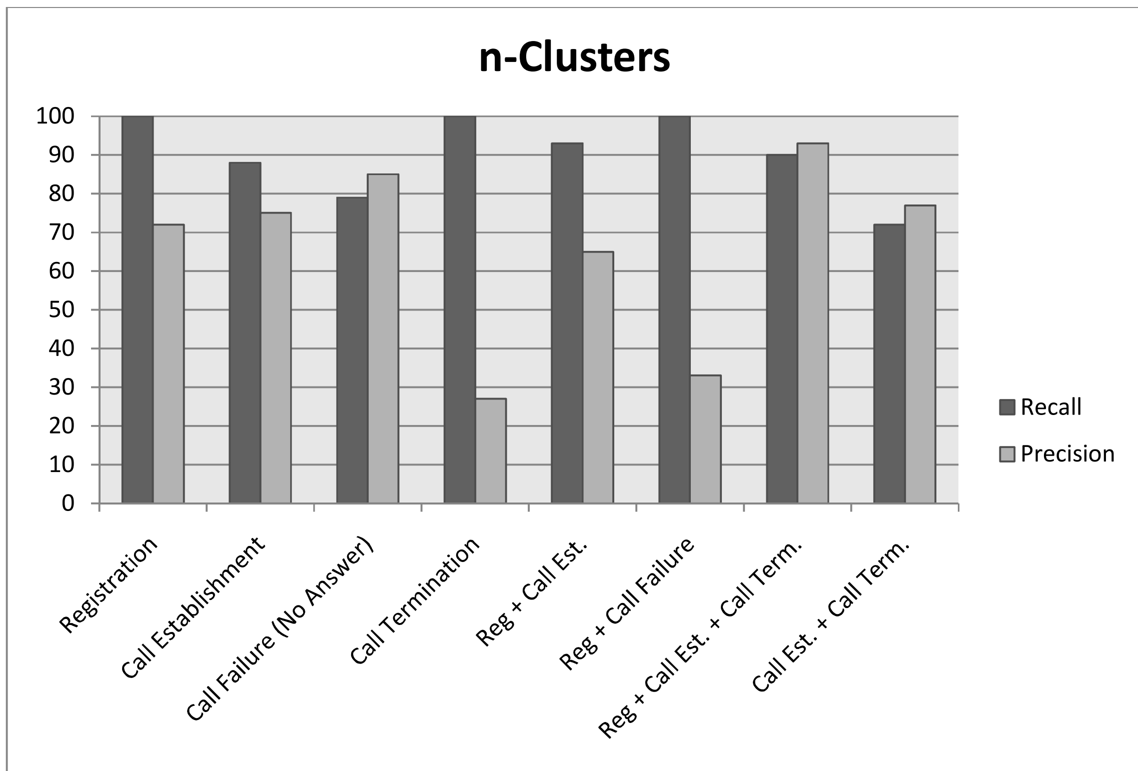

Figure 17 - Recall/Precision Using the N-Clusters Technique

5.3 Use Case Determination

For the evaluation of the use case determination process, we have experimented with a number of different use case scenarios and combinations of these scenarios to obtain actual event logs from the NIST implementation of the Session Initiation Protocol. The experiments run in three major phases. In the first phase we run the system with different use cases and obtained the maximum number of events by running the system in the debug mode. These events were then considered to be the golden standard for each use case and formed a data pool for comparison. The second step of the experimentation process was to run the system in a known use case, obtain events using the TPTP

tool and select *seed* events. The third step was to perform clustering based on the *seed* events and compare the obtained cluster against the golden standard collection of events for each use case. The evaluation of the process was based on whether the comparison process could yield the use case that run, among all the possible ones. In this respect, the use case that corresponds to a cluster of golden standard events that associates the most with the obtained cluster using the seed events is a possible active use case. The obtained process evaluation results indicated that precision and recall were very accurate metrics for determining the use case that runs. The results are illustrated in Table 4. In this table we considered four use cases each one active at a time. By selecting seed events (2 seed events for the first, second and third use cases, and 1-2-1 events for the fourth use case) performed clustering and compared the results against the golden standard events for each use case. The results indicate that the estimators for each case yield the highest value when this case was actually the active case. The same observation holds for the fourth use case that is a sequential composition of the first three use cases. The experiments also indicated that the algorithm is stable in the sense that if the user adds some noisy or unrelated to the use case events as initial seed events the algorithm is still able to determine correctly the active use case.

Table 4 - Use Case Determination Results

Active Use Case	Reg. Estimator	Call Est. Estimator	Call Term. Estimator	Reg. + Call Est. + Call Term. Estimator
Reg.	**137**	13	56	62
Call Est.	36	**128**	68	91
Call Term.	70	8	**115**	22
Reg. + Call Est. + Call Term.	31	64	90	**121**

5.4 Stability Analysis

In order to test the stability of our approach, we repeated the above experiments, however this time using 2 different techniques. Both techniques try to change the way we use beacon events, in order to gain a better understanding of the strength and stability of the proposed framework, and also know the optimal manner in which beacon events are selected to achieve the best results. In the first technique, we systematically decrease the number of beacon events used for each scenario, until we reach 50% of the original number of beacon events as interpreted from the specifications. We then compute the precision and recall values and present the results. The second technique also removes beacon events for each scenario, but this time it also adds random events from the log file. Precision and recall results are also calculated.

Figure 18 and Figure 19 illustrate the precision and recall results obtained by running technique 1 using 2 scenarios: Registration + Call Failure and Registration + Call Establishment + Call Termination, respectively. The first pair of bars in each graph represents the original recall/precision values that we obtain by running the system using all beacon events. The rest correspond to the results obtained after removing beacon events in order. As expected, the recall drops significantly with the decrease in the number of beacon events. In Figure 18, for instance, the recall drops with a factor of more than 50%, even after the removal of only one beacon event. A similar pattern is observed in Figure 19, where recall drops, with the exception of the 6 beacon events case, more than 50%. The rest of the fluctuations in the recall values can be attributed to the heuristic nature of the clustering algorithm.

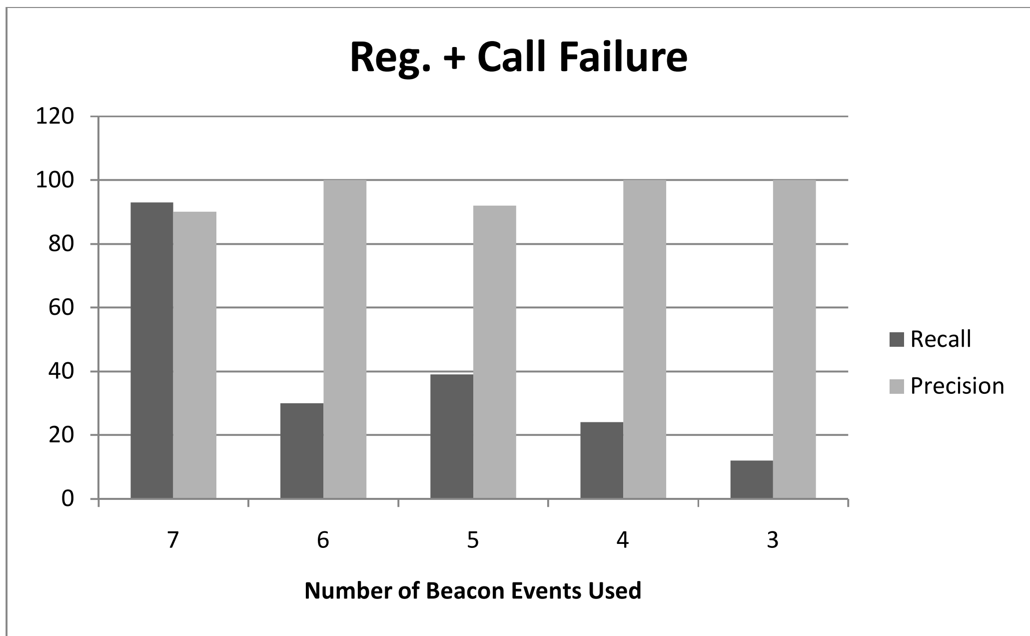

Figure 18 – Precision/Recall using stability technique 1

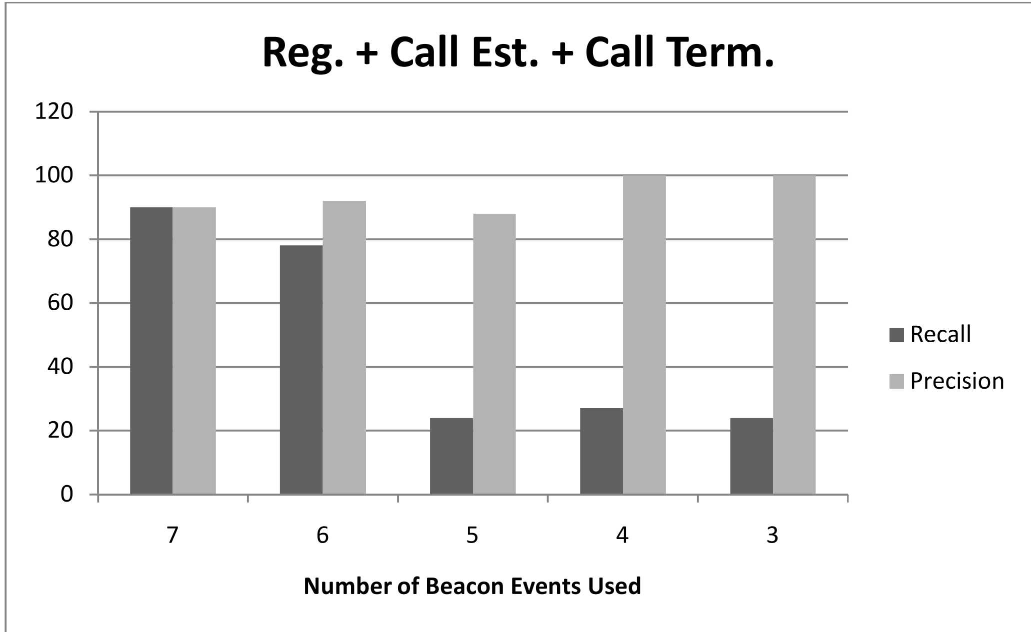

Figure 19 – Precision/Recall using stability technique 1

Surprisingly, however, the precision values are still very high. In Figure 18, all of the test cases have a precision value of above 80%. Even with approximately half of the original beacon events, we get 100% precision, which means that we don't have any noise in our cluster. Again, the pattern is repeated with Figure 19, with precision values all over 80%. This gives us a good advantage when trying to identify a running use case on the system, as we know for certain what scenario the events in our cluster pertain to, without, in most cases, any noise. The main reason behind these values seems to be due to the uniqueness of beacon events for each scenario. The beacon events selected where obtained from the specifications of the SIP communicator system, as indicated in the SIP RFC. As a result, each scenario has a unique set of signals/methods, in other words beacon events, making such precision possible. This also raises a vital point on the importance of selecting the beacon events for each case. In the case of the SIP communicator, we were able to compare the specifications against the RFC and identify those events. However, we believe that in any system, given the appropriate specifications, an expert user can identify all the important beacon events and as a result obtain satisfying results.

The second technique in our analysis aims at testing the robustness of the proposed approach by removing relevant beacon events from each scenario and adding random ones instead. Notice that in this case, the total number of beacon events per scenario remains fixed. Yet, the quality of the resulting cluster(s) will be affected due to the nature of the random events, and the kind of relations, if any, that exist between them and the actual beacon events. We applied this technique to the same scenarios presented above, namely the Registration + Call Failure and the Registration + Call Establishment + Call Termination scenarios. Figure 20 and Figure 21 show the precision and recall results obtained, and they clearly highlight the number of random events added in each case.

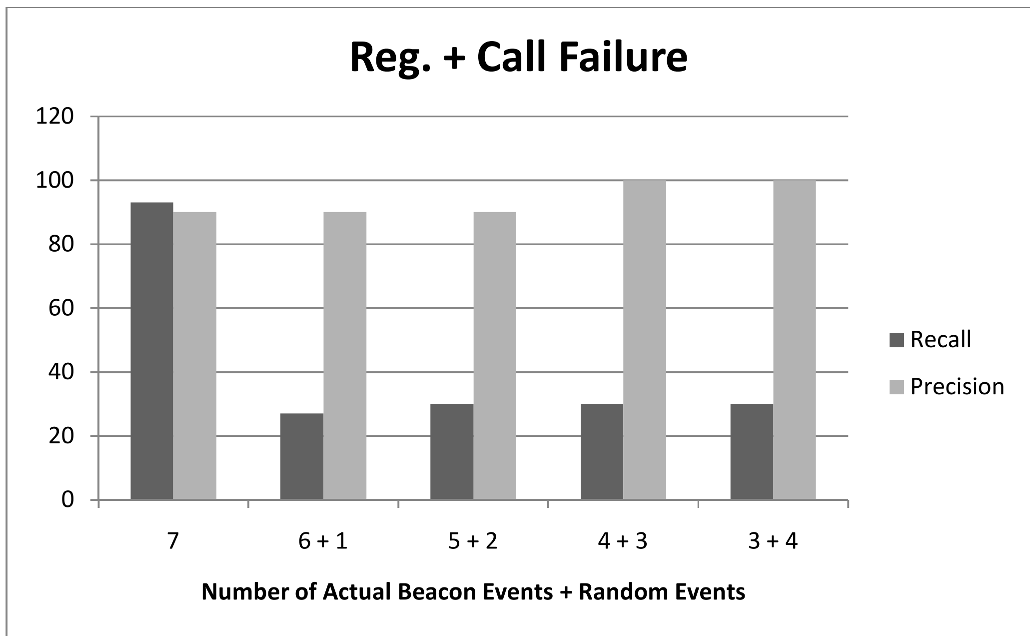

Figure 20 - Precision/Recall using stability technique 2

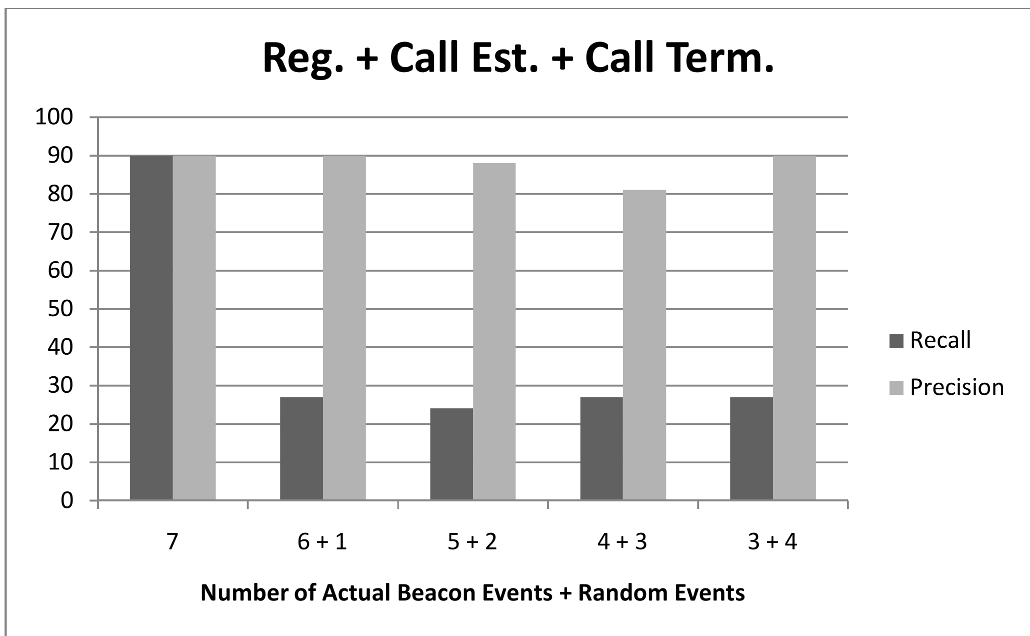

Figure 21 - Precision/Recall using stabillity technique 2

When we observe the above graphs, we notice a very interesting characteristic. In **all** cases, there is a considerable drop in recall values, even after the addition of only 1 random event to the cluster. In Figure 20, the highest recall value achieved with random events is 30%, which is 1/3 of the original recall we get with all the actual beacon events. The same results are also found in Figure 21, where recall values reach a high of 27%, even less than 1/3 of the original recall. Precision values, on the other hand, exhibit very similar values to the results obtained in technique 1. In both scenarios, precision values never dropped below 80% in all test cases, even after adding more than 50% random events to the original set of beacon events. As a result, use case identification can still be achieved with a high degree of accuracy.

These results, again, prove the importance of the beacon events selection process. By carefully selecting the relevant beacon events, we can successfully reduce large log files and be able to keep the important entries at the same time, allowing us to apply future analysis operations such as performing root cause analysis and other diagnostics. This also provides evidence for the validity of our log reduction technique. Using our algorithm, we were able to find the optimal criterion that strikes a balance between achieving significant log reduction and keep the data relevant at the same time.

5.5 Summary

In this chapter, we have presented the experimental results obtained by running our framework using two sets of data. The first was data collected using TPTP by running real life scenarios using the SIP communicator. The second was a collection of simulated log files that were automatically generated. The results were focused on 4 main areas, namely Timing, Filtering, Scenario Identification and Stability. Timing results illustrated how long it takes to create our Event Dependency Graph, and the time required to apply the clustering. We then presented precision and recall computations after the filtering process. Next we showed how our framework could be used to perform operations such as threat determination by presenting the scenario identification results obtained, concluding that we can always identify a running scenario with a 100% accuracy. Finally, a stability analysis was demonstrated to validate our approach and prove its robustness.

Chapter 6

Conclusions

In this chapter, we provide some concluding remarks on the work presented in this thesis. First, we discuss the contributions of this thesis. We then summarize our findings and results. Finally, we outline some opportunities for future research.

6.1 Contributions

The goal of this work was to come up with a new technique that would make the analysis of large volumes of log files easier. Due to their size, log analysis is very computationally expensive if not impractical in some instances. Our solution to this problem was to introduce a technique that would enable us to filter log files and achieve significant reductions in terms of size. However, we need to keep in mind that this reduction should not affect the important data contained within this log file, otherwise there is no point of performing the analysis. To address this, our technique is based on filtering log files with respect to a specific use case. By creating a model of event relations, the Event Dependency Graph, we are able to capture structural and behavioral associations among different

events in the log file. Then by identifying key events from this file, beacon events, which relate to the use case being considered, we run a clustering algorithm that groups relevant events together leading to a reduced log file size. In this context, the major contributions of the proposed solution are as follows:

- It defines the concept of the Event Dependency Graph that is formed by a collection of relations that aim to denote structural and behavioral associations between events in one or more log files.
- It introduces a novel technique to filter or slice logs, using a heuristic clustering algorithm, with respect to a particular use case. This enables system operators to better focus their attention to specific events that relate to specific operations.
- It proposes an approach for the determination of active use cases running on the system with a small number of initial seed events.
- The techniques presented in this work can be utilized to aid root cause analysis and system understanding.

6.2 Concluding Remarks

Many software maintenance tasks such as, root cause analysis or program understanding, require analysis of dynamic system information obtained from log files. However, for large systems this may be a computationally expensive and in some cases an intractable for practical purposes, process. A possible solution is to develop techniques to filter the logged events so that logs can be reduced in size and simplified in complexity and can be easier analyzed. In this context, this paper

discusses two event analysis approaches. The first approach aims to reduce log files to collection of events that are mostly related to a given use case. The approach is based on pre-processing logged events to form an Event Dependency Graph. The analysis of the graph is performed at per use case basis, with initial conditions in the form of beacon events. The result is obtained by considering all the events in the cluster that contains all initial beacon events. The benefit of the proposed approach is that in order to perform log filtering the users are required to have minimal information of the system being analyzed. The results obtained by analyzing various use cases in an implementation of the Session Initiation Protocol indicate that not only we can achieve high precision and recall values but also we can obtain a significant reduction in the number of events that we need to consider when examining the operation of the system for any given use case. The second approach aims to utilize the clustering technique to yield a cluster of events that can be compared against a golden standard of beacon events for various possible use cases of the system. The results indicate that the approach can be used to tractably identify in a running system the active use cases with a relatively small number of initial seed events.

6.3 Future Work

This work can be further extended in a number of ways. First, one could consider more innovative techniques to identify beacon events. These techniques could be based on the examination of the system configuration as well structural and deployment information. A second possible extension is the automatic or semi-automatic selection of the optimal set of relations to be used for clustering purposes at per use case basis. Lastly, since we used a heuristic hill climbing clustering algorithm, one could test our approach using other techniques such as genetic algorithms. Currently, we are experimenting with this technique to reduce logs emitted from Service Oriented systems. We are also

looking into ways to incorporate goal trees together with our use case determination technique to aid root cause analysis. The idea here is that we use a SAT solver on a goal tree to identify the possible failing paths of a program. Then, using our use case determination algorithm, we can determine which of these paths are actually executing and find out the failing sub-goals by elimination.

References

[1] E. Braun, D. Amyot and T. Lethbridge A. Hamou-Lhadj, "Recovering Behavioral Design Models from Execution Traces," in *Proceedings of the 9th European Conference on Software Maintenance and Reengineering*, 2005.

[2] T. C. Lethbridge and F. Lianjiang A. Hamou-Lhadj, "SEAT: a usable trace analysis tool," in *Proceedings of the 13th International Workshop on Program Comprehension*, 2005, pp. 157-160.

[3] M. B. Dwyer and G. Rothermel A. Kinneer, "Sofya: Supporting Rapid Development of Dynamic Program Analyses for Java," in *International Conference on Software Engineering*, 2007, pp. 51-52.

[4] T. Gottschalk and B. Luo B. Bruegge, "A framework for dynamic program analysis," in *Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1993, pp. 65-82.

[5] T. Gottschalk, and B. Luo B. Bruegge, "A framework for dynamic program analyzers," in *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1993, pp. 65-82.

[6] K. Driesen, L. Hendren and C. Verbrugge B. Dufour, "Dynamic metrics for java," in *Proceedings of the International Conference on Object Oriented Programming, Systems, Languages and Applications*, 2003, pp. 149-168.

[7] M. Traverso, and S. Mancoridis B. S. Mitchell, "An architecture for distributing the computation of software clustering algorithms," in *In The Working IEEE/IFIP Conference on Software Architecture*, 2001.

[8] D. Bell. (2004) UML's Sequence Diagram. [Online]. http://www.ibm.com/developerworks/rational/library/3101.html

[9] D. Bridgewater. (2004) Standardize messages with the Common Base Event model. [Online]. http://www.ibm.com/developerworks/autonomic/library/ac-cbe1/index.html

[10] K. Templer W. Zhou, and M. Brazell C. L. Jeffery, "A lightweight architecture for program execution monitoring," in *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop Program Analysis for Software Tools and Engineering*, 1998, pp. 67-74.

[11] Y. Cheon, "A runtime assertion checker for the Java modeling language," Iowa State University,

Technical Report 03-09 2003.

[12] B. Naveh and Contributors. JGraphT. [Online]. http://www.jgrapht.org

[13] Y. Ahmad, M. Balazinska, U. ¸ Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik D. Abadi, "The design of the borealis stream processing engine," in *Proceedings of the CIDR*, 2005.

[14] C. Fischer, M. Moller, and H. Wehrheim D. Bartetzko, "Jass-Java with assertions," *Electronic Notes in Theoretical Computer Science*, vol. 55, no. 2, pp. 1-15, 2001.

[15] U. C¸ etintemel, M. Cherniack, C. Convey, S. Lee and G. Seidman D. Carney, "Monitoring Streams - A New Class of Data Management Applications," in *Proceedings of the 28th VLDB Conference*, 2002.

[16] S. Mancoridis, and B.S. Mitchell D. Doval, "Automatic clustering of software systems using a genetic algorithm," in *Proceedings of Software Technology and Engineering Practice*, 1999.

[17] D. Drusinsky, "The Temporal Rover and the ATG Rover," in *Proceedings of the 7th International SPIN Workshop*, 2000, pp. 323-330.

[18] O. Greevy and S. Ducasse, "Correlating Features and Code Using a Compact Two-Sided Trace Analysis Approach," in *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, 2005, pp. 314-323.

[19] T. Richner and S. Ducasse, "Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information," in *Proceedings of the IEEE International Conference on Software Maintenance*, 1999, pp. 13-22.

[20] L. Belady and C. Evangelisti, "System partitioning and its measure," *Journal of Systems and Software*, vol. 2, pp. 23-29, 1981.

[21] Eclipse Foundation. Atlas Model Weaver. [Online]. http://www.eclipse.org/gmt/amw/

[22] Eclipse Foundation. Eclipse Test & Performance Tools Platform Project. [Online]. http://www.eclipse.org/tptp/

[23] M. Di Penta and M. Zazzara G. Antoniol, "Understanding Web applications through dynamic analysis," in *Proceedings of the 12th IEEE International Workshop on Program Comprehension*, 2004, pp. 120-129.

[24] S. Chodrow and M. Gouda, "Implementation of the Sentry system," *Software Practice and*

Experience, vol. 25, no. 4, pp. 373-387, 1995.

[25] The STREAM Group, "STREAM: The Stanford Stream Data Manager," Stanford InfoLab, Technical Report 2003.

[26] S. R. Tilley and K. Wong H. A. Muller, "Understanding software systems using reverse engineering technology perspectives from the Rigi project," in *IBM Centre for Advanced Studies Conference*, 1993, pp. 217-226.

[27] G. Camarillo, A. Johnston, J. Peterson, R. Sparks , M. Handley and E. Schooler J. Rosenberg H. Schulzrinne, "RFC 3261," The Internet Engineering Task Force, Available online at http://www.ietf.org/rfc/rfc3261.txt 2002.

[28] A. Hamou-Lhadj, "The concept of trace summarization," in *Proceedings of the 1st International Workshop on Program Comprehension through Dynamic Analysis*, 2005, pp. 43-47.

[29] R. W. Schwanke and S. Hanson, "Using Neural Networks to Modularize Software," *Machine Learning*, pp. 137-168, 1998.

[30] A. E. Hassan, "Mining Software Repositories to Assist Developers and Support Managers," in *International Conference on Software Maintenance*, 2006, pp. 339-342.

[31] V. Tzerpos and R. C. Holt, "ACDC: An algorithm for comprehension-driven clustering," in *Proceedings of the Working Conference on Reverse Engineering*, 2000, pp. 258-267.

[32] V. Tzerpos and R. C. Holt, "MoJo: A distance metric for software clustering," in *Proceedings of the Working Conference on Reverse Engineering*, 1999.

[33] V. Tzerpos and R. C. Holt, "On the stability of software clustering algorithms," in *Proceedings of the International Workshop on Program Comprehension*, 2000.

[34] IBM. Common Base Event. [Online]. http://www.ibm.com/ developerworks/library/specification/ws-cbe/

[35] A. E. Hassan and R. C. Holt J. Wu, "Comparison of Clustering Algorithms in the Context of Software Evolution," in *International Conference on Software Maintenance*, 2005.

[36] K. Templer and C. Jeffery, "A configurable automatic instrumentation tool for ANSI C," in *Proceedings of the 13th IEEE International Conference on Automated Software Engineering*, 1998, pp. 249-258.

[37] D. Li and X. Shi D. Li K. Di, "Mining association rules with linguistic cloud models," in

Proceedings of the Second Pacific-Asia Conf. on Research and Development in Knowledge Discovery and Data Mining, 1998, pp. 392-393.

[38] T. Mannisto, T. Systa and J. Tuomi K. Koskimies, "SCED: A Tool for Dynamic Modelling of Object Systems," Department of Computer Sciences, University of Tampere, Finland, Technical Report A-1996-4, 1996.

[39] K. Kontogiannis, and F. Mavaddat K. Sartipi, "Architectural Design Recovery using Data Mining Techniques," in *Proceedings of the IEEE European Conference on Software Maintenance and Reengineering*, 2000, pp. 129-139.

[40] N. H. Cohen and K. T. Kalleberg, "EventScript: an event-processing language based on regular expressions with actions," in *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, 2008, pp. 111-120.

[41] J. Kobielus, "Complex event processing: still on the launch pad," Computer World August 2007.

[42] K. Sartipi and K. Kontogiannis, "Component Clustering Based on Maximal Association," in *Proceedings of the Eighth Working Conference on Reverse Engineering*, 2001, pp. 103-114.

[43] S. Neginhal and S. Kothari, "Event Views and Graph Reductions for Understanding System Level C Code," in *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, 2006, pp. 279-288.

[44] A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte and W. White L. Brenna, "Cayuga: a high-performance event processing engine," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007.

[45] Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. Ru. M. Leino, and E. Poll L. Burdy, "An overview of JML tools and applications," in *Software Tools for Technology Transfer*, 2005, pp. 212-232.

[46] A. Lakhotia, "A unified framework for software subsystem classification techniques," *Journal of Systems and Software*, pp. 211-231, March 1996.

[47] N. Anquetil and T. Lethbridge, "File clustering using naming conventions for legacy systems," in *Proceedings of CASCON*, 1997, pp. 184-195.

[48] A. Hamou-Lhadj and T. Lethbridge, "Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System," in *14th IEEE International Conference on Program Comprehension*, 2006, pp. 181-190.

[49] A. K. Mok and G. Liu, "Efficient Run-Time Monitoring of Timing Constraints," in *Proceedings of the 3rd IEEE Realtime Technology and Applications Symposium*, 1997, pp. 252-262.

[50] D. Luckham, *The Power of Events.*: Stanford University Press, 2002.

[51] D. C. Luckham, "Rapide: A language and toolset for simulation of distributed systems by partial orderings of events," in *DIMACS Partial Order Methods Workshop IV*, 1996.

[52] K. Cockrell, W. G. Griswold and D. Notkin M. D. Ernst, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99-123, February 2001.

[53] M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky M. Kim, "MaC: A framework for runtime correctness assurance," University of Pennsylvania, Technical Report MS-CIS-98-37 1998.

[54] S. Mancoridis, G. Antoniol and M. Di Penta M. Salah, "Scenario-driven dynamic analysis for comprehending large software systems," in *Proceedings of the 10th European Conference on Software Maintenance and Reengineering*, 2006, pp. 71-80.

[55] J. Gargiulo and S. Mancoridis, "Gadget: A Tool for Extracting the Dynamic Structure of Java Programs," in *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*, 2001.

[56] A. S. J. Schiefer and C. McGregor, "Correlating events for monitoring business events," in *International Conference on Enterprise Information Systems*, 2004.

[57] F. Meyer, "Grey-weighted, ultrametric and lexicographic distances," in *Proceedings of the 7th International symposium on mathematical morphology*, 2005, pp. 289-298.

[58] Microsoft. CLR Profiler. [Online]. http://msdn.microsoft.com/ en-us/library/ms979205.aspx

[59] Sun Microsystems. Java Virtual Machine Profiler Interface. [Online]. http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html

[60] Sun Microsystems. Java Virtual Machine Profiler Interface. [Online]. http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html

[61] Sun Microsystems. Java Virtual Machine Tool Interface. [Online]. http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html

[62] Sun Microsystems. Java Virtual Machine Tool Interface. [Online].

http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html

[63] B. S. Mitchell, "A Heuristic Search Approach to Solving the Software Clustering Problem," Drexel University, PhD Thesis 2002.

[64] H. Muccini, "Using Model Differencing for Architecture-level Regression Testing," in *Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications*, 2007, pp. 59-66.

[65] J. M. Neighbors, "Finding reusable software components in large systems," in *Proceedings of the Third Working Conference on Reverse Engineering*, 1996, pp. 2-10.

[66] NIST. jain-sip: Java API for SIP Signaling.

[67] L. Wendehals and A. Orso, "Recognizing Behavioral Patterns at Runtime using Finite Automata," in *Proceedings of the 2006 international workshop on Dynamic systems analysis*, 2006, pp. 33-40.

[68] G. Antoniol and M. Di Penta, "A Distributed Architecture for Dynamic Analyses on User-Profile Data," in *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering*, 2004.

[69] G. A. Di Lucca and M. Di Penta, "Integrating Static and Dynamic Analysis to Improve the Comprehension of Existing Web Applications," in *Proceedings of the Seventh IEEE International Symposium on Web Site Evolution*, 2005, pp. 87-94.

[70] K. Driesen, D. Eng, L. Hendren, J. Jorgensen, C. Verbrugge and Q. Wang R. Brown, "STEP: A Framework for the Efficient Encoding of General Trace Data," *SIGSOFT Software Engineering Notes*, vol. 28, no. 1, pp. 27-34, 2003.

[71] K. Havelund and G. Rosu, "Monitoring Java programs with Java PathExplorer," *Electronic Notes in Theoritical Computer Science*, vol. 55, no. 2, pp. 97-114, 2001.

[72] J. J. Jeng and H. Chang S. K. Chen, "Complex Event Processing using Simple Rule-based Event Correlation Engines for Business Performance Management," in *The 8th IEEE International Conference on and Enterprise Computing, E-Commerce, and E-Services*, 2006.

[73] B.S. Mitchell, Y. Chen, and E.R. Gansner S. Mancoridis, "Bunch: A clustering tool for the recovery and maintenance of software system structures," in *Proceedings of International Conference of Software Maintenance*, 1999, pp. 50-59.

[74] B.S. Mitchell, C. Rorres, Y. Chen, and E.R. Gansner S. Mancoridis, "Using automatic clustering

to produce high-level system organizations of source code," in *Proceedings of the 6th International Workshop on Program Comprehension*, 1998.

[75] H. Safyallah and K. Sartipi, "Dynamic Analysis of Software Systems using Execution Pattern Mining," in *14th IEEE International Conference on Program Comprehension*, 2006, pp. 84-88.

[76] S. Choi and W. Scacchi, "Extracting and restructuring the design of large systems," *IEEE Software*, pp. 66-71, 1999.

[77] A. S. J. Schiefer, "Management and controlling of time-sensitive business processes with sense and respond," in *International Conference on Computational Intelligence for Modelling Control and Automation*, 2005.

[78] R. W. Schwanke, "An intelligent tool for re-engineering software modularity," in *Proceedings of the 13th International Conference on Software Engineering*, 1991, pp. 83-92.

[79] S. Sharma, *Applied Multivariate Techniques.*: John Wiley, 1996.

[80] C. Lindig and G. Snelting, "Assessing modular structure of legacy code based on mathematical concept analysis," in *Proceedings of the International Conference on Software Engineering*, 1997.

[81] T. Systa, "Understanding the Behavior of Java Programs," in *Proceedings of the Seventh Working Conference on Reverse Engineering*, 2000.

[82] J. Oberleitner and M. Pinzger T. Gschwind, "Using run-time data for program comprehension," in *11th IEEE International Workshop on Program Comprehension*, 2003, pp. 245-250.

[83] V. Tzerpos, "Comprehension Driven Software Clustering," University of Toronto, PhD Thesis 2001.

[84] S. Babu and J. Widom, "Continuous Queries over Data Streams," in *SIGMOD*, 2001.

[85] V. V. Kumar and I. R. Babu G. Prasad Y. Dhanalakshmi, "Modeling An Intrustion Detection System Using Data Mining And Genetic Algorithms Based on FUzzy Logic," *International Journal of Computer Science and Network Securtiy*, vol. 8, no. 7, July 2008.

[86] A. Adi, M. Barnea, D. Botzer and E. Rabinovich Y. Magid, "Application Generation Framework for Real-Time Complex Event Processing," in *32nd Annual IEEE International Computer Software and Applications Conference*, 2008, pp. 1162-1167.

[87] A. Zaidman, "Scalability Solutions for Program Comprehension through Dynamic Analysis," in

Proceedings of the Conference on Software Maintenance and Reengineering, 2006, pp. 327-330.

[88] H. Lee and B. Zorn. (Available online at http://www.cs.colorado. edu/hanlee/) ByteCode Instrumentation Toolkit.