Requirement-based Root Cause Analysis Using Log Data

by

Hamzeh Zawawy

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2012

Abstract

Root Cause Analysis for software systems is a challenging diagnostic task due to the complexity emanating from the interactions between system components. Furthermore, the sheer size of the logged data makes it often difficult for human operators and administrators to perform problem diagnosis and root cause analysis. The diagnostic task is further complicated by the lack of models that could be used to support the diagnostic process. Traditionally, this diagnostic task is conducted by human experts who create mental models of systems, in order to generate hypotheses and conduct the analysis even in the presence of incomplete logged data. A challenge in this area is to provide the necessary concepts, tools, and techniques for the operators to focus their attention to specific parts of the logged data and ultimately to automate the diagnostic process.

The work described in this thesis aims at proposing a framework that includes techniques, formalisms, and algorithms aimed at automating the process of root cause analysis. In particular, this work uses annotated requirement goal models to represent the monitored systems' requirements and runtime behavior. The goal models are used in combination with log data to generate a ranked set of diagnostics that represent the combination of tasks that failed leading to the observed failure. In addition, the framework uses a combination of word-based and topic-based information retrieval techniques to reduce the size of log data by filtering out a subset of log data to facilitate the diagnostic process. The process of log data filtering and reduction is based on goal model annotations and generates a sequence of logical literals that represent the possible systems' observations. A second level of investigation consists of looking for evidence for any malicious (i.e., intentionally caused by a third party) activity leading to task failures. This analysis uses annotated anti-goal models that denote possible actions that can be taken by an external user to threaten a given system task. The framework uses a novel probabilistic approach based on Markov Logic Networks. Our experiments show that our approach improves over existing proposals by handling uncertainty in observations, using natively generated log data, and by providing ranked diagnoses. The proposed framework has been evaluated using a test environment based on commercial off-the-shelf software components, publicly available Java Based ATM machine, and the large publicly available dataset (DARPA 2000).

Acknowledgements

I would like to thank my supervisor Kostas Kontogiannis whose supervision, support, enthusiasm and vision have been of great value to me all along my PhD journey and enabled me to develop the present thesis.

I am deeply grateful to my supervisor, John Mylopoulos, for his guidance and patience. Professor Mylopoulos' important support throughout this work was essential for the success and completion of this thesis.

I would like to also thank my supervisor Paul Ward for his extensive discussions around my work. I owe my most sincere gratitude to Professor Ladan Tahvildari and Professor Michael Godfrey for their detailed and constructive comments.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Root cause analysis (hereafter RCA) refers to the identification of errors that lead to failures in a software system. By *failure* we mean any deviation of the system's observed behavior from the expected one, while by *error* we refer to a system bug or internal misconfiguration.

Overall, the RCA problem is a difficult and challenging task to solve. The software engineering community has responded by proposing different types of techniques for addressing the problem. One class of techniques is based on the association of event patterns with known system errors when a failure is observed. Another class of techniques is based on collections of rules that apply diagnostic expert knowledge. A third class of techniques is based on model driven root cause diagnosis where a model of the system is built and an analysis process attempts to identify the components or the conditions which can explain an observed failure.

1.1 Motivations

Over the recent years, unprecedented increase in demand for processing and storage resources led to a growth of enterprise IT systems at phenomenal rates. For example, the server count at *Facebook* grew from 30,000 to 60,000 in just six months during 2010. *Intel* is estimated to have at least 100,000 servers, while widely circulated estimates put the number of *Google* servers at 450,000 servers [30]. To maintain service quality at agreeable levels, IT systems are constantly monitored by system administrators. The size of the monitored system has direct impact on the size of the human workforce needed to maintain it.

Another factor impacting the number of staff needed to maintain IT environments is the heterogeneity of these environments. For stateful systems (e.g., databases, application servers), it takes approximately one administrator for each terabyte of data [40]. On the other hand, stateless applications (e.g., Web servers) can be scaled through replication and one administrator can manage hundreds of them. In a hybrid environment, a rule of thumb is to assign one administrator on average per 100 terabytes of data or per 20 servers. As an example, current systems at *Walmart* process more than 1 million customer transactions per hour generating more than 2.5 petabytes of data. At the pace of current growth of IT systems, staff costs for maintaining IT systems will be prohibitively expensive.

Furthermore, the sheer size of log data often makes manual analysis intractable. It is estimated that 40 percent of large organizations collect more than 1 Terabyte of log data per month, where 11 percent collect more than 10 Terabytes of log data monthly [40]. Systems at *Facebook* generate 25 Terabytes of log data daily. The sharp rise in the number of servers owned per enterprise and the size of the log data generated by these systems clearly necessitates an automated support for operators and administrators. A key challenge for enterprises is to decrease the administrator to server ratio; however, this ratio depends on how much automation can be rolled into the maintenance of a server farm. Thus, a challenge in this area is to provide the necessary concepts, tools, and techniques to automate the diagnostic process.

1.2 Problem Statement

The challenges in automating RCA in software systems stem from the complexity of modeling the interdependencies between system components, the often incomplete logged data that could fall short explaining the cause of a failure and, the lack of models that could be used to denote conditions and requirements under which a failure could be manifested. The problem becomes even more complex when the failures are not due to internal system errors, but due to external actions by third parties that intentionally or unintentionally cause the failure of a software system.

The complex interdependencies among the different components in heterogeneous software systems make the modeling of such systems a challenging task. In addition, the management of large system models presents challenges related to consistency, completeness, maintenance, and reuse. Furthermore, the lack of accurate models that represent the conditions under which fault/failures can occur further complicates the automation of the RCA process. Most modeling approaches are not suitable for modeling concurrent failures, latent (not yet known) failures or failures involving subtle timing issues. Attack models, representing the alternative manifestations of malicious behavior, suffer from similar disadvantages especially when it comes to addressing unexpected security vulnerabilities.

Typically, in large enterprise environments, the log data of interest is scattered in different files, over various machines, may have partially captured significant events, and/or may not include enough information for diagnosis. Our experimental evaluation of the quality of log data generated by a set of commercial-off-the-shelf software systems indicated that, although most native log data formats contained "essential" fields such as description and timestamps; other natively generated log data were sparse. For example, log data generated by systems such as *Apache Web Server* contained basic information such as the source system's IP address, timestamp, request line and user id; others such the *IBM WebSphere MQ* included more information including the process id, detailed description, cause, etc. Furthermore, the completeness of log data and its coverage of all events of interests varied among the set of studied systems. The log data generated by some systems such the Microsoft SQL Server 2008 contained granular events such as the addition of one row to a database table, while others only reflected macro events such as system start-up, system shutdown, etc.

1.3 Approach

In this thesis, we take a novel approach to RCA by using a probabilistic based inference combined with some of the best features of existing types of RCA techniques. More specifically, we utilize model driven approaches to denote not only the conditions, constraints, actions and tasks, a system is expected to enact in order to deliver its functionality but also, the conditions, constraints, actions and tasks that can be taken by an external user to invalidate or threaten a given system action or task. We refer to the first type of models as goal models and, the second types of models as anti-goal models. Second, we utilize pattern based approaches to limit the size of the software logs that can be considered as observation supporting or denying the occurrence of a specific system goal or task. Third, we built a probabilistic diagnostic knowledge base by denoting goal and anti-goal models as weighted diagnostic rules in the form of a Markov Logic Network. Thus, when a failure is observed, the corresponding goal model is analyzed to identify the tasks and actions that may explain the observed failure and the anti-goal model is investigated to increase the confidence of the goal model analysis. The confidence increase is manifested when a goal model task or action fails while its corresponding anti-goal succeeds. Confidence levels are computed using Markov Logic Network inference as a function of the reasoning process combined with past observations that provide a level of training for the reasoning process.

1.4 Motivating Example

Throughout this thesis, we use as a running example the RCA for a failed execution of a loan evaluation business process implemented by a service oriented system. The normal execution scenario for the system starts upon receiving a loan application in the form of a Web Service request. The loan applicants' information is extracted and used to build another request that is sent to a credit evaluation Web Service. The credit rating of the applicant is returned as Web Service reply. Based on the credit rating of the loan applicant, a decision is made on whether to accept or reject the loan application. This decision is stored in a table before a Web Service reply is sent back to the front end application. The motivating scenario is implemented as a proof of concept and includes 6 systems/services: the front end application (soapUI), a process server (IBM Process Server 6.1), a loan application business process, a message broker (IBM WebSphere Message Broker v7.0), a credit check Web Service and an SQL server (Microsoft SQL Server 2008).

1.5 Contributions

The aim of this thesis is to assist in the automation of RCA for software systems by providing a set of tools and techniques for the enterprise system administrators to focus their attention, in the event of a system failure, on the tasks and components that could be at the source of the observed failure. The contributions of the thesis are summarized as follows:

1. The root cause framework described in this thesis improves over existing approaches by using a probabilistic approach based on Markov Logic Networks. Furthermore, this framework does not require the instrumentation of the monitored systems and uses the natively generated log data as the basis for observation. Our approach uses weighted formulas instead of hard constraints allowing for conflicts such as when a goal/task is satisfied and denied at the same time. Traditional approaches for RCA based on conventional logic depend on accurate observation and their analysis suffers when using potentially incomplete and inaccurate heterogeneous log data. Using a probabilistic approach allows for the handling of uncertainty in observations and resolving any resulting conflicts. Thus, monitoring and diagnosis is applied by relying

on natively generated log data and without the need for instrumentation which is intrusive and usually impractical when analyzing systems consisting of off-the-shelf commercial sub-systems.

2. This framework generates a set of ranked diagnostics consisting of tasks suspected of being causes of the failure. The weight of each diagnosis represents the likelihood of a task/component contained in the diagnosis to be the root cause for the failure. Finally, this framework has the advantage of being able to diagnose multiple simultaneous faults because the generated diagnostic include the set of all tasks/components that did not behave according to the corresponding requirement goal model.

3. The use of annotated requirement goal models to represent the runtime behavior of the monitored systems by modeling the complex relationships between their different components. Goal models are well known formalisms used for capturing functional and non-functional system requirements. Although the use of this class of models to represent the runtime behavior of software applications is not novel, one of the contributions of this thesis is the use of pattern language based annotations to enrich these models in order to support diagnosis. This work also applies a similar process of model enrichment to anti-goal models which are used to model malicious behavior against the monitored system. By annotating anti-goal models, the diagnostics analysis can be extended to produce evidence if the failed tasks that triggered the overall system failure are in fact the result of malicious behavior.

4. Another contribution of this thesis is the evaluation of different information retrieval techniques and their applicability to the problem of log reduction and filtering. We experiment with semantic document ranking techniques such as latent semantic indexing as well as syntax-based information retrieval techniques such as *Grep* and standard SQL queries.

5. Finally, we propose a common log schema in order to address the heterogeneity problem caused by different log data formats used by each logger. In large systems, different logging mechanisms are used by subsystems; thus, the generated logged data comes in different formats. The identification of the mappings between the native log schema of each monitor component and the unified schema can be compiled using semi-automated techniques discussed in detail by Alexe et al. [5] or compiled manually by domain experts. For the purposes of this study, we have implemented the mappings manually, as tables. The proposed unified schema for this work contains 10 fields classified into four categories: general, event specific, session information and environment related. This proposed schema represents a comprehensive list of

data fields that we consider to be useful for diagnosis purposes. After the log data is transformed into a unified format, it is stored in a centralized database. In a typical industrial setup, millions of log data entries can be generated daily, thus careful attention should be done to avoid performance and storage issues.

1.6 Thesis Organization

Figure 1.1 shows the three main topics in this thesis: modeling, log data reduction/filtering and diagnosis. Chapter 3 is dedicated to the modeling component, while Chapters 4 and 5 discuss the log reduction and filtering techniques. Chapters 6 and 7 describe two approaches that we adopted for RCA: deterministic and probabilistic. In the following paragraphs, we provide a detailed outline of the thesis:

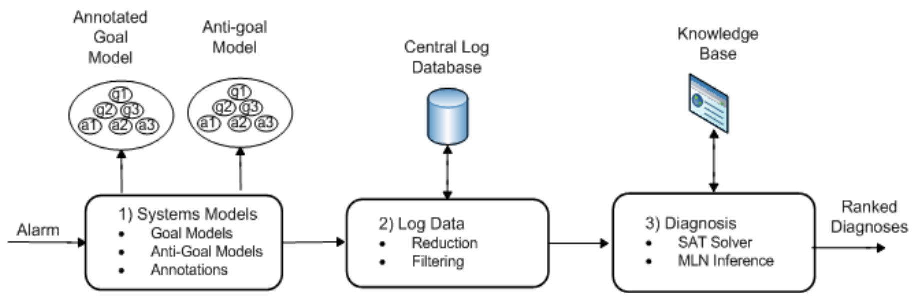


Figure 1.1: The Thesis Components

In Chapter 2, we present and discuss the related work and provide a comparison to the work described in this thesis. Next, Chapter 3 outlines our research baseline and describes the formalisms used in the thesis. In particular, this Chapter focuses on the syntax and semantics of the goal and anti-goal models used to represent the monitored applications as well as the formalism used to describe the expressions annotating these models. We also present a motivating use case scenario based on a loan application that we use as a running example throughout the thesis.

In Chapter 4, we present and compare two log reduction approaches aimed at reducing the amount of log data generated in a distributed environment while keeping those entries that are deemed useful by the system administrator/user. As a case study, we use a live network of service oriented distributed systems in addition to a publicly available large dataset (DARPA 2000).

In Chapter 5, we present two log filtering approaches. The first is based on applying the latent semantic indexing approach to log data. In the second approach, we present a hybrid approach for log data filtering based on merging regular expressions with latent semantic indexing.

In Chapter 6, we extend and evaluate an existing framework for RCA based on SAT solvers. The extension focuses on using natively generated log data as system observation by using filtering techniques and models described in the previous chapters.

In Chapter 7, we introduce a probabilistic approach for RCA based on Markov Logic Networks and using requirement goal models to represent monitored applications. This approach is evaluated using the publicly available ATM machine case study. Furthermore we extend and apply this RCA framework to detect another class of failures, mainly those caused malicious behavior. We use anti-goal models to describe malicious behavior executed by third parties in order to disrupt the normal operation of the monitored systems.

In Chapter 8, we discuss some of the practical aspects related to this research work propose. We describe a unified logging approach that aims at optimizing the contents and size of log data and providing guidelines for when logging should be applied.

We conclude this thesis in Chapter 9 with a discussion of future work and a set of conclusions.

Chapter 2

Related Work

In this chapter, we discuss related work in the areas of RCA, trace analysis, complex event processing and log generation in software systems.

2.1 Root Cause Analysis

In this thesis, we develop a set of tools and techniques that can assist in the automation of RCA in large enterprise environments. In general, RCA (also known as fault localization) represents a class of techniques for the detection and identification of the concealed fault(s) that is at the source of a system failure or a user reported incident. Prior to the advent of service oriented architecture (SOA), most of the literature on RCA of software systems has been in the area of low level communication systems; however, the advent of SOA has led to more interest in developing and adopting RCA techniques in higher layers such as business processes and software services layers.

The approaches to RCA in software systems can be classified into the following three categories: rule-based, probabilistic and model-based. Some approaches rely on natively generated low level trace information [49, 15] as a source for system observation. Others generate their own log data using instrumentation [104] or by intercepting system calls [110]. In the following sections, we describe each of these approaches and compare them to the approach used in this thesis.

2.1.1 Rule-based approaches

This class of RCA approaches uses rules to capture the domain knowledge of the monitored IT environment. Such rules are typically of the form <*If symptoms then diagnosis*> where symptoms are mapped to root causes. This set of rules is built by interviewing system administrators or automatically using machine learning methods using existing empirical data. To find the root cause of an incident, the set of rules are searched for matching symptoms. When a match is found and sufficient conditions of a rule are satisfied, the rule is triggered and a conclusion for the root cause of the failure is derived.

There are multiple techniques to represent the rules such as using decision trees [15] or rule sets [49, 45] or decision matrices [71, 72]. Miyazawa et al. [68] propose an approach based on a decision tree to detect root causes for failures in telecommunication systems. This approach consists of building an hierarchical alarm information model to classify alarms based on their types of failures. When an alarm is raised, the RCA framework first collects enough information that allows it to classify the alarm according to the hierarchical information model. Finally, related alarms are identified and the root cause component is identified.

The decision-based approach by [71] is more sophisticated and requires building two decision matrices: the first matrix represents relationships between the different monitored components; the second matrix represents the calendar changes of the system. By matching the time of the current incident to the closest (time-wise) system change using the second matrix, the incident is correlated with the change. Using the first matrix, a ranked list of the components associated with the change/incident is identified. One of the drawbacks of the aforementioned approaches is that the preparatory work required to identify the root cause for the failure is not trivial and need to be repeated for new incidents. On the other hand, the work by [53] requires building a fishbone by looking into historical data and identifying common incidents. For each common incident, a fishbone is built by associating with it the high level reasons that can cause this incident such as process, people, technology, etc. thus building the fishbone main skeleton. Next, a set of detailed reasons is associated with each class of causes thus building short lines for each bone. The result is a fishbone that can be used to classify new incidents. Jalote et al. [53] uses this approach to detect defects in iterative software developments.

Hanemann [45] proposes a hybrid reasoning approach where the root cause of an incident is identified by first searching a set of rules that map symptoms to root causes. If no matching symptom is found, this incident is treated on a case based approach by manually resolving it the first time it is encountered, and then storing it for future reference. Yuan et al. [110] use system behavior information such as system event traces to build correlations with already solved problems. To diagnose a reported problem, a subset of the trace is collected and compared to previously collected sets of trace data that is already mapped to symptoms. Based on the result of the comparison, the problem is classified and a potential root cause is identified. The limitation of this approach is that the automatic identification of root causes can only be applied for known problems.

In general, the common drawback of the rule-based RCA approaches is that it is hard to have a comprehensive set that covers all the needed rules. Furthermore, the overhead work needed before these approaches can be used in practice is not trivial and does require to be updated for new incidents. This is an advantage for our approach which also require preparatory work but (unless the system configuration has changed) as in the case using rule-based approaches.

2.1.2 Probabilistic Approaches

Traditional RCA techniques assume that the system components inter-dependencies are known with certainty or that the information about the monitored system state is accurate and complete, which is not always correct. Probabilistic approaches models the cause and effect relationships between IT systems using graphical models where edges represent relationships and nodes represent components. In this context, an edge from nodes x to y shows that a problem in x could cause a problem in y. Moreover, the strength of the relationships is modeled through probabilities.

An example of using the probabilistic approach is the work of Steinder et al. (2004) [93]. Steinder et al. use Bayesian networks to represent dependencies between communication systems and diagnose failures while using dynamic, missing or inaccurate information about the system structure and state. The advantage of our approach over Steinder's approach is that, using first order logic, we are able to model complex relationships between the monitored systems and not just simple one to one causality relationships. Moreover, our approach is similar in that we use a probabilistic model (Markov networks (undirected) as opposed to Bayesian Network (directed)) in order to fit the evidence. However, our approach is similar to [93] because it also requires preparatory work of modeling the system to be monitored before the RCA framework can be used. Xing et al. [107] address the problem of detecting root causes for common-cause failures which are simultaneous failures of multiple components within a system due to a common-cause or a shared root cause. The approach of Xing et al. is a hybrid one of using dynamic fault trees and a probabilistic analysis. In particular, this approach consists of three steps: first, modeling the common cause event as being a basic event contributing to the failures of a set of components affected by that common cause of failure. The outcome of this first step is to build the set of components that fail together due to common causes. Second, a set of conditional probability that the system fails given that the different sets of components (found in the first step) fail. This step takes advantage of the dynamic fault trees. Finally, the probabilities of the common cause event occurring are calculated and a root cause is identified.

Another example is the approach by Al-Mamory et al. [4]. Al-Mamory et al. makes use of a history of alarms and resolutions in order to improve future root cause analysis. In particular, Al-Mamory et al. use RCA to reduce the large number of alarms by finding the root causes of false alarms. This approach uses data mining to group similar alarms into generalized alarms and then analyze these generalized alarms and classify them into true and false alarm. Later, the generalized alarms can be used as rules to filter out false positives. Al-Mamory et al. assume that log data comes in a unified form that contain fields such as IP (source, target), port, etc. Next, hierarchies are built by classifying the possible enumerations (e.g., internal versus external or DMZ IP addresses). Next, clustering is applied using a distance function that relies on the already built hierarchical trees to calculate distances. The result is a set of clusters that are represented using generalized alarms. The generalized alarms follows the same pattern as any alarm but the values set inside the generalized alarm are values taken from the hierarchical trees (so if ports are 80, 90, etc. then they are grouped as [80-1023], and IP addresses are grouped as DMZ, Internal, External, etc.). The experimental results show a reduction of 70 percent.

2.1.3 Model-based Approaches

This class of RCA techniques consist of building a model representing the normal behavior of the system, and detecting anomalies when the monitored behavior does not adhere to the model. An example of this approach is the approach by Wang et al. which is based on the propositional satisfaction of system properties using SAT solvers [104]. More specifically, given a propositional formula f, the propositional satisfiability (SAT hereafter) problem consists of finding values for the variables of the propositional formula that can make an overall propositional formula evaluate to true. The propositional formula is said to be true if such a truth assignment exists. A SAT solver is a procedure that determines the satisfiability of a propositional formula, identifying the satisfying assignments of variables. Most SAT solvers use a Conjunctive Normal Form (CNF hereafter) representation of the Boolean formula f. In CNF, the formula is represented as a conjunction of clauses, each clause is a disjunction of literals, and a literal is a variable or its negation. Note that in order for the overall formula f to be satisfied, each clause must evaluate to true. Our approach introduces a probabilistic diagnostic component allowing the handling of inaccuracies in modeling the monitored IT systems dependencies as well as missing and inaccurate observations. We use weighted formulas instead of hard relationships in the case of [104]; thus, allowing for conflicts (when a goal/task is both satisfied and denied at the same time). Alternatively, the occurrence of conflicts in a deterministic approach such as [104] would render a world of hard rules impossible, thus resulting in no diagnostic. Our work is a hybrid approach combining the probabilistic approach with the model-based approach.

2.2 Trace Analysis

Trace analysis aims at analyzing computer-generated log data in order to provide an audit trail. Audit trail is defined as a sequence of steps documenting the real processing of a transaction flow. The work described in this thesis applies trace analysis techniques onto the log data generated by a monitored system to assist in producing a diagnostic on this system's health. Generally, trace analysis is performed to enforce security, to assist on compliance with respect to regulations, standards or, system requirements, or to perform RCA.

In order to extract information from the log data, there exist a set of tools commonly used in the industry based on a variety of approaches such as pattern matching, clustering, semantic analysis, etc. One of the traditional techniques is Grep. Built in 1973, Grep is a simple yet efficient tool that uses a rich regular expression language [7]. Structured Query Language (SQL) is also a common tool used to index and retrieve log data stored in databases. SQL have the advantage over Grep that allows for logical joins. Logzilla and Sawmill are based on an enhanced SQL and provide graphical reporting [29, 85]. Splunk is a log search and analysis tool. The search language used by Splunk is based on simplified syntax similar to command line type of syntax. It reduces the expression building complexity by using terms defined in configuration files to represent the underlying system details.

The academic literature on trace analysis techniques can be classified according to the reason for performing the trace analysis: security, compliance and RCA. In the security related literature, log analysis was used by Denning [25] to present a real-time intrusion detection model based on profiling log data and detecting abnormal patterns of system usage. Shieh et al. [86] present a system to detect anomalies in operational security based on models that represent various patterns caused by the unintended use of programs and data. In [6], Andrews proposes a state machine based log analysis framework with applications in unit and system testing. In [98], Vaarandi presents a data mining algorithm for profiling and pattern recognition in log file data sets. Cho et al. [16] present an intrusion detection system based on hidden Markov models representing the normal operation of a system. The system incorporates a soft computing based analysis technique to detect intrusions by noting significant deviations from the model.

Log analysis is used for business compliance purposes typically by storing events in an intermediate storage medium which are then subject to human queries and/or data mining processes [76].

Jiang et al. discuss log analysis applications for RCA and troubleshooting purposes and conclude that relying on a single failure message is a poor indication of root cause failure but aggregating messages can increase the accuracy by a factor of three [54]. Wang et al. [104] instrument the source code of monitored applications by linking the nodes of a goal models that represent the monitored applications to specific points in the monitored application's source code; next, the generated log data is used to verify the requirements satisfaction as well as for system diagnostics. Similarly to [104], we use goal models to represent the monitored systems [113]. Moreover, we use annotations to semantically link the log data to the goal model nodes and then build an audit trail by extracting log data using the pattern expressions embedded in the goal model annotations.

2.3 Complex Event Processing

Complex event processing (CEP) is a set of tools and techniques for analyzing and controlling the complex series of interrelated events that drive modern distributed information systems [65]. CEP, also known as stream or data stream processing, involves first, the identification of patterns, sequences or relationships among streaming events, and next taking subsequent action in real time. In this context, an event represents something that happened in the real world (e.g., stock price change, sensor reading). A complex event is an aggregation of two or more events with temporal (e.g., after 2 minutes) or spatial (e.g., within 1 meter) constraints. In practice, a complex event processor is an engine that operates on data streams by performing complex real-time queries that represent patterns of interest such as attack scenarios, KPI measurements for business processes, or movement detection in sensor networks, etc. CEP is applied to a broad range of applications ranging from intrusion detection, IT system management, business process management, sensor networks, etc.

Luckham introduced CEP in his book titled "The Power of Events" [65] which focused on the identification of complex sequences of events and the control actions that occur in response to these sequences. The explosion in networking and the advent of middleware has led to an exponential increase in the use of messaging and event processing. Real-time event processing of continuous data streams required high performance querying which can't be delivered by traditional databases. This resulted in a new generation of databases that support continuous queries optimized for processing sequential patterns in stream data [94]. Performance gains in stream databases are achieved by parallelizing these systems and limiting the expressiveness of the corresponding query languages. Examples of these systems are Aurora [13], PIPES [14], STREAM [97] and Borealis [12].

Full CEP systems such as *Cayuga* [96], developed at Cornell University, use the publish-subscribe pattern and are built on top of stream databases. In addition, these systems support temporal event processing over windows, and allows for the handling of complex events. Using *Cayuga*, event patterns can be detected in unrelated event streams. *Cayuga* has an advantage over other CEP systems in that it supports re-subscription, thus each query output is produced in real time and thus can be used as input to another.

Another CEP system is *Esper* [32]. *Esper* supports conditions with temporal nature, joining of different event streams, as well as filtering and sorting them. *Esper* includes an event processing language (*EPL*) that allows for the expression of rich event conditions, correlation that span over time windows. *EPL* statements aggregate information from one or more streams of events, to join or merge event streams, and to use results from one event stream to subsequent statements. Similarly to traditional SQL, *EPL* uses the *Select* clause and the *Where* clause. On the other hand, *Esper* uses event streams instead of traditional tables. Similar to traditional databases, *Esper* support the concept of views. Views can represent windows over a stream of events and can also be used to sort events, derive statistics from event properties or group events. The following is a sample *EPL* statement to compute the average price for the last 20 seconds of stock tick events:

```
select avg(price) from StockTickEvent.win:time(20 sec)
```

Another example of EPL is a query that returns the average price per symbol for the last 1000 stock ticks:

```
select symbol, avg(price) as averagePrice
    from StockTickEvent.win:length(1000)
    group by symbol
```

A third example of EPL consists of joining 2 event streams. The first event stream consists of fraud detection events for which we keep the last 20 minutes (1200 seconds). The second stream contains withdrawal events for the last 20 seconds. The streams are joined based on account number.

```
select fraud.accountNumber as accntNum, fraud.warning as warn,
    MAX(fraud.timestamp, withdraw.timestamp) as timestamp,
    'withdrawlFraud' as desc, withdraw.amount as amount
    from FraudWarningEvent.win:time(20 min) as fraud,
    WithdrawalEvent.win:time(20 sec) as withdraw
    where fraud.accountNumber = withdraw.accountNumber
```

2.4 Logging in Software Systems

In general, computer systems generate log data to report errors, trigger alerts, or collect information for audit purposes. Before extracting any useful knowledge from the log data, most trace analysis approaches require a set of preprocessing steps such as transforming the natively generated log data into a common format before it can be used for analysis. This preprocessing step is necessary due to the lack of a logging standard for software systems across industries.

However, there is a set of a per-industry or per-vendor logging standards. For example, the Web Services Distributed Management Event Format (WEF) [73] is used by service oriented systems while the intrusion detection message exchange (IDMEF) [23] is designed (but not widely adopted) for security systems. As for the vendor logging standards, we mention the Common Base Event (CBE) [50] by IBM, Syslog for UNIX systems, log4j for Java based applications, and the system, security and application event logs for Windows based systems.

On the academic side, there were efforts to propose guidelines for logging. Salfner et al. proposed a set of guidelines to make log data more useful for autonomic computing [83]. In his proposal, Salfner suggests that log records should include 5 categories of information: general (e.g., timestamp), structural (e.g., component ID), runtime/configuration related (e.g., process ID, software version, etc.), resource related (e.g., source of the event) and event specific (e.g., method that created the event). Each log record is populated using *key=value* pairs of data, and each data type is explicitly identified (e.g., by quoting text, prefixing hexadecimal with 0x). Each record is uniquely identified based on a record id generated using a hierarchical numbering scheme.

In [74], Oliner et al. presents a study of system logs generated by a set of 5 super-computers. This study describes the methods of log collection used in these systems, and proposes a filtering algorithm, and provides a summary of missing information.

Gunter et al. [42] provides a summary of existing logging approaches and argues that log data are commonly modeled as natural language sentences and that a standardly formatted, well-structured and strongly typed model is needed. Gunter et al. also propose a system model that include contextual information that explicitly indicate if the log entry is generated by the event of an activity starting, ending or failing.

To reduce the impact of logging on the monitored application performance, Borghetti et al (2009) suggest a method for dynamic tracing [11]. This method consists of a preliminary phase of discovering a software application's critical patterns using existing run-time analysis and statistical analysis tools. In this context, a critical pattern is the complex path inside the software code that impacts the functionality of the overall system. Next, only all events identified to be coming from outside the critical path are filtered out. Our approach is similar to [11] in terms of limiting the size of the trace file by capturing only events relevant to goal models representing the monitored systems.

Cinque et al. [17] proposes a set of logging rules to be followed at design time in order to improve the quality of log data and simplify the process of log reconciliation. The proposed logging rules did not only describe the format of the log record but also included a system model to provide guideline on when to log. Cinque et al. classify the systems in an environment into entities (processes, threads, etc.) and resources (database, web server, etc.) that interact with each other and suggested to log the start/end of each process and the start/end of each interaction between processes and resources. Similarly to [17], we suggest in this thesis a logging approach that proposes how and when to log based on our experimentation with the log format of a set of commercial software. The limitation of the model proposed by [17] is that it only allows for a very basic set of event types to be logged (service or interaction start/end). The model proposed in this thesis uses the goal models as a guideline to describe what events of interest need to be captured and logged. Such models are built by analysts as part of the requirement gathering and analysis phases or by reverse engineering the source code using techniques well documented in the literature [109].

2.5 Summary

In this chapter, we discuss the related work to the thesis. First, we discuss and provide a classification of the root cause analysis approaches applied to software systems. Next, we look into the trace analysis tools and techniques used to extract audit trail in software systems. Finally, we compare the software logging styles and schema and discuss their advantages and disadvantages.

Chapter 3

Research Baseline

In this chapter, we describe the foundations that our research work is based on. First, we provide an overview for the framework for the root cause analysis of software systems presented throughout this thesis. Next, we describe the requirement goal model and its applications in this thesis as a formalism for modeling the monitored software systems at runtime. Moreover, we describe extensions and annotations that we apply to the goal models in order to enrich it for the purpose of root cause analysis. Next, we describe anti-goal modeling which is a variation of goal modeling and is used to describe malicious behavior against requirements goals and tasks. We also provide an overview of the semantic analysis as well reasoning techniques used later in this thesis. Finally, we describe a use case scenario and a test environment that is used throughout the thesis.

3.1 Root Cause Analysis Framework Overview

In this section, we outline the root cause analysis process describing the main steps in this process as well as the fundamental concepts that this root cause analysis are based upon. As shown in Figure 3.1, the root cause analysis framework consists of three phases: modeling, observation generation and diagnostics. The first phase is an offline preparatory phase. It consists of generating a set of models to represent the runtime behavior of the monitored applications. For this we choose the requirement goal modeling (described later in section 3.2) as the formalism to represent the monitored applications. In addition, we annotate the nodes (goals/tasks) in the goal tree models with additional information that represent the precondition, postcondition and the occurrence of each of these nodes. The extra information is essential in order to extract the observation from the log data in the second phase of this RCA framework. The second phase is started when an alarm is raised indicating the monitored application has failed. In this phase, the goal model annotations are used to build queries that applied using a variety of techniques in order to reduce the log data size and extract an observation in the form of an audit trail. This observation is used, along with the goal and anti-goal models, to build a knowledge base. In addition, we use this observation in a feedback loop to improve the quality of the goal/anti-goal models enrichment process. In the third phase, we describe two approaches for reasoning: a deterministic one based on SAT solvers and a probabilistic one based on Markov Logic Networks. Both approaches use the knowledge base to generate a set of diagnostics. This diagnostics describes the set of components (ranked in the case of probabilistic reasoning) that have failed leading to the failure of the monitored applications.

3.2 Requirement Goal Models

Goal Models are tree based formalisms that can be used to represent and denote the conditions and the constraints under which functional and non-functional requirements of a system can be delivered. In addition, goal models represent and denote positive and negative contributions, a requirement or a design decision may have on other requirements. Goal models have been found to be effective in concisely capturing large numbers of alternative sets of low-level tasks, operations, and configurations that can fulfill high-level stakeholder requirements.

Goals and tasks are connected with each other using AND- and OR-decompositions. The AND-decomposition of a goal G into other goals or tasks C_1, C_2, ..., C_n implies that the satisfaction of each of G's children is necessary for the decomposed goal to be fulfilled. The OR-decomposition of a goal G into other goals or tasks C_1, C_2, ..., C_n implies that the satisfaction of one of these goals or tasks suffices for the satisfaction of the parent goal. In Figure 3.2, we show goals g_1 and g_3 as examples of AND and OR decompositions.

Furthermore, a goal model can include soft-goals to represent non-functional requirements. In this respect, *functional requirements* are represented as hard goals, while *non-functional requirements* are represented as soft goals [108]. As opposed to hard goals, soft-goals do not have a clear-cut criterion to be used in order to decide whether they are satisfied or not.

Goal models can also have positive (make) or negative (break) contribution links from a hard-goal/soft-goal to a hard-goal/soft-goal indicating that satisfaction/*satisficing* of one goal/softgoal leads to full/partial satisfaction or denial of another goal/softgoal. The break

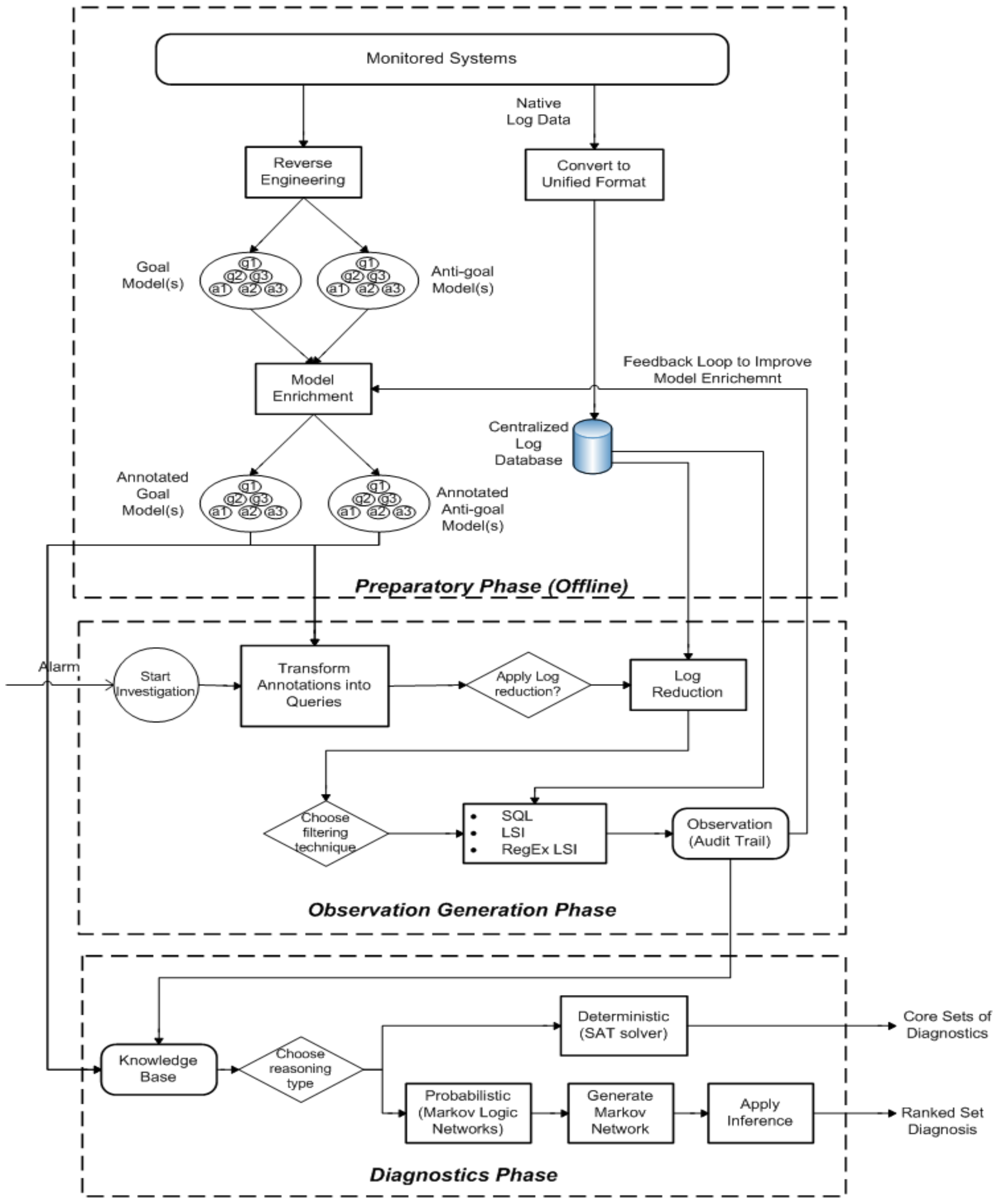


Figure 3.1: Overall Root Cause Analysis Framework

link means that satisfaction of the origin of the link causes denial of the destination. The make link means that satisfaction of the origin implies satisfaction of its destination. In both cases though, denial of the origin does not imply anything about the destination. The make link from goal g_1 to task a_1 and the break link from goal g_3 to tasks a_4 and a_5, shown in Figure 3.2, are examples of contribution links.

Giorgini et al. [35] provide a formal semantics for goal/softgoal relationships and propose reasoning algorithms which make it possible to check (a) if root goals are (partially) satisfied/denied assuming that some leaf-level goals are (partially) satisfied/denied; (b) search for minimal sets of leaf goals which (partially) satisfy/deny all root goals/softgoals. In this thesis, we adopt the formalizations for goal modeling as proposed by Giorgini et al. [35].

Lapouchnian et al. proposes the contextual variability in goal models to capture the impact of domain variability on the requirements [62]. Traditional requirements specifications do not consider contextual properties and do not explicitly specify how domain variability affects the requirements. Lapouchnian et al. enhances goal models in order to better represent the variability of domain models, thus reflecting the changing, dynamic nature of operational environments for software systems, and its impact on software requirements. In the context of the proposed RCA framework, the use of the context aware goal models to represent the monitored systems makes these models less susceptible to future changes in these systems and their operational environments.

In practice, goal models can be represented using standard software tools such as UML models, Eclipse modeling framework, BPMN, etc. We cite the work of Eriksson et al. [31] who propose a representation of goals as objects where UML object diagrams are used to represent the relationships between goals and subgoals. In this context, a goal model is a UML diagram that illustrates goal object instances where each goal object is indicated as a goal class stereotyped to Goal. Wang et al. [104] uses Eclipse modeling framework to represent goal models. On the other hand, Santos et al. [84] propose to represent variability in business processes, as described in BPMN (Business Process Modeling Notation), using AND-OR goal models. The aim of this approach is to use the formalism and techniques of goal models to derive the configuration and representation of business processes.

3.3 Model Annotations

Because a generic goal model is a highly abstract description of the early requirements, it needs to be annotated with extra information in order to facilitate design decisions. Wang et al. annotate goals/tasks in goal models with preconditions, effects and links to source code in order to prove the satisfiability/deniability of the goals/tasks [104]. Zawawy et al. use OCL based expression as goal model node annotations and adds the occurrence of a goal/task as a new type of annotation [111]. Yu et al. annotates goal tree AND relationships as sequential (;) or parallel (||), and OR as inclusive or exclusive [108].

Occurrence as well as, preconditions and postconditions are evaluated by a pattern matching approach that is discussed in detail in [114].

We experimented and used a set of formalisms to build the annotations expressions. In the following sections we describe the different formalisms, their advantages and disadvantages in the context of our work.

3.3.1 Propositional Logic

The propositional logic is defined as "the branch of logic that studies ways of joining and/or modifying entire propositions, statements or sentences to form more complicated propositions, statements or sentences, as well as the logical relationships and properties that are derived from these methods of combining or altering statements" [2].

A proposition is a sentence that is capable of having a truth-value, such as being true or false. Propositions can be joined using the logical connectives "and" and "or" (represented by $\&$ and $\vee$ respectively) to form more complex propositions. By prefixing a statement with "not" (represented by $\neg$), an affirmative proposition is changed into a negative statement and vice versa. Moreover, simple propositions are considered as indivisible wholes, thus, no logical relationships can be extracted from parts of statements such as their subjects and objects. Finally, propositions can be related using the logical relationships such as implication ($\rightarrow$) and equivalence ($\iff$). More details on proposition logic can be found in [2].

In this work, we use the propositional logic as a formalism to represent the goal model annotations. Examples of annotation expressions for the node *ag2:SQL Injection* in Figure 3.2 written using propositional logic are the following:

Precondition: Web Service request received $\&$ Credit history request processing started
Effect Credit history request processing started $\&$ SQL Drop Table executed Table name

As we show throughout this thesis, we use the goal model annotations' propositions to extract the log data of interest for the root cause analysis. The log data extraction is done by first, transforming such expressions into regular expressions by replacing the spaces among keywords in the propositional expression with the string ".*", and second applying tools such as Grep to extract log entries that match the generated regular expressions. We describe this process in detail in Chapter 5.

3.3.2 Object Constraint Language

Object Constraint Language (OCL) is a declarative strongly typed text language with expressive power for describing queries and constraints that apply and align with the Unified Modeling Language (UML) and MetaObject Facility (MOF) models, and is a key component of Queries/View/Transformation (QVT), the Object Management Group (OMG) standard recommendation for model transformation [75]. One of the design objectives of OCL is to be used to augment a UML class diagram with additional information, such as class invariants, that cannot be otherwise expressed using standard UML specifications. In this context, OCL expressions are independent of the programming languages, and can be used to express class invariants, pre- or postconditions. An OCL formula generally includes the following components:

context: <a qualifier identifier expression for the whole formula>
def: <some definitions of variables and types appearing in the formula>
inv: <some invariant constraints related to values appearing in the formula>
pre: <an expression related to the preconditions for the formula>
body: <the body of the OCL formula>
post: <an expression of postconditions for the formula>

In this thesis, we use OCL as one type of formalism that we use to express the goals/tasks' annotations in a goal model. This is done by setting, for each node, the *context* OCL component to contain a qualifier (name or ID) for this node. The *pre*, *body* and *post* components contain the precondition, occurrence and postcondition logical expressions for this specific node. The remaining *def* and *inv* sections are set to NULL. Each of these components are built using a (*logical_expression*) which is a Boolean formula consisting of the intersection (AND) or union (OR) of a set of *atomic_logical_expressions*. Each atomic logical expression is of the form:

$(self.attribute\_name,$

$comparison\_operator,$

attribute_value)

where *attribute_name* represents an attribute of the log data schema (such as description, source, etc.), *comparison_operator* is of the form $(=, <>, <, >, >=, <=)$, and *attribute_value* is the value of *attribute_name*.

An example based on the motivating scenario loan application in section 3.7 is as follows:

Precondition: *self.address = 'Machine2' and self.event_id = 'POST' and self.event_severity = Severity::Information and (self.reporter_component = 'Server' or self.reporter_component = 'Loan Application')*

Occurrence: *self.address = 'Machine1' and self.event_id = 'PROCESSED' and self.event_severity = Severity::Information and (self.reporter_component = 'Server' or self.reporter_component = 'Loan Application')*

Effect: *self.address = 'Machine1' and (self.event_id = 'APPROVED' or self.event_id = 'REJECTED') and self.event_severity = Severity::Information and (self.reporter_component = 'Server' or self.reporter_component = 'Loan Application')*

3.3.3 SQL Query Language

Structured Query Language (SQL) is the standard programming language designed for communicating and managing data in relational database management systems (RDBMS). SQL is designed not only to retrieve data from a database, but also to insert and update data. One of the important concepts in SQL is the query which is used to retrieve the data based on specific criteria. A query consists of a series of optional and mandatory keywords and clauses. A query is built by first using the keyword *SELECT* followed by a list of table columns or by " *" to specify that the query should return all columns of the queried tables. Next, the query contains the *FROM* keyword and clause which indicates the tables where the data is to be extracted. Third, the WHERE clause which restricts the data returned by the query by using a comparison predicate. The query can also include the *GROUP BY*, *HAVING* and *ORDER BY* keywords and clauses to aggregate, filter and order the results returned by the query. More details on SQL can be found in [20].

In this work, we are particularly interested is the *WHERE* clause. The *WHERE* clause consists of a set of atomic conditions that are bound together using an *OR* (return rows satisfying any of the conditions) and/or an *AND* (return rows satisfying all of the conditions) of a set of conditions. Each atomic condition is of the form:

$(column\_name,$

$operator,$

$value)$

where *column_name* represents a column of the table hosting the log data (such as description, source, etc.), *operator* is of the form *(=, <>, >, >=, <, <=, IN, BETWEEN, LIKE, IS NULL or IS NOT NULL)*, and *value* is the value of *column_name*.

An example based on the motivating scenario loan application in section 3.7 is the following:

Description like 'Windows.%starting.up' AND Source_Component like 'hume.eyrie.af.mil'

The advantage of using this type of formalism for building the goal model annotations is that it can be used readily (without any transformation) to build a query that can be applied on the log database table.

3.4 Anti-Goal Modeling

Anti-goals represent the tasks and actions of an intruder with the intention to threaten or compromise specific security goals. Similarly to goal models, anti-goal models are also denoted as AND/OR trees built through systematic refinement until leaf nodes are derived. Leaf nodes represent tasks (actions) an intruder can perform to fulfill an anti-goal, and consequently deny a stakeholder's goal for the system. Anti-goal models were initially proposed by Lamsweerde [99] to model security concerns during requirements elicitation. The trees with roots ag_1 and ag_2, shown in Figure 3.2, are examples of anti-goal models.

Formally, anti-goals are defined as follows: let G be a goal and Dom a set of domain properties. An assertion O is said to be an *obstacle* (anti-goal) to G in Dom if and only if the following hold:

1. {O, Dom} $\models \neg$ G (O)
2. {O, Dom} | $false_{Dom}$

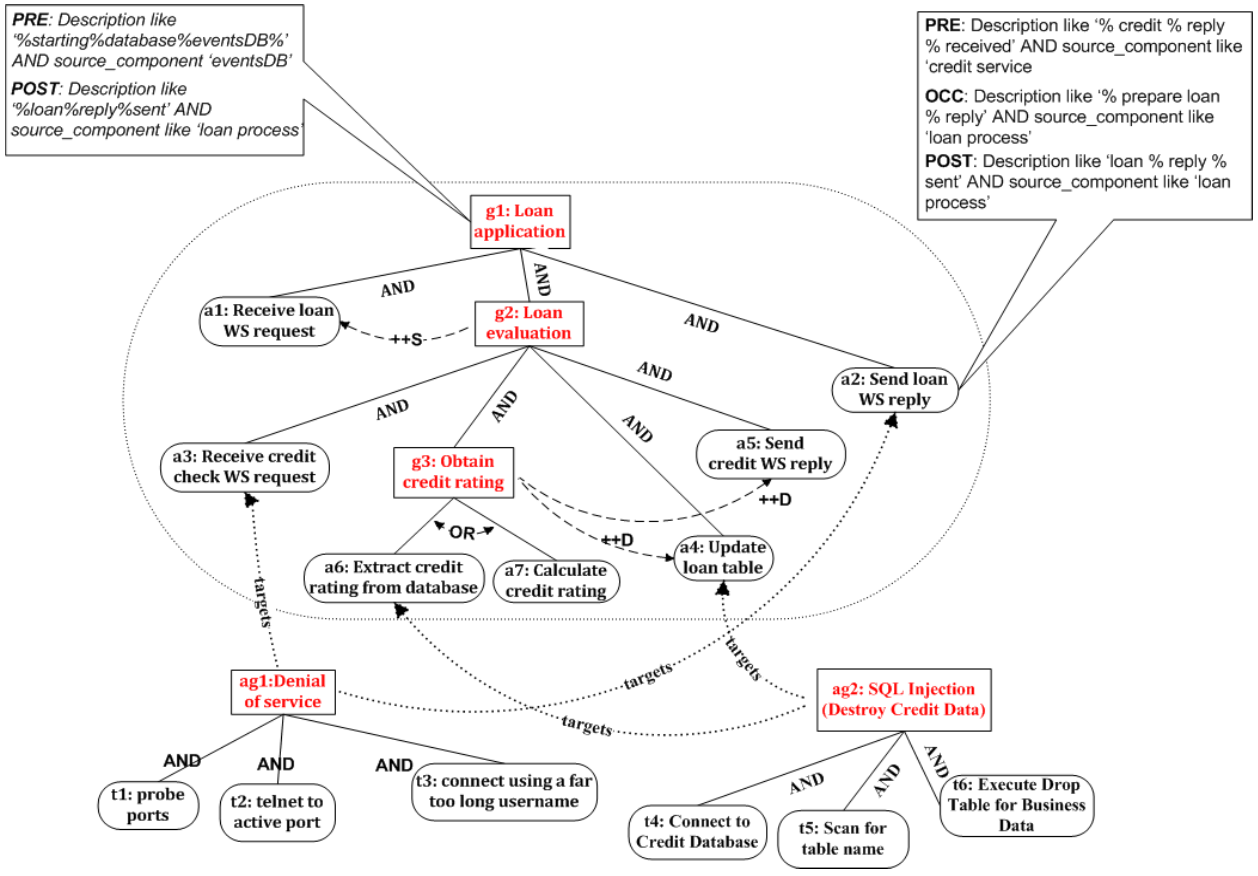

PRE: Description like '%starting%database%eventsDB%' AND source_component 'eventsDB'
POST: Description like '%loan%reply%sent' AND source_component like 'loan process'
PRE: Description like '% credit % reply % received' AND source_component like 'credit service
OCC: Description like '% prepare loan % reply' AND source_component like 'loan process'
POST: Description like 'loan % reply % sent' AND source_component like 'loan process'
g1: Loan application
AND
a1: Receive loan WS request
++S
g2: Loan evaluation
a2: Send loan WS reply
a3: Receive credit check WS request
g3: Obtain credit rating
a5: Send credit WS reply
++D
OR
a6: Extract credit rating from database
a7: Calculate credit rating
a4: Update loan table
targets
ag1:Denial of service
ag2: SQL Injection (Destroy Credit Data)
t1: probe ports
t2: telnet to active port
t3: connect using a far too long username
t4: Connect to Credit Database
t5: Scan for table name
t6: Execute Drop Table for Business Data

Figure 3.2: Loan Application Goal Model and Corresponding Anti-Goal Attacks.

Condition (1) states that the negation of a goal is a logical consequence of the obstacle specification and the set of domain properties available; condition (2) states that the obstacle may be logically consistent with the domain of the logs and the goal models. The target links from anti-goal ag_1 to tasks a_2 and a_3 as well as from anti-goal ag_2 to tasks a_4 and a_6, shown in Figure 3.2, exemplify the relationship between anti-goals and their targets.

The process of elaborating anti-goal models can be either done informally by asking HOW/WHY questions or formally by regression through the goal model and the domain theory or by use of refinement and obstruction patterns as discussed in [100, 101]. The refinement technique proposed by Lamsweerde et al. involves building two models interactively and concurrently that is, a) a goal model representing the functional and non-functional requirements of the system-to-be; and b) an anti-goal model derived from the requirement goal model that specifies how assets of the application-to-be can be compromised. The anti-goal refinement process can follow a process similar to goal trees by AND-OR decomposition until leaf nodes are specified. In this context, leaf nodes represent the malicious steps or actions implementable by the attacker.

Souza et al. [91] use goal and anti-goal models to represent monitored applications and the related attacks in the context of malicious detection. The monitoring framework described in [91] infers if the deniability of a goal/task is due to a concurrent deniability attack modeled as an anti-goal tree. Zawawy et al. adopted a similar approach based on requirement goal to model the monitored systems and anti-goal models to model malicious behavior but used instead a probabilistic approach [112].

3.5 Log Filtering and Reduction

The framework described in this thesis relies on mining system log files in order to discover events of interest. These events of interest serve as observation while inferring the root causes for the monitored system's failure. In this context, we consider log reduction as the process of reducing the size of the logged data by eliminating data (noise) that are not relevant to a particular criterion. Log filtering becomes the problem of retrieving log data entries that are relevant to a particular query. Log reduction and filtering are done by analyzing textual logs and applying text mining techniques. Generally, text mining seeks to extract useful information from unstructured textual data through the identification of interesting patterns by using techniques from the fields of natural language processing, data mining, machine learning and information retrieval.

Text mining techniques are typically used for document clustering, classification or information retrieval from a corpus of documents. Vector space models, dimensionality reduction and k-means are commonly used for clustering documents. However, vector space model and latent semantic indexing is typically used for document retrieval. For document classification, the common text mining techniques are based on naive Bayes classifier, decision trees or support vector machines. In this thesis, we are interested in the application of document retrieval using text mining.

3.5.1 Latent Semantic Indexing

Some of the well-known document retrieval techniques include latent semantic indexing (LSI) [24], probabilistic semantic analysis (PLSI) [48], and latent Dirichlet allocation (LDA)

[10] and the correlated topic model [63]. In this context, semantic analysis of a corpus of documents consists of building structures that identify concepts from this corpus of documents without any prior semantic understanding of the documents.

LSI is an indexing and retrieval method to identify relationships among terms and concepts in an unstructured collection of text. LSI was first patented in 1989 by Deerwester et al. [24]. LSI takes a vector space representation of documents based on term frequencies as a starting point and applies a dimension reduction operation on the corresponding term/document matrix using the singular value decomposition (SVD hereafter) algorithm [39]. Similarities among documents and queries can be more reliably estimated in the reduced space representation than in the original representation. This is because documents which share frequently co-occurring terms will have a similar representation in the reduced space representation, even if they have no terms in common. LSI is commonly used in areas such as web retrieval, document indexing [106] and feature identification [77]. In our work we consider each log entry as a document. In this respect, LSI can be used for identifying those log entries that mostly associate with a particular query denoting a system feature, a precondition, an occurrence or an effect of an operation or a failure. In a nutshell, the LSI based algorithm can be outlined as follows:

1. The first step is to create a vocabulary from the log data. This is referred to as the text tokenization. This done by extracting keywords from the corpus of log data entries. These keywords represent concepts, actions, etc. During this process, all capitalization, punctuation and extraneous markup are stripped away. Next, the vocabulary is refined by applying a list of stop words and removing commonly used words that do not carry semantic meaning. Finally, a stemming process is applied to the vocabulary by removing common endings from words, leaving behind an invariant root form. The result of this step is a list of keywords that are stripped, pruned and stemmed. The tokenization and stemming steps are applied to all log entries creating a vocabulary that contains flat words from all log data.

2. The second step is to create the term-document matrix. This is an m by n dimensional matrix where m represents the total number of documents in the corpus and n is the total number of terms (vocabulary) generated in step 1. Thus, a row of this matrix is a signature vector corresponding to a log entry. A column of this matrix is a vector representing a term in the vocabulary. The two dimensional matrix is populated using a weight function *w(d,t)* where d represents a document and t represents a term in the vocabulary. The simplest form of the weighting function wf is the following:

$$wf_{t,d} = \begin{cases} 1 & \text{if } t \text{ has a match in the log entry d} \\ 0 & \text{otherwise.} \end{cases} \tag{3.1}$$

Another form of weighting function wf is the following term frequency (*tf*):

$$tf = count(t, d) \tag{3.2}$$

which computes the number of occurrences of keyword t in log entry d in a log data corpus L.

Another form of weighting function wf commonly used in ranking web documents is the following term frequency-inverse document frequency (*tf-idf*):

$$tf\text{-}idf = \frac{count(t, d)}{\Sigma_{i \in L-\{d\}}(count(t, i))} \tag{3.3}$$

As shown in equation 3.3, *tf-idf* is computed as the ratio of the frequency of term t in the document d (*tf(d,t)*) over the frequency of the term in the whole collection of documents *idf(t)* (inverse document frequency). (*tf-idf*) is used to reduce the weight of common terms.

3. The third step consists of a transforming the term-document matrix in a lower dimension matrix. Using the singular value decomposition (SVD) algorithm, an m by n term-document matrix A is factorized into a product of three matrices ($U * S * V^T$), where U is an m by n matrix, S is an n by n diagonal matrix containing the singular values of A and V^T is an n by n matrix. The rank of a matrix is defined as the number of nonzero singular values of this matrix. Next, a rank (dimensionality) reduction process is applied where the k highest singular values are kept and the remaining ones are discarded. This rank lowering reduces $S_{n.n}$ (diagonal matrix containing the singular values) into $S_{k.k}$. Thus, the product ($U_{m.n} * S_{n.n} * V^T_{n.n}$) is transformed into ($U_{m.k} * S_{k.k} * V^T_{k.n}$). The transformed matrix is called the concept-document matrix. By transforming the data from a high-dimensional space to a space of fewer dimensions, the reduction process eliminates less significant "concepts" and keeps the more prominent k "concepts". In other words, the dimensionality reduction allows for the collapse of the common terms that belong into the same latent concept to be grouped together. The choice of the number of concepts k is done heuristically.

4. Finally, information retrieval (the selection of relevant log entries in context of this work) can be achieved by use of simple vector operations. Documents are represented using a new vector representation based on the concept matrix as follows:

$$d_{1.k} = d_{1.m}^T * U_{m.k} * S_{k.k}^{-1} \tag{3.4}$$

Queries can be performed by encoding the queries similar to the documents using a query vector as shown in the following equation:

$$q_{1.k} = q_{1.m}^T * U_{m.k} * S_{k.k}^{-1} \tag{3.5}$$

Finally, the last step is to rank the log data entries based on the logical expression. In this respect, a logical expression is considered as a document and is converted into a vector in the reduced semantic space. We rank documents in descending order of cosine similarities as described in function f (equation 3.6). f generates a column vector matrix representing the relevance of each log data entry l with respect to the query q being considered with L and Q being the corresponding row and column vector matrices respectively. Note that this is a similarity function so the higher the value of $f(q,l)$ the more relevant are q and l to each other.

$$f(q,l) = \frac{Q.L^T}{|Q|.|L|} \tag{3.6}$$

3.5.2 Probabilistic Latent Semantic Indexing

PLSI is a variation of LSI [48] that provides a more solid theoretical foundation by considering the existence of three types of random variables (topic, word and document). The three types of class variables are:

- An unobserved class variable $z \in Z = \{z_1, z_1, ...z_K\}$ that represents a topic
- Word $w \in W = \{w_1, w_2, ...w_L\}$ where each word is the location of a term t from the vocabulary V in a document d
- Document $d \in D = \{d_1, d_2, ...d_M\}$ in a corpus M

Using PLSI, each observation consists of a pair *(d,w)* while the latent class variable z is discarded. A generative model can be defined in the following way:

- Select a document d with probability P(d),
- Pick a latent class z with probability P(z/d),
- Generate a word w with probability P(w/z).

Expectation Maximization Algorithm

In order to estimate the P(d), P(d/z) and P(w/z) distributions, we use the log data corpus as a training set for building a statistical model for this log data by applying the Expectation Maximization (EM) algorithm. EM consists of alternating two steps of expectation and maximization until the result eventually converge into a set of probabilistic model that best represent the observations (log data corpus):

- An expectation step where posterior probabilities are computed for the latent topic variables z, based on the current estimates of the parameters. The probability P(z/d,w) that a word w in a particular document or context d is explained by the factor corresponding to z as follows:

$$P(z/d,w) = \frac{P(z)P(d/z)P(w/z)}{\sum_{z'} P(z')P(d/z')P(w/z')} \tag{3.7}$$

- A maximization step, where parameters are updated for given posterior probabilities computed in the previous step as follows:

$$P(w/z) = \frac{\sum_{d} n(d,w)P(z/d,w)}{\sum_{d,w'} n(d,w')P(z/d,w')} \tag{3.8}$$

$$P(d/z) = \frac{\sum_{w} n(d,w)P(z/d,w)}{\sum_{d',w} n(d',w)P(z/d',w)} \tag{3.9}$$

$$P(z) = \sum_{d,w} n(d,w)P(z/d,w) \tag{3.10}$$

These two stepsmaximization and expectationare iterated until the approximated probability distributions values converge into a set of values that best fit the observations. Details on the EM algorithm and its application in PLSI are in [48].

Applying Query using PLSI

After calculating the probability distributions for P(d), P(d/z) and P(w/z) as described in section 3.5.2, we obtain the word-document distribution P(w/d) for each document d as follows:

$$P(w/d) = \sum_{z} P(w/z)P(z/d) \tag{3.11}$$

where P(z/d) is calculated as follows:

$$P(z/d) = \frac{P(d/z)P(z)}{\sum_{z'} P(d/z')P(z')} \tag{3.12}$$

Next, we calculate the word-document distribution *P(w/d)* for each query using Equation 3.11 similarly to the calculation of the word-document probability distribution for a document in the corpus.

Note that PLSI is not a generative model of new documents, i.e., if the queries are not known a priori then using the already calculated *P(d)*, *P(d/z)* and *P(w/z)* probability distributions in order to calculate the word-document distribution *P(w/d)* for each query can lead to inaccurate results. However, in this thesis, the queries are known a priori and are used as part of the training corpus.

A document is considered to be relevant to a query if their corresponding multinomial P(w/d) distributions are similar. We estimate the relevance as the opposite of the dissimilarity calculated using the *KL* divergence described in the following section.

Distribution Divergence

The dissimilarity between two documents *d1* and *d2* can be estimated by measuring the *Jenson-Shannon* divergence between the corresponding words distributions over the two documents as follows:

$$js\_D(P||Q) = \frac{1}{2}(kl\_D(P||Q) + kl\_D(Q||P)) \tag{3.13}$$

where kl_D is the Kullback-Liebler (KL) divergence, defined as follows:

$$kl\_D(P||Q) = \sum_{i} P(i) log \frac{P(i)}{Q(i)} \tag{3.14}$$

Note that js and kl are dissimilarity functions (as opposed to the similarity function 3.6), thus the lower the values of js and kl for P and Q, the more relevant these two documents are for each other.

Finally, in addition to providing a mathematical foundation for semantic analysis, the advantages of PLSI is that it has a better handling of synonymy, as well as polysemy which is not handled by standard LSI.

3.5.3 Other LSI Variations

The Variable latent semantic indexing (VLSI) [19] is another variation of LSI. VLSI is a query dependent (hence "variable") low-rank approximation of the term-document matrix that minimizes the approximation error for any specified query distribution. VLSI tailors the information retrieval to particular queries resulting in improved performance over traditional LSI.

In 2006, Guven et al. proposed an n-gram based LSI [43] by using n-grams formed by word couples and triplets. In this thesis, we propose a generalized approach for n-gram based LSI by using regular expressions with LSI. We call this hybrid approach *regex LSI*. Regex LSI is a generalization of the approach by [43] because it extends the semantic space with regular expressions instead of n-grams formed by word couples and triplets.

3.6 Reasoning Techniques

In this section, we describe two approaches for reasoning that are used in this thesis. The first is a deterministic approach based on SAT solvers. The second is a probabilistic approach based on Markov Logic Networks.

3.6.1 SAT Solvers

The propositional satisfiability problem consists of determining whether there exists a truth assignment μ to variables of a propositional formula ϕ that makes the formula true. If such a truth assignment exists, the formula is said to be satisfiable. A SAT solver is any procedure that determines the satisfiability of a propositional formula, identifying one or more satisfying assignments of variables. In this thesis, we use an existing approach ([104]) that consists of transforming the diagnostic problem into a propositional satisfiability problem that can be solved by existing SAT solvers. This approach consists of two steps: first, the model representing the monitored applications and the observation for its runtime behavior are turned into a propositional logical formula and second, applying the SAT solver to find the truth assignment to variables of this propositional formula that makes it true. This truth assignment represents a diagnostic of the monitored system where negated variables represent failed tasks. The earliest and most prominent SAT algorithm is DPLL (Davis-Putnam-Logemann-Loveland) (Davis et al. 1962), which uses backtracking search. Even though the SAT problem is inherently intractable, there have been many improvements to SAT algorithms in recent years. Chaff (Moskewicz et al. 2001), BerkMin (Goldberg and Novikov 2002) and Siege (Ryan 2004) are among the fastest SAT solvers available today. Similarly to Wang et al. (2009), we use SAT4 (Le Berre 2007).

3.6.2 Markov Logic Networks

Markov Logic Networks (MLNs) have been recently proposed in the research literature as a way of providing a framework that combines first order logic and probabilistic reasoning. In this context, a knowledge base denoted by first-order logic formulae represents a set of hard constraints on the set of possible worlds that is, if a world violates one formula, it has zero probability of existing. Markov logic softens these constraints by making a world that violates a formula to be less probable but still, possible. The more formulas it violates, the less probable it becomes. A detailed discussion on MLNs can be found in [28].

Markov Network Construction

In MLNs, each logic formula F_i is associated with a nonnegative real-valued weight w_i. Every grounding (instantiation) of F_i is given the same weight w_i. In this context, a Markov Network is an undirected graph that is built by an exhaustive grounding of the predicates and formulas as follows:

- Each node corresponds to a ground atom x_k which is an instantiation of a predicate.
- If a subset of ground atoms $x_{\{i\}} = \{x_k\}$ are related to each other by a formula F_i with weight w_i, then a clique C_i over these variables is added to the network. C_i is associated with a weight w_i and a feature function f_i defined as follows,

$$f_i(x_{\{i\}}) = \begin{cases} 1 & F_i(x\{i\}) = True, \\ 0 & otherwise \end{cases} \tag{3.15}$$

Thus, first-order logic formula serves as templates to construct the Markov Network. In the context of a Markov network, each ground atom, X, represents a binary variable. The overall Markov network is then used to model the joint distribution of all ground atoms. The corresponding global energy function can be calculated as follows,

$$P(X = x) = \frac{1}{Z} \exp(\sum_i w_i f_i(x_{\{i\}})) \tag{3.16}$$

where Z is the normalizing factor calculated as,

$$Z = \sum_{x \in X} \exp(\sum_i w_i f_i(x_{\{i\}})) \tag{3.17}$$

where i denotes the subset of ground atoms $x_{\{i\}} \in X$ that are related to each other by a formula F_i with weight w_i and feature function f_i.

Learning

Learning of an MLN consists of two steps: structure learning, or learning the logical clauses, and weight learning, or determining the weight of each logical clause. Structure learning is typically done by heuristically searching the space for models that have a statistical score measure that fit to the training data [59]. As for weight learning, Singla et al. extend the existing voted perceptron method and applies it to generate the Markov logic network's parameters (weights for the logical clauses). This is done by optimizing the conditional likelihood of the query atoms given the evidence [87].

Inference

Assuming $\phi_i(x_{\{i\}})$ is the potential function defined over a clique C_i, then $log(\phi_i(x_{\{i\}})) = w_i f_i(x_{\{i\}})$. We use the constructed Markov network to compute the marginal distribution of events. Given some observations, probabilistic inference can be performed. Exact inference is often intractable, thus Markov Chain Monte Carlo sampling techniques, such as Gibbs sampling, are used for approximate reasoning [28]. The probability of an atom X_i given its Markov blanket (neighbors) B_i is calculated as follows:

$$P(X_i = x_i / B_i = b_i) = \frac{A}{(B + C)} \tag{3.18}$$

where,

$$A = \exp(\sum_{f_j \in F_i} w_j f_j(X_i = x_i, B_i = b_i)) \tag{3.19}$$

$$B = \exp(\sum_{f_j \in F_i} w_j f_j(X_i = 0, B_i = b_i)) \tag{3.20}$$

$$C = \exp(\sum_{f_j \in F_i} w_j f_j(X_i = 1, B_i = b_i)) \tag{3.21}$$

F_i is the set of all cliques that contain X_i and f_j is computed as in Equation 3.15.

The Markov network can then be used to compute the marginal distribution of events and perform inference. Because inference in Markov networks is #P-complete, approximate inference is proposed to be performed using the Markov chain Monte Carlo (MCMC), and the Gibbs sampling [80].

The Applications of Markov Logic Networks

MLN is applied in a variety of domains, including information extraction and integration, natural language processing, vision, social network analysis, computational biology, etc. Tran et al. [95] addresses the problem of event modeling and recognition in visual surveillance by using Markov Logic Networks. This is done by integrating the domain common sense knowledge with the uncertainty of the primitive event detection. The common sense knowledge is represented using first order logic statements, while the uncertainty of the analysis, produced by computer vision and object detection algorithms, is represented using the detection probabilities. The objective of this work is to improve the tracking and recognition of the movement of objects in visual surveillance.

In this thesis, we use Markov Logic Networks for the purpose of conducting RCA in software systems. In particular, we use weighted first order logic statements to represent the relationships between the different systems and components in the monitored environment. The weight of each statement represents the confidence in the corresponding relationship. We use the audit trail extracted from the log data as observation. In this observation, we use negation to represent an event that did not occur, and we use the interrogation to represent an event that may or may not have occurred. The probabilistic inference based on the resulting Markov Network is used to generate a set of diagnostics indicating the failed task or component that lead to the overall system's failure. We list the corresponding code and a sample set of observations in Appendix A.

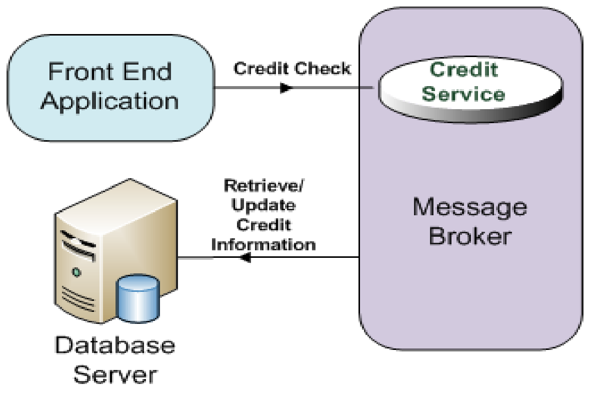


Figure 3.3: Layout of the Test Distributed Environment

3.7 Motivating Scenario

Throughout the thesis, we use the loan application goal model, shown in Figure 3.2, as a running example to better illustrate the inner workings of the proposed root cause analysis tools and techniques. The scenario is built based on a test environment that contains a financial set of applications and emulates an enterprise environment. As illustrated in Figure 3.3, the experimentation environment includes a business process layer and a service oriented infrastructure, and is built using commercial off-the-shelf software. In particular, this test environment (Figure 3.3) includes 4 systems/services: the front end application (soapUI), a message broker (IBM WebSphere Message Broker v7.0), the credit check Web Service and an SQL server (Microsoft SQL Server 2008).

Starting from the top, we built a business process (Apply_For_Loan) depicting the process of an online user applying for a loan and having their loan evaluated and finally a loan accept/reject decision is made based on the information supplied by the user. Technically, the business process is exposed as a web service. Apply_For_Loan process is started when it receives a SOAP request containing loan application information. It evaluates the loan request by checking the applicant's credit rating. If the applicant's credit rating is "good", his/her loan application is accepted and a positive reply is sent back to the applicant. Otherwise, a loan rejection is sent back to the applicant. If an error in processing occurs, a SOAP fault is sent back to the requesting application. The credit rating evaluation is done via a separate web service (Credit_Rating). After the credit evaluation is done, the Credit_Rating service sends a SOAP reply, back to the calling application, containing the credit rating and the ID of the applicant. If an error occurs during the credit evaluation, a SOAP fault is sent back to the requesting application. During the credit rating evaluation,

Table 3.1: Loan Application Goal Model's Nodes/Preconditions/Postconditions Identifiers

Node	**Precondition**	**Postcondition**
g1	InitialReq_Submit	Reply_Received
g2	ClientInfo_Valid	Decision_Done
a1	BusinessProc_Ready	BusProc_Start
a2	LoanRequest_Submit	LoanRequest_Avail
a3	Decision_Done	LoanReply_Ready
a4	SOAPMessage_Avail	SOAPMessage_Sent
a5	ClientInfo_Valid	Prepare_CreditRating_Request
a6	Prepare_CreditRating_Request	Receive_CreditRating_Rply
a7	Receive_CreditRating_Rply	Valid_CreditRating
a8	Valid_CreditRating	CreditRating_Available
a9	CreditRating_Available	Decision_Done

the Credit_Rating application queries a database table and stores/retrieves the applicants' details.

3.7.1 Goal Model for the Motivating Scenario

The loan application is represented by the goal model in Figure 3.2 which contains three goals (rectangles) and seven tasks (circles). The root goal g_1 (loan application) is AND-decomposed to goal g_2 (loan evaluation) and tasks a_1 (Receive loan web service request) and a_2 (Send loan web service reply), indicating that goal g_1 is satisfied if and only if goal g_2, tasks a_1 and a_2 are satisfied. Similarly, g_2 is AND-decomposed to goal g_3 (extract credit rating) and tasks a_3 (Receive credit check web service request), a_4 (Update loan table) and a_5 (Send credit web service reply), indicating that goal g_2 is satisfied if and only if goal g_3, tasks a_3, a_4 and a_5 are satisfied. Furthermore, subgoal g_3 is OR-decomposed into tasks a_6 and a_7. This decomposition indicates that goal g_3 is satisfied if either tasks a_6 or a_7 are satisfied. Contribution links ($++D$) from goal g_3 to tasks a_4 and a_5 indicating that if g_3 is denied then tasks a_4 and a_5 should be denied as well. Contribution link ($++S$) from g_2 to task a_1 indicates that if g_2 is satisfied then so must be a_1. Table 3.1 provides a complete of the goals and tasks in the loan application goal model.

3.7.2 Anti-goal Model for the Motivating Scenario

This scenario also contains two anti-goal models ag_1 and ag_2. The anti-goal ag_1 is a denial of service attack where the attacker sniffs for weaknesses in the targeted system and then attempts to disrupt the logging-in process in order to take the system out of service. The anti-goal ag_2 describes an attacker connecting using legitimate credentials with limited access rights but then attempts to execute malicious code in order to dropping the table containing the credit scores.

3.7.3 Test Environment

Vertically, the test environment contains three layers:

- Business process layer: contains the business processes (including Apply_For_Loan business process) and the process server (IBM WebSphere Business Process).
- Service Layer: contains application integration logic and services such as the Credit_Rating service and the IBM WebSphere Message Broker (ESB).
- Infrastructure Layer: represented by the MQ communication protocol and product (IBM WebSphere MQ) as well as the Microsoft SQL Server 2008 database management system which hosts the LogData database table.

Horizontally, the test environment is a three tier environment:

- The first tier contains the front end application, which is simulated using a web service invocation tool used by developers for testing purposes called soapUI [90].
- The middle tier contains the business logic and middleware components.
- The third tier contains the backend database.

3.7.4 Logger Systems in the Test Environment

The test environment includes four logging systems that emit the log data/events. These logging systems are described as follow:

1. Windows Event Viewer: is a Windows component that acts as a Windows centralized log repository. Events emanating from applications running as Windows services (Message Broker, MQ and SQL Server) can be accessed and viewed using the Windows Event Viewer.

2. Front end application (soapUI): is an Open Source Web Service Testing Tool for Service Oriented Architecture. soapUI logs its events in local log file.

3. Middleware Service: the Credit_Rating application generates events that are stored directly in the log database.

4. WebSphere Process Server and Deployed Business Processes: The process server enables the generation of events for each step in the business processes. These events are generated as CBE events and are stored in the Common Event Infrastructure (CEI) which is a common business events repository.

3.8 Summary

As mentioned in Chapter 1, this thesis work can be divided into modeling, log analysis and diagnosis. In this chapter, we focused on the modeling part and described the use and the enrichment of requirement goal models as runtime models. We also present an overview of the techniques used for log reduction and filtering and in particular those based on text mining and semantic analysis. We also describe the Markov Logic Network technique for probabilistic reasoning used in this thesis and we describe some of its applications in software systems. In the following chapters, we handle the log analysis and diagnosis and in particular, we describe the use of the aforementioned techniques for log reduction and filtering as well as our approach for diagnosing failures in software systems.

Chapter 4

Log Reduction

In this chapter, we discuss the topic of log data reduction in the context of our proposed root cause analysis approach. In this context, log reduction aims at reducing the size of the logged data by eliminating data that are not relevant to a particular diagnostic hypothesis (noise), and to focus the attention of the human operators to events that may relate to root causes of an observed failure. This process can be used prior to the root cause analysis process and aims at decreasing the large size of the natively generated log data so that further analysis (using this log data) can be tractable. The challenge in this process is to provide an optimal reduction without losing any of the log data entries that are of interest in the subsequent root cause analysis.

The chapter is organized as follows. In Section 4.1, we present the motivation for the work described in this thesis and its relevance to the overall work in the thesis. In Section 4.2 the overall architecture of the proposed framework is presented, while in Section 4.3 the formalization of the proposed process is discussed. Section 4.4 discusses a unified schema for the storing the log data. Section 4.5 presents the process of query formation, log reduction, and log merging. Section 4.6 presents the experimental results obtained by applying the reduction framework to various operational scenarios of a SOA system. Section 4.7 presents a summary of this chapter.

4.1 Motivation

Driven by regulatory compliance, security, forensics and IT operations, the collection and analysis of log data in large enterprise environments has become a Terabyte size issue.

In fact, software logs are commonly considered part of the *big data* phenomenon which includes other phenomenon such as climate, weather, wireless sensor networks, etc. Such phenomena increasingly grow in size as data is being gathered, until they become too complex to search, store, analyze or visualize.

Regulatory constraints in different industries often include the retention of audit log data for a certain amount of time. For example, it is a common requirement in the retail industry that log data, generated when using electronic cards, is stored for one year with the last three months available in an easily accessible storage. The compliance is applied to not only systems that store or process card data, but also systems that are directly connected to them. In addition, the health and banking sectors impose regulations on the retentions of information related to patients and clients. For instance, the US health act (HIPAA) enforces that logs be maintained for a minimum of six or seven years, depending on the type of data. Because the log data generated by the IT systems in such environment could potentially contain business or client related information, this automatically makes it bound to the same regulations as the business data.

To deliver high quality customer service that is based on specific service level agreements, a computer system needs to be monitored in order to ensure reliable operations. Typical enterprise environments generate large amounts of data to exhaustively log the activity of users and devices. It is estimated that 44% of large organizations collect at least 1TB of log data monthly and 11% collect more than 10TB of data [40]. This sharp increase in the amount of log data generated in enterprise environments leads to an increase in the cost of storage which is out-pacing the decline in storage prices. For example, to back up log data that is being collected at a rate of 1TB monthly using an annual estimated cost of storage $5 per gigabyte (2008 estimate), the total storage cost is $60,000 annually.

Another challenge is to have an efficient and knowledge extraction search capability across log data incoming from different sources. The sheer size of the log data makes it intractable to examine the log data without automated support.

4.2 Overall Architecture

In this chapter, we present the log reduction framework. This framework is structured as a pipe-and-filter system as illustrated in Figure 4.1. The input to the system consists of streaming log data, information about the monitored systems represented by annotated requirements goal models and, monitoring objectives defined by the system administrator using the framework. The output of the framework consists of a collection of log data that

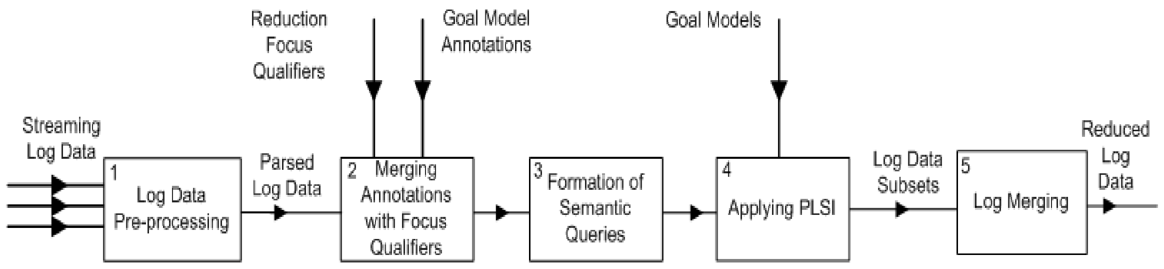


Figure 4.1: Log Reduction Process

is a subset of the total sum of log data generated by all monitored components. This subset pertains to the specific objectives and monitoring requirements the user has specified and consists of system events that are related to the specific problem or situation that the system administrator is interested in. The following sections describe in more detail the inputs and output of the proposed framework.

4.2.1 Framework Inputs

Log Data

This is the natively generated log data coming from the logger component of each system in the monitored environment. Log data entries are received as sequences or bursts of events from monitored applications which is later pre-processed in preparation for reduction. The processing is a step process: first, log data transformed from its native format into a common format that we describe in detail in Section 4.5.1. Next, the transformed log data is eventually stored in a centralized database using a unified table format (see Figure 4.1).

Annotated Goal Models

The second input to the framework is a set of annotated goal model(s), representing the requirements of the monitored applications. Goal models that we use in this work have been originally formalized in Giorgini et al. [35]. These models are built prior to using the reduction framework. Similarly to [104], we associate nodes in goal models (goals and tasks) with *preconditions* and *postconditions*. In this work, *preconditions* and *postconditions* are represented by logical expressions that are expressed using OCL formalism and that must be satisfied before and after respectively, a task is successfully executed or a goal is satisfied. As presented also in [111], we annotate each goal/task node in the goal tree model with the logical expression that represents the conditions under which each goal tree node can occur. We annotate the AND relationships of a goal tree as sequential (;) or parallel (||), and the OR as inclusive or exclusive [108]. Figure 3.2 illustrates an example of a goal model representing the loan application business process used in our experimentation scenario.

Reduction Focus Qualifiers

The third input is the collection of reduction *qualifiers* that represent the user's particular points of interest such as intervals, names and types of specific systems or services, server descriptors, IP address, and user names and IDs. The reduction *qualifiers* are expressed using logical expressions based on OCL. An example of a reduction *qualifier* for events that their description contains the string "John Smith" and have occurred between timestamps [T1, T2], is the following:

context Qualifier
body: self -> forAll(e | e.Report_time >= '2008-08-22 10:00:00' and e.Report_time < '2008-08-22 12:00:00' and e.Description = 'Loan request submitted for John Smith'

4.2.2 Framework Output

Figure 4.1 illustrates the main components in the reduction framework. The incoming log data are normalized and filtered. A subset of log data is selected for each of the queries that annotate the goal model nodes in the form of *preconditions*, *occurrence* and *postconditions*. Finally, the filtered log data subsets are merged, thus forming a unified log data set that enables the system administrator to screen out noise, and to focus on log data that relate to the requirement being monitored. The output of the proposed framework is an amalgamation of the filtered log data generated in the previous section and sorted in chronological order, with a reduction rate that can reach over 70% with respect to the original log data corpus. Furthermore, we propose an algorithm to generate an interpreted version of the log data in the form of a planfile which consists of a sequence of logical literals indicating the (non) occurrence of events. More details about the interpretation algorithm are in section 4.5.4.

4.3 Formalization of the Reduction Framework

In this section we present in more detail the formalization of the major elements of the log reduction process.

Let $\mathcal{D}$ be a distributed system being monitored. We represent such a system as a collection of components C_1, C_2, ... C_m that is:

$$\mathcal{D} = \{C_1, C_2, \ldots C_m\}$$

In this context, a component is an artifact that can represent a software application (e.g., a service), a hardware related component (e.g., a server logger) or an infrastructure related device (e.g., a network router logger). We also consider that the system may provide different functionalities where each functionality can be exercised by several usage scenarios. Let $S_{i,j}$ be the j$^{\mathrm{th}}$ scenario for i$^{\mathrm{th}}$ component. For each scenario $S_{i,j}$ that is applied, each component C_i generates a set of events:

$$E_{i,j} = \{\mathrm{e}^{\mathrm{k}}{}_{i,j} | k \in \{1, 2 ... n\}\}$$

where $\mathrm{e}^{\mathrm{k}}{}_{i,j}$ is the k$^{\mathrm{th}}$ event generated by component C_i, and n is the total number of events generated by component C_i when triggered by scenario $S_{i,j}$. Note that n depends on i and j, but we do not show this to keep the notation simple.

We also consider that event $\mathrm{e}^{\mathrm{k}}{}_{i,j}$ is represented by a set of attribute-value pairs:

$$\mathrm{e}^{k}{}_{i,j} = \{<\mathrm{a}^{\mathrm{i,1}},\ \mathrm{v}^{\mathrm{i,1}}>,\ <\mathrm{a}^{\mathrm{i,2}},\ \mathrm{v}^{\mathrm{i,2}}>, ... <\mathrm{a}^{\mathrm{i,m}},\ \mathrm{v}^{\mathrm{i,m}}>\}$$

where m is the number of attributes in the events generated by the logger of component C_i. Again, m depends on i, j, k, but this is not shown to keep the notation simple.

The first step of the process that can be applied as part of the proposed framework is a normalization step where events generated by different loggers of the different components of the system, are normalized in a common schema format. This step could be formalized as a mapping

$$E_{i,j} \rightarrow E'_{i,j}$$

where $E'_{i,j}$ represents events emitted by component C_i for scenario S_j transformed for a common meta model illustrated in Table 4.1. The *LogData* schema includes the essential fields from the WSDM Event Format (WEF) [73].

The second step of the log reduction process is the selection of events of interest (L) based either on the user specified queries that relate to preconditions, postconditions and occurrence annotations for a particular goal in the goal tree by considering PLSI among

Table 4.1: Unified Schema for Log Data

Unified Schema Field	**WEF Field**	**Description**
report_time	ReportTime	Time when the event was generated
source_component	SourceComponent/-Name	Name of the resource experiencing the event
source_address	SourceComponent/-ComponentAddress	Host Name/IP address and/or Application Name and/or Component Name
event_id	EventId	Identifier for each event, unique across the application domain
event_severity	Situation/Severity	(Unknown, Information, Warning, Critical and Fatal)
event_situation	Situation/-SuccessDisposition	(Successful, Failed)
event_type	Situation/CategoryType	Type or class of events
correlation_id	ExtendedElements	Identifier to correlate events emitted during the same or related business transaction (unique within the transaction)
description	Situation/Message	General description of the event

the log data and user specified queries that represent precondition, occurrence and post-condition annotations for the goal of interest g. We can formalize the reduction process as a mapping

$$Q_{pre,g} \text{ X } Q_{occ,g} \text{ X } Q_{post,g} : E'_{i,j} \rightarrow R^{\mathrm{g}}{}_{i,j}$$

where, $E'_{i,j}$ is the set of events of component C_i for scenario S_j, represented in the common schema; $Q_{pre,g}$ is the query related to the precondition annotation of goal node g; $Q_{occ,g}$ is the query related to the occurrence annotation of goal node g; $Q_{post,g}$ is the query related to the postcondition annotation of goal node g; $\mathrm{R}^{\mathrm{g}}{}_{i,j}$ is the reduced set of log data for component C_i, for the scenario S_j and concerning goal node g. In other words, the conjunction of all the queries is a function that maps a set of events associated with *i,j* to a subset thereof.

$$F : R \rightarrow R' \text{ where } R'^{\mathrm{g}}{}_{i,j} \subset R^{\mathrm{g}}{}_{i,j}$$

The final step of the process in the proposed framework is the merging of events from all the sets $\mathrm{R}^{\mathrm{g}}{}_{i,j}$. By merging we mean that we merge events from sets $\mathrm{R}^{\mathrm{g}}{}_{i,j}$ that concern only one session of the scenario S_j, as in a real life system there may be many concurrent instances of a scenario S_j running (e.g., many instances of a *withdrawal* transaction). This step yields a set

$$R'_j = \bigcup_i^g R'^g_{i,j}$$

of log data that represents events associated with all goals g in the goal model and generated by all systems C_i for a given scenario S_j during a session s.

4.4 Unified Schema for Log Data

One of the contributions of this thesis is to develop a unified schema for the purpose of root cause analysis. In this chapter, we define and use a unified schema (Table 4.1) based on a subset of the WSDM Event Format (WEF) [73] which is a standardized version of the Common Base Event (CBE) schema [50]. The design of this unified schema format is driven by performance and costs concerns. The design of log table schema chosen should include enough information for the subsequent root cause analysis, but at the same time minimize the space needed to store the log data as well as allowing for an efficient retrieval of the log data of interest when dealing with gigabytes of log data.

WEF is an extensible XML format. It includes the three basic fields: *EventID*, *ReportTime* and *SequenceNumber*. It also contains three subsections: *Source*, *Reporter* (the reporting system can be different than the source system) and *Situation*. *Source* and *Reporter* are also extensible to include other application specific information. *Situation* includes information such as priority, severity, detailed message, etc. *Source* and *Reporter* sections include fields such as component name and id. WEF defines a standard set of priorities, severities, and situation categories such as *StartSituation*, *StopSituation* and *CreateSituation*.

The unified schema fields can be categorized into four categories: general, source system specific, event specific and correlation information. To further speed up the retrieval from the log data table contents, we add an extra timestamp field to represent in POSIX time format (total of seconds since January 1, 1970). Next, we index the table based on this field in order to improve the log data selection and retrieval. Table 4.1 contains the unified schema fields and their descriptions.

4.5 Log Reduction

The log reduction process consists of the following three steps: First, log data is parsed into a unified format and placed in a centralized database location. Second, we generate a set of semantic queries based on the OCL expressions annotating the goal models. Third, a log reduction step is applied using PLSI generating a collection of log data subsets corresponding to each annotation (precondition, occurrence and postcondition) of the goal model's nodes (goals and tasks). The log data subsets are then trivially merged into one set that contains the log entries of interest. Note that the size of this final set is less than or equal to the sum of the sizes of the individual subsets. This is because the same log entry may match more than one query and thus may be retrieved multiple times. The following sections describe in detail the three steps for the log reduction process:

4.5.1 Log Data Pre-Processing

As mentioned earlier, the log data is pre-processed by first transforming it from its native format into a unified schema, and then storing in a centralized database location. The mapping can be done based on semi-automated techniques as proposed by [5] or using mapping tables populated by subject matter experts. The details for the transformation are beyond the scope of this thesis.

4.5.2 Formation of Semantic Queries

The semantic queries are built by aggregating the goal model annotations with the user's provided reduction focus qualifiers. We refer to these queries as "semantic" since they are applied using a semantic based reasoner and the results are ranked. Note that the goal models representing the functional and non-functional requirements of the monitored systems are made ready prior to performing the log data querying and reduction. Furthermore, these models are pre-annotated with OCL expressions representing the precondition, occurrence and postcondition of each node of the goal model.

OCL-based Annotations

As described in Section 3.3.2, one class of goal models' annotations is based on OCL. We represent the *preconditions*, *occurrence* and *postconditions* constraints for each node in the goal tree using OCL expressions. Thus, we map the annotations of a node in the goal model to the *pre*, *body* and *post* constructs of an OCL formula. The remaining *def* and *inv* constructs of the OCL formula are set to NULL. The *context* section of the OCL formula is set to the name of the goal tree node.

As described in Section 3.3.2, each *pre/body/post* is a Boolean formula consisting of a logical collection of *atomic_logical_expressions* of the form:

$(e.attribute\_name\ comparison\_operator\ attribute\_value)$

where *attribute_name* represents an attribute of the log data, *comparison_operator* is a comparison operator of the form (=, <>, <, >, >=, <=), and *attribute_value* is a value of the attribute *attribute_name*.

As an example, the *a15 (Retrieve Credit Rating)* goal node of Figure 3.2 has a precondition and postcondition of,

context: a15
pre: self.precondition -> forAll(e | e.Description = 'Database LOGDATA started')
post: self.effect -> forAll(e | e.Description = 'Web Service Reply Containing Credit History/Rating Received')

Merging Annotations with User's Focus Qualifiers

Each of the OCL-based goal tree annotations (denoted as Q_a) is augmented with reduction *qualifiers* (denoted as Q_m), that are optional user imposed *"filters"*, to yield a more restrictive OCL expression (denoted as Q_r). Such *qualifiers* provide specific limit values that relate to the name of the process affected, the name of the server involved and the interval by which the analysis is applied on. These qualifiers aim to increase the quality and performance of the log reduction process. The annotations' augmentation algorithm is described as follows:

Algorithm I: Augmentation Algorithm

1. Add time restrictions: The Q_m annotation specifies a time interval within which the user is interested in monitoring the system or focusing his attention to. The augmentation algorithm adds the time restrictions specified in Q_m into Q_r as follows:
   T1 = Q_m.LowerTimeLimit
   T2 = Q_m.UpperTimeLimit

context Goal
pre/body/post: self.annotation -> for All(e | Q_r *AND* *e.report_time* $>= T1$ *and* *self.report_time* $<= T2$).

2. Disjunction of similar attributes: for each atomic expression EQ_a in Q_a of the form $(e.a_i\ op_1\ v_i)$, where e is an object and a_i is the i^{th} attribute on that object, op_1 is a comparison operator of the form $(<, >, >=, <=, <>)$, and v_i is a value for the attribute a_i; if there is an atomic expression EQ_m in Q_m of the form $(e.b_j\ op_2\ v_j)$ where $a_i{=}b_j$, then the two expression EQ_a and EQ_m are added to Q_r as follow:
$Q_r = Q_r\ AND\ ((e.a_i\ op_1\ v_i)$ OR $(e.a_i\ op_2\ v_j))$

3. Conjunction of remaining atomic expressions: all other expressions EQ_a in Q_a and EQ_m in Q_m are added to Q_r as follow:
$Q_r = Q_r\ AND\ EQ_a$
$Q_r = Q_r\ AND\ EQ_m$

The augmentation algorithm is applied repeatedly for each OCL goal model annotation of a goal tree node (precondition, postcondition or, occurrence annotation). In this respect, a set of three OCL expressions are generated for each goal tree node. For example, the occurrence of an event of type *REPLY* generated by the business service Credit_History goal node of Figure 3.2, is represented using the following OCL expression:

context a15
body: self.Constraint -> forAll(e | Q_a*)*
where Q_a *:: e.Address = '127.0.0.1' and e.Source_component = 'message broker' and e.Description.contains('request for credit rating received')*

The reduction focus qualifiers represent a set of constraints such as interval, host name, specific customer or users names. An example of an OCL expression representing the reduction focus qualifiers is the following:

context Focus_Qualifier
body: self -> forAll(e | Q_m*)*
where Q_m *:: e.Report_time >= '2010-08-22 10:00:00.0' and e.Report_time <= '2010-08-22 12:00:00.0' and e.Source_component = 'credit rating service'*

Using the augmentation algorithm, we augment the OCL expressions Q_a with Q_m, resulting in Q_r:

context a15
body: self.Constraint -> forAll(e | Q_r*)*

where Q_r *:: e.Report_time >= '2010-08-22 10:00:00.0' and e.Report_time <= '2010-08-22 12:00:00.0' and e.Address = '127.0.0.1' and (e.Source_component = 'credit rating service' or e.Source_component = 'message broker') and e.Description = 'request for credit rating received'*

We denote Q_r as the *semantic query*. Q_r is used to generate the query vector as described in section 4.5.3. Although syntactically Q_r is a typical OCL expression but it has some distinctive semantic characteristics. In particular, the formulation of the semantic queries emphasizes the inclusion of a distinctive set of keywords regardless of the order of these words. The performance of this algorithm is depends on the size of the goal model, the number of annotations on each goal node and the number of attributes. Since the remaining factors are fixed with respect to the monitored environment, the performance of this algorithm is a linear function of the size of the goal model.

4.5.3 Log Reduction based on PLSI

In Section 3.5.2, we describe the PLSI technique to reduce the log data. Using PLSI, we can estimate the word distribution per document which in turn can be used to find what documents are "close" to each other or relevant to a query. In the process of estimating the word distributions per document, a set of intermediary distributions are calculated using the Expectation-Maximization (EM) algorithm. In particular, the EM is applied by iteratively calculating $P(z/d, w)$ the probability distribution of a topic z given a document d and a word w using Equation 3.7 (expectation step), and then calculating the probability distributions $P(w/z)$, $P(d/z)$ and $P(z)$ using Equations 3.8, 3.9 and 3.10 (maximization step).

Without any loss of generality, we use a small subset of log data, shown in Table 4.2, to describe the intermediate steps of PLSI. The vocabulary extracted from this subset of log data consists of 17 terms. We apply 100 iterations of EM alternating steps to estimate the word-document distribution ($P(w/d)$). Each row represents a log entry from the subset in Figure 4.2. The following is the word-document distributions for the log entries *d1*, *d2* and *d3*:

P(w/d1)={*0.05, 0.05, 0.14, 0.14, 0.1, 0.1, 0.1, 0.09, 0.04, 0.09, 0, 0, 0, 0, 0, 0, 0.09*}
P(w/d2)={*0, 0, 0, 0, 0, 0, 0, 0, 0.12, 0.12, 0.24, 0.12, 0.12, 0.12, 0.12, 0.06, 0*}

Table 4.2: A Small Subset of Log Data

report_time	source_comp.	event_id	event_sev.	event_situat.	description
2010-07-05 17:47:44.153	Credit Service	112345	Information	success	Loan Application received and successfully validated for applicant Hamzeh
2010-07-05 17:47:44.153	soapUI	223342	Debug	success	Date: Mon, 05 Apr 2010 17:52:53 GMT, localhost
2010-07-05 17:47:44.153	soapUI	223456	Debug	success	"http://www.hamzeh-.org /uno /getCreditRating"

P(w/d3)=$\{$*0, 0, 0, 0, 0, 0, 0, 0, 0.12, 0.12, 0.24, 0.12, 0.12, 0.12, 0.12, 0.06, 0*$\}$

By treating the following query as a document, we can obtain the corresponding word-document probability distribution.

context a15
body: self.Constraint -> forAll(e | Q_a*)*
where Q_a *:: e.Source_component = 'soapui' and e.Description.contains('submit loan application')*

The corresponding probability distribution is:

P(w/q)=$\{$*0.05, 0.05, 0.14, 0.14, 0.09, 0.09, 0.09, 0.09, 0.05, 0.09, 0, 0, 0, 0, 0, 0, 0.09* $\}$

Next, we rank documents that are similar to each other based on having similar $P(w/d)$ word-document distributions. As described in Section 3.5.2, we measure the distribution similarity between two documents *d1* and *d2* using the *Jenson-Shannon* (*JS*) divergence. The outcome of the *JS* divergence is a value bounded by 0 and 1 where 0 indicates total similarity. The higher the divergence value the more the two documents are dissimilar. In this respect, we consider two different approaches for ranking the "closeness" of log data entries with respect to a query:

- The first approach is based on choosing the top r relevant documents (with lowest distribution divergence).

- The second ranking criterion is based on selecting all documents that have a distribution divergence lower than a threshold value α.

The values of the parameters r and α have a direct but varied impact on the size of the reduced set and the number of recalled/missed positives. We evaluate this impact in the experimental section 4.6.1. The results of comparing the query distribution with the distributions of the three log entries are the following: *[0, 0.82, 0.82]* indicating that the query is the closest to the first entry.

4.5.4 Amalgamation of the Log Data Subsets

As a result of the previous step, each node in the goal model is now associated with three subsets of filtered log data corresponding to its annotations (precondition, occurrence and postcondition). The last step consists of merging the filtered log data subsets associated with each node (tasks and goals) into one global view of log data. This is done based on Algorithm II shown as follows.

Algorithm II: Log Data Amalgamation Algorithm

```
node = top_goal
logData = empty_list
timeStep= 1
node = top_goal
visit(node) visitGoal(Node node, int timeStep) {
 precondition = node.PrecondID + timeStep
 if (!node.PrecondQueryResult is Empty)
  logData.Append(node.PrecondQueryResult)

 children = node.getChildren;
 for (int i=0; i¡ children.size; i++) {
  child = children.elementAt(i)
  if (child is “task”))
   visitTask(child);
  else if (child is (“goal”))
   visitGoal(child);
  }
  postcondition = goal.getPostconditionID + (timeStep);
  if (!goal.getPostconditionQueryResult is Empty)
   logData.Append(goal.getPostconditionQueryResult) }

visitTask(Task task) {
  precondition = task.getPrecondID + (timeStep);
  if (!task.getPrecondQueryResult is Empty)
   logData.Append(task.getPrecondQueryResult)
  if (!task.getOccurQueryResult is Empty)
   logData.Append(task.getOccurQueryResult)
  if (!task.getPostconditionQueryResult is Empty)
   logData.Append(task.getPostconditionQueryResult)
}
```

Algorithm II generates (*logData*) which is a list of log data entries. The merging in Algorithm II is done according to the pre-order traversal of the goal model. Thus, for each task, the log data subsets corresponding to the precondition, occurrence and postcondition annotations are merged in that order. The list of log entries (*logData*) represents a compact view of the log data with reduced noise (but not completely removed) and containing most or all of the events of interest.

Algorithm II is a tree traversal which typically is of the order *O(n)* where n is the total number of nodes in the tree. In the case of using recursion, the algorithm will consume memory in the order of the deepest level, which on a balanced tree it would be log(n).

4.6 The Framework in Action

In the set of experiments described in this section, we applied the proposed framework to reduce the log data corpus from the original size of 9856 log entries to a subset of 2079 log entries while still including all the positives. In these experiments, we use the *Loan Application* goal model that consists of 2 goals and 9 tasks (shown in Figure 3.2) as a basis for the log reduction. In Table 3.1, we list the set of nodes that make up the *Loan Application* goal model as well as the corresponding annotations for the *Apply For Loan* goal model. The experimental setup is based on the loan application's motivating scenario and test environment described in Section 3.7.

4.6.1 Empirical Evaluation

The first experiment evaluates the quality of the log data reduction using the proposed framework, while the second experiment is a study of its performance.

Loan Application Scenario

To evaluate the functionality of the reduction framework, we applied the framework to the log data generated by the Loan Application described earlier in this section and we measured the resulting reduction, precision and recall. The reduction is measured as the ratio of the filtered out number of documents over the original corpus size. The precision is measured as the ratio of true positives over the filtered log data, and the recall as the ratio of the retrieved true positives over the total number of positives in the original corpus.

The guiding scenario for log reduction is based on the goal model shown in Figure 3.2 and consists of 2 goals and 9 tasks. Our implementation of PLSI is based on the publicly available *mltool4j* project [1]. The log database table contained 9856 log entries generated from all systems in our test environment.

To build a list stop words for the purpose of extracting the relevant log data, we used the following criteria:

1. Use the top 120 most common English words.
2. Include conjunctions (for, and, nor, ...), articles (a, an, the), determiners (this, that, these, ...)
3. Include the top 30 terms that appeared in the log data and that we considered as not essential in the formulation of the goal model annotated queries.

Prior to running this experiment, we built the requirement goal models and formulated the OCL expressions annotations as described in section 4.5.2. These queries are treated as documents and included in the general set of log data used in building the probabilistic model using PLSI as described in section 4.5.3 by alternating the expectation and maximization steps in order to estimate the marginal probability distributions described in equations 3.7, 3.8, 3.9 and 3.10. We set to 1000 the number of iterations of the expectation and maximization steps. The number of topics is set to 100 and is empirically chosen since it results in best reduction/recall results for this specific corpus of log data.

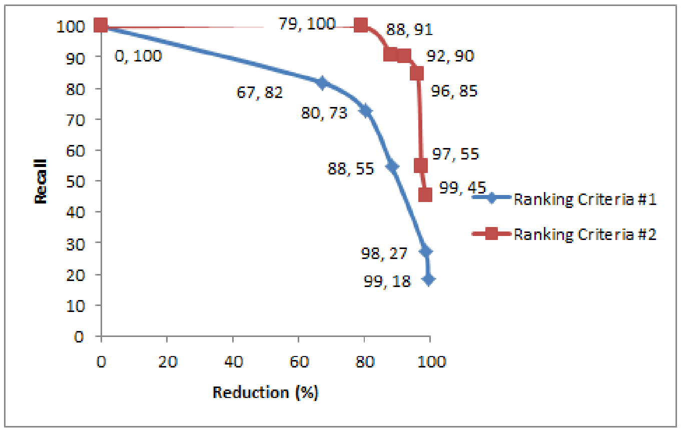


Figure 4.2: Recall (y-axis) versus reduction (x-axis) using 100 topics, 1000 iterations

After the probabilistic model was built, we calculated the relevance of documents with respect to each query by applying the *Jenson-Shannon* divergence among the words over documents ($P(w/d)$) distributions for the query and each document (equation 3.11).

We ranked the relevance of the log data using the two criteria described in section 4.5.3. The first ranking criteria consists of selecting the first r documents that have the smallest distribution divergence with respect to each query's distribution, where $r \in \{20, 50, 100, 500, 1000, 3000\}$. The second ranking criteria consists of selecting documents whose distribution divergence is above a threshold value $\alpha \in \{0.01, 0.05, 0.07, 0.1\}$. Note that some of the log data subsets may contain the same log entries, thus the size of the merged subsets is less than the total sum of the sizes of the individual sets.

Figure 4.2 shows that a 100% recall was achieved with 79% reduction using the first ranking criteria (choosing the nearest r documents to a query). We also note that the reduction and precision results obtained using the first ranking criteria are consistently superior to those using the second ranking criteria, where documents with a distribution divergence below a certain threshold value α are retrieved. The reason is that different queries have different $P(w/d)$ distributions and thus different optimal divergence threshold values, whereas in the first criteria, the nearest r documents are always retrieved regardless of the shape of the distribution.

Our results are compared to a study by Al-Mamory [4] who used log data analysis to find the root causes that are behind the false alarms triggered by intrusion detection systems. A 70% reduction in alarms was achieved while missing only 955 alarms when Al-Mamory et al. applied their framework to the DARPA dataset. The results can vary depending on the formulation of the queries and the quality of the log data; however, our results show an improvement both in the reduction and recall rates compared with the aforementioned study.

4.6.2 Performance Evaluation

In order to study the scalability of the log reduction approach, we performed an initial set of measurements based on the log data corpus of 9856 entries generated by the loan application business process and services. In particular, we are interested in finding the impact of the topics and iteration numbers on the overall performance of the reduction approach. We acknowledge that, due to the limitations of the scenario used, the size of the log data corpus used in this study is relatively small for a typical scalability study; however, the intention of this experiment is to provide an initial evaluation of the behavior of the processing time and its correlation with of number of iterations performed as well as the choice of the number of topics. The machine used for this experiment is based on an Intel Core 2 Duo CPU (2.20 GHz) with 4 GB of system memory, and running on Windows 7 Professional. The results of this experiment are plotted in Figure 4.3. In particular, we see a linear relationship between the log reduction processing time and the number of EM algorithm iterations for the same number fixed of topics (25, 50 and 100).

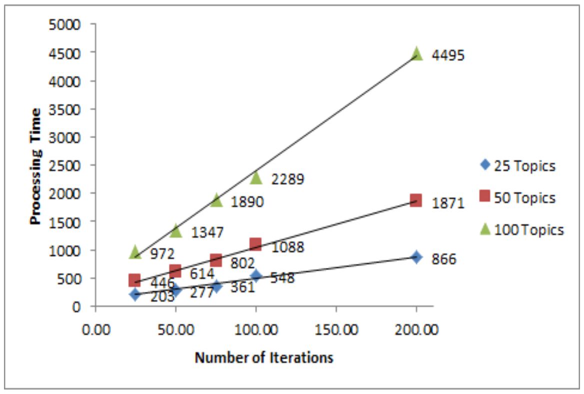


Figure 4.3: Processing Time versus the Number of Iterations

4.7 Summary

This chapter presents a log reduction framework for component based systems. First, given a collection of goal models representing different functional and non-functional requirements for the systems being monitored, events of interests such as preconditions, occurrences and effects for goals/tasks of the goal model are added as logical expression annotations. Next, log data incoming from the monitored applications are transformed and stored in a centralized database table. Using PLSI, the queries embedded in the goal tree nodes annotations are applied to extract a set of log data subsets. Finally, the extracted log data subsets are merged into a global view that represents a reduced version of the overall log data. Experimental results indicate the feasibility of the approach and that the proposed framework is capable of significantly reducing the size of the log data to be considered for a given task/objective, allowing the system operators and administrators to focus their attention on the specific subset of log data that contains the necessary information relating to the specific requirement being investigated. Further scalability experiments using larger data sets can be applied in the future to evaluate the scalability of the proposed approach. One of future enhancements of this approach is to parallelize the log data reduction process by using techniques such as Map-Reduce, where multiple parallel reduction processes are applied to different intervals of log data and then the outcomes of all processes are merged in one set.

Chapter 5

Log Filtering

In this chapter, we present a framework that assists the root cause identification and system diagnostics in software systems by analyzing streaming log data. In particular, this framework is applied to the log data corpus and filters out a set of reduced log data extracts that can be used to generate an audit trail useful for RCA tools.

We propose two alternative filtering techniques. The first is based on Latent Semantic Indexing (LSI) and second combines LSI and regular expressions in order to improve on the precision and recall and reduce some of the limitations of traditional LSI. The filtered log data can be used to help produce possible diagnoses and assist in the root cause analysis. This log filtering framework uses requirement goal models annotated with *precondition*, *occurrence*, and *postcondition* predicates as described in Section 3.2. The use of goal model annotations is central to the process of log filtering. In this respect, when the user experiences a system failure, the goal models' annotations are used to generate queries that yield reduced log data, hence provide an initial focus to the human operator towards identifying the root cause of the problem being observed. In particular, these annotations can either be used to generate SQL queries to be applied against the log data pool, or can be used for identifying related log entries by using Latent Semantic Indexing (LSI). To evaluate the proposed framework, we have applied it to a service oriented system we have built using a set of commercial-off-the-shelf products (see Section 3.7 for details).

This chapter is structured as follows. Section 5.1 presents the motivation and relevance of this work to the thesis. Section 5.2 provides a description of the architecture of the log filtering framework. Section 5.3 provides a formalization of the four main processes that the proposed framework consists of. Section 5.4 describes the process of the log filtering based on two techniques: SQL and LSI. Section 5.6 describes the empirical evaluation of the framework. Section 5.7 summarizes the contributions and conclusions.

5.1 Motivation

A common approach to RCA, adopted in this thesis, is to analyze log files and identify deviations from normal or expected behavior. These deviations are then examined so that potential causes can be identified. RCA performance is limited by the size of the logs considered. This problem becomes more apparent when large software systems are concerned. These systems are comprised of many components each of which may have different logging and monitoring mechanism so that, the logged data may often be too many and too complex for a human operator to analyze. In this respect, log filtering becomes the problem of retrieving only log data entries that are relevant to a particular query or event of interest.

It is important to note the difference between the log filtering and log reduction that was described in Chapter 4. Log reduction focuses on reducing the noise in the log data corpus by removing data that are not relevant to a particular diagnostic model. However, log filtering selects only relevant entries to an event or based on a query. In this chapter, we describe a set of approaches for log filtering based on traditional file search and database querying techniques as well as the semantic analysis-based for information retrieval.

5.2 Overall Architecture

As illustrated in Figure 5.1, the proposed framework has three components. The first component is log data normalization component. It transforms streaming log data into a common format and stores it into a centralized database table. The second component is a log data aggregation component. It uses a set of constraints defined in object constraint language (OCL hereafter) to generate an equivalent set of queries by merging them with user's focus qualifiers. The third component uses the generated queries to extract a subset of log data from the log database. For the purpose of identifying the best log data querying technique, we use two different techniques that we compare/contrast: first, the traditional database query language (SQL), and second, the latent semantic indexing which is traditionally applied for document indexing and retrieval. The output of this framework is a collection of highly cohesive log data sets that can be used to verify/deny a goal model by using a RCA algorithm to produce a tractable set of diagnoses showing which components

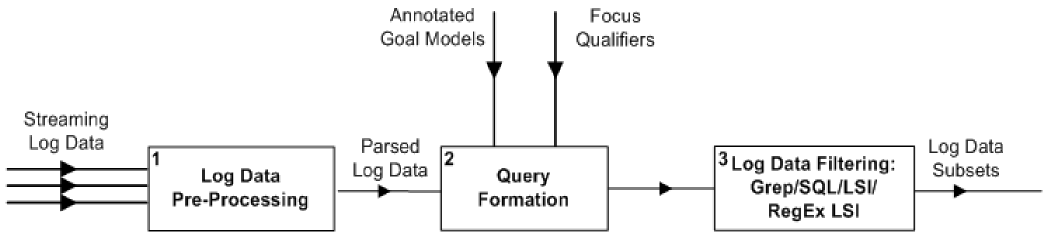


Figure 5.1: Logical Architecture of the Log Filtering Framework.

may have failed. The following sections describe in detail the inputs and output of the framework.

Log Data

The first input to the monitoring framework is the natively generated log data. Log data are obtained as sequences of events from different loggers. Natively generated log data goes through a preprocessing phase which consists of two steps: first, designing a unified log schema, and second, building a set of model transformations.

Similarly to the unified schema for reduction described in Section 4.4, we propose a unified schema, that we call *LogData*, to contain fields that are common to the different logging systems in our test environment as well as fields that we deem necessary for our analysis. We use as a reference in our schema design the WSDM Event Format (WEF) standard which is an OASIS standardized version of the IBM's common based event (CBE) [50, 73]. We also add to *LogData* some fields that we considered to be essential but did not appear in WEF such as the *EventTypeID* field from Windows Event Viewer which represents a class of events and is different from the *EventRecordID*. *LogData* contains the following fields: *EventRecordId*, *ReportTime*, *SourceComponent*, *SourceComponentAddress*, *EventSituation*, *CorrelationId*, *EventTypeId* and *Description*. The second step consists of building model to model transformations to map fields from individual log sources into the unified schema.

Annotated Goal Models

The second input to the framework is a set of annotated goal model(s), each representing the requirements of the monitored applications. Goal models can be developed either manually by the system analysts or through reverse engineering the source code [109] and are built a priori to using the monitoring framework. The goal models used for log filtering are annotated with *precondition*, *occurrence*, and *postcondition* predicates that are used to generate queries and extract a collection of log data subsets that in turn are used to verify the particular properties of the goal model, or equivalently the functional or non-functional system requirement, being considered.

A set of examples based on our use case scenario of loan application are the following:

Precondition: *self.address = 'Machine2' and self.event_id = 'POST' and self.event_ severity = Severity::Information and (self.reporter_component = 'CA Wily' or self.reporter_component = 'Loan Application')*

Occurrence: *self.address = 'Machine1' and self.event_id = 'PROCESSED' and self. event_severity = Severity::Information and (self.reporter_component = 'CA Wily' or self- .reporter_component = 'Loan Application')*

Postcondition: *self.address = 'Machine1' and (self.event_id = 'APPROVED' or self. event_id = 'REJECTED') and self.event_severity = Severity::Information and (self.reporter- _component = 'CA Wily' or self.reporter_component = 'Loan Application')*

Monitoring Focus Qualifiers

The third input is the collection of monitoring *qualifiers* that represent the user's particular points of interest such as time interval, server name, IP address, and user names, etc. The monitoring *qualifiers* are expressed using OCL expressions. An example of such a monitoring *qualifier* for events related to "John Smith" and occurring between timestamps [T1, T2] is the following:
self.report_time >= T1 and self.report_time <= T2. and self.description.contains('John Smith')

Framework Output

After post-processing the results of the SQL queries, the framework produces subsets of interpreted log data as a stream of logical literals. The presence (or lack of) of the logical literals represents the occurrence (or the non-occurrence respectively) of the set of events of interest (i.e., preconditions, occurrence and effects of the goal model nodes). This generated stream of literals is most useful for a recent RCA technology in software systems based on the propositional satisfaction of system properties [104, 111]. The RCA frameworks described in Chapters 7, 8 and 9 use log filtering techniques described in this chapter.

5.3 Formalization of the Interpretation Framework

In this section, we provide the formalization of the basic components of the log filtering process. Let $\mathcal{D}$ be a distributed system being diagnosed. We represent such a system as a collection of components $C_1, C_2, \ldots C_m$ that is:

$$\mathcal{D} = \{C_1, C_2, \ldots C_m\}$$

In this context, a component is an artifact that can represent a software application (e.g., a service), a hardware related component (e.g., a server logger) or an infrastructure related device (e.g., a network router logger).

We also consider that the system may provide different functionalities where each functionality can be exercised by several usage scenarios. Let $S_{i,j}$ be the j^{th} scenario for i^{th} component. For each scenario $S_{i,j}$ that is applied, each component C_i generates a set of events:

$$E_{i,j} = \{\mathrm{e^k}_{i,j} | k \in \{1, 2 ... n\}\}$$

where $\mathrm{e^k}_{i,j}$ is the k^{th} event generated by component C_i, and n is the total number of events generated by component C_i when triggered by scenario $S_{i,j}$.

We also consider that event $\mathrm{e^k}_{i,j}$ is represented by a set of attribute-value pairs:

$$\mathrm{e^k}_{i,j} = \{<\mathrm{a^i}_1,\ \mathrm{v^i}_1>,\ <\mathrm{a^i}_2,\ \mathrm{v^i}_2>, ... <\mathrm{a^i}_m,\ \mathrm{v^i}_m>\}$$

where m is the number of attributes in the events generated by the logger of component C_i.

The first step of the process that can be applied as part of the proposed framework is a normalization step where events generated by different loggers of the different components of the system are normalized in a common schema format. This step can be formalized as a mapping

$$E_{i,j} \rightarrow E'_{i,j}$$

where $E'_{i,j}$ represents events emitted by component C_i for scenario S_j transformed for a common meta model.

The second step of the log filtering process is the selection of events of interest (L). We can formalize the filtering process as a mapping

$$E'_{i,j} \text{ X } Q_{pre,g} \text{ X } Q_{occ,g} \text{ X } Q_{post,g} \rightarrow \mathrm{R^g}_{i,j}$$

where, $E'_{i,j}$ is the set of events of component C_i for scenario S_j, represented in the common schema, $Q_{pre,g}$ is the query related to the precondition annotation of goal node g, $Q_{occ,g}$ is the query related to the occurrence annotation of goal node g, $Q_{post,g}$ is the query related to the postcondition annotation of goal node g, and $\mathrm{R^g}_{i,j}$ is the reduced set of log data for component C_i, for the scenario S_j and pertaining to goal node g.

5.4 Log Filtering

5.4.1 Formation of Goal Annotations

Nodes in the goal models are annotated with OCL expressions Q_a pertaining to *preconditions*, *occurrence* and, *postcondition* constraints. The OCL expressions used to populate the annotations are described in Section 3.3.2. Each of these OCL goal tree annotations Q_a can be augmented with user's filtering *qualifiers* Q_m, that are optional user imposed filters, to yield a more restrictive OCL expression Q_r. Such *qualifiers* provide specific limit values that relate to the name of the process affected, the name of the server involved and the time interval by which the analysis is applied on. These qualifiers aim to increase the performance and accuracy of the log filtering process.

The augmentation algorithm for filtering (*Algorithm III*) consists of the following three steps:

1. Add time restrictions: Q_m contains a time interval that is of interest to the user. Only data within this interval is considered in the filtering process. The augmentation algorithm adds the time restrictions specified in Q_m into Q_r as follows:
   $Q_r = Q_r$ *AND self.report_time* $>= T1$ *and self.report_time* $<= T2$.

2. Disjunction of similar attributes: for each atomic expression EQ_a in Q_a of the form (*self.*a_i op_1 v_i), where a is an attribute, op_1 is a comparison operator of the form ($<, >, <=, >=, <>$), and v_i is a value for the attribute a_i; if there is an atomic expression EQ_m in Q_m of the form (*self.*b_j op_2 v_j) where a_i=b_j, then the two expression EQ_a and EQ_m are added to Q_r as follows:
   $Q_r = Q_r$ *AND* ((*self.*a_i op_1 v_i) OR (*self.*a_i op_2 v_j))

3. Conjunction of remaining atomic expressions: all other expressions EQ_a in Q_a and EQ_m in Q_m are added to Q_r as follows:
   $Q_r = Q_r\ AND\ EQ_a$
   $Q_r = Q_r\ AND\ EQ_m$

The augmentation algorithm is applied repeatedly for each OCL goal model annotation of a goal tree node (precondition, post-condition or, occurrence annotation). In this respect, a set of three OCL expressions are generated for each goal tree node. For example, the occurrence of an event of type *REPLY* generated by the loan application goal node of Figure 3.2, is represented using the following OCL expression:

Q_a *:: self.address = 'localhost' and self.description. contains('credit') and self.description. contains('receive')*

The monitoring focus qualifiers represent a set of constraints such as time interval, host name, specific customer or users names. An example of an OCL expression representing the monitoring focus qualifiers is shown is the following:

Q_m *:: self.report_time >= T1 and self.report_time <= T2 and self.source_component = 'soapUI'*

Using the augmentation algorithm, we augment the OCL expressions $Q1$ with $Q2$, resulting in Q:

Q_r *:: self.report_time >= T1 and self.report_time <= T2 and self.address = 'localhost' and self.source_component = 'soapUI' and self.description.contains('receive') and self.description .contains('credit')*

Similarly to algorithm I, this algorithm is *O(n)* where n is the total number of nodes in the goal model tree.

5.4.2 Log Filtering based on SQL Query Generation

The first log filtering technique we consider is based on applying a generated SQL query to the database table containing the log data. In this process, each augmented OCL expression for a goal tree node is used to generate one SQL query which is applied against the log data pool.

The SQL query generation algorithm (Algorithm IV) is described as follows:

1. Create skeleton for SQL Query. The skeleton is of the form:

   SELECT $*$ FROM <Log DataBase Table> WHERE <generated SQL query expression>

2. Create the <generated SQL query expression> from the logical_expressions in a *select* operation in the body of the corresponding OCL formula for the given goal tree node. The creation of such SQL query expression is a transformation that:

   a) Maps each OCL logical_expression of the form:

   $(self.attribute\_name$

   $comparison\_operator$

   $attribute\_value)$

   to

   $attribute\_name$

   $comparison\_operator$

   $attribute\_value$ SQL expression and,

   b) Maintains the structure of the Boolean expression in the OCL expression to the corresponding SQL query expression. For example, the OCL expression Q_r:

   Q_r :: self.report_time > T1 and self.report_time < T2 and self.source_component = 'soapUI' and self.description.contains('receive loan application')

   leads to the generation of following SQL expression:

   $SELECT \star FROM$ $[EVENTS].[LogData]$
   $WHERE$ $report\_time > T1$ AND $report\_time < T2$
   AND $source\_component = 'soapui'$
   AND *description like %receive loan application%*

The performance of this algorithm depends on the size of the monitored environment which directly impact the number of nodes in the corresponding goal model.

Table 5.1: Term-document Matrix

Log Entries	receive	hamzeh	getcreditrating	validate	loan	debug	service	application	credit	soapui
2010-04-05 13:52:51.684105, Credit Service, INFORMATION, SUCCESSFUL, Loan Application received and successfully validated for applicant Hamzeh, 1111, localhost	1	1	0	1	1	0	1	1	1	0
2010-04-05 13:52:54.0, soapUI, DEBUG, Date: Mon, 05 Apr 2010 17:52:53 GMT, localhost	0	0	0	0	0	1	0	0	0	1
2010-04-05 13:52:50.0, soapUI, DEBUG, “http://www.hamzeh.org/uno /getCreditRating”, localhost	0	1	1	0	0	1	0	0	0	1

5.4.3 Log Filtering based on Latent Semantic Indexing

The second log filtering technique we consider is based on LSI [24]. This technique has the advantage that in order to increase the recall of the process, we allow users to use more relaxed queries and to obtain ranked results as opposed to using SQL queries to obtain one fixed result. LSI is a technique commonly used in web search for ranking and indexing documents (here the log data represent the documents searched and indexed).

We describe the theoretical foundation of the LSI in Section 3.5.1. In this section, we describe the LSI application to a subset of the log data and the vocabulary. In particular, we show the intermediate steps of LSI to filter log data and using the SELECT query example shown in the previous section. Typically, the two dimensional matrices built in the process of LSI, such as the term-document matrix, are very large and consist of hundreds or thousands of columns and rows. In Table 5.1, we show the term-document for the subset of log data/vocabulary but in transposed form for display purposes.

The next step consists of factoring the term-document matrix M using SVD into a product of three matrices ($U_{m.n} * S_{n.n} * V^T_{n.n}$) where m is the size of the vocabulary and n is the size of the log data corpus.

$$M = \begin{bmatrix} 0.34 & -0.16 & 0.10 \\ 0.47 & 0.18 & -0.53 \\ 0.12 & 0.34 & -0.63 \\ 0.34 & -0.16 & 0.10 \\ 0.34 & -0.16 & 0.10 \\ 0.18 & 0.58 & 0.36 \\ 0.34 & -0.16 & 0.10 \\ 0.34 & -0.16 & 0.10 \\ 0.34 & -0.16 & 0.10 \\ 0.18 & 0.58 & 0.35 \end{bmatrix} \begin{bmatrix} 2.71 & 0.0 & 0.0 \\ 0.0 & 2.21 & 0.0 \\ 0.0 & 0.0 & 0.85 \end{bmatrix} \begin{bmatrix} 0.93 & -0.37 & 0.09 \\ 0.13 & 0.53 & 0.84 \\ 0.35 & 0.77 & -0.54 \end{bmatrix}$$

In this example, the vocabulary (columns of Table 5.1) consists of $m = 10$ terms. The log data corpus consists of $n = 3$ log entries. If we set the number of concepts to $k = 2$, we can apply a rank reduction to $(U_{m.n} * S_{n.n} * V^T_{n.n})$, thus obtaining: $(U_{m.k} * S_{k.k} * V^T_{k.n})$.

$$M^* = \begin{bmatrix} 0.34 & -0.16 \\ 0.47 & 0.18 \\ 0.12 & 0.34 \\ 0.34 & -0.16 \\ 0.34 & -0.16 \\ 0.18 & 0.58 \\ 0.34 & -0.16 \\ 0.34 & -0.16 \\ 0.18 & 0.58 \end{bmatrix} \begin{bmatrix} 2.71 & 0.0 \\ 0.0 & 2.21 \end{bmatrix} \begin{bmatrix} 0.93 & -0.37 & 0.09 \\ 0.13 & 0.53 & 0.84 \end{bmatrix}$$

The next step is to calculate the new vector representation of each document (log entry) with respect to the concepts. Remember that the original vector representation of each document shown in Table 5.1 was in function of vocabulary terms. The new set of vector representation is given by equation 3.4. The resulting document-concept matrix is as follows:

$$\begin{bmatrix} 0.93 & -0.37 \\ 0.13 & 0.53 \\ 0.35 & 0.77 \end{bmatrix}$$

We calculate the transposed query vector representation for the example ($SELECT \star FROM\ [EVENTS].[LogData]\ WHERE\ report\_time > T1\ AND\ report\_time < T2\ AND\ source\_component = 'soapui'\ AND\ description\ like\ \%receive\ loan\ application\%$) as follows:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{bmatrix}$$

We calculate the query representation with respect to the reduced concept matrix based on the equation 3.5:

$$\begin{bmatrix} 0.25 & 0 - .15 \end{bmatrix}$$

We calculate the similarity matrix based on the cosine function in equation 3.6:

$$\begin{bmatrix} 0.99 & -0.29 & 0.11 \end{bmatrix}$$

This similarity matrix indicates correctly that the query is closest to the first log entry in Table 5.1. Note that the output of the cosine based similarity function range from $[-1; +1]$, where the higher the value, the more similar the two documents are.

5.5 Log Filtering based on Regular Expressions and Latent Semantic Indexing

In the previous section, we applied LSI for the purpose of filtering log data based on a set of queries annotating goal models. The application of LSI to log analysis showed better results and flexibility when compared to traditional techniques such as SQL and Grep; however, LSI suffers from two inherent problems that in some cases lead to degradation in the quality of filtered log data. The first problem is the handling of polysemy (words with multiple meanings). Each occurrence of a word is treated as having the same meaning. An example, consider the following expressions:

L1: "New application for business loan submitted by customer John Smith"
L2: "Loan business process APPLY_FOR_LOAN application started"

The keyword "application" has different meanings (request in $L1$ and service in $L2$). LSI does not distinguish between the different meanings of the same word. This is because

LSI treats document using the bag of words model (BOW), where a text is represented as an unordered collection of words, where the vector representation of a document is simply an average of all the keyword's occurrences (even when a keyword has other polysemous meanings). Let us take the example of the two expressions: "application for a loan" and "loan application". In the first expression, "application" refers to one user's request to get a loan, whereas in the second expression "application" refers to a financial software program. LSI does not distinguish between the two meanings of "application" because the original order of the keywords is not preserved in the LSI analysis.

In this chapter, we present a hybrid approach that combines traditional LSI with regular expressions in order to reduce the impact of the LSI shortcomings. In order to loosen the BOW assumption in LSI, we propose the building of the semantic space (term-document matrix) by using n-grams expressions instead of terms. N-grams (a contiguous sequence of n items from a given sequence of text or speech) can be used to represent a collection of two or three words as well as preserve the order of the appearance of these words. We use regular expressions to represent N-grams; hence the naming of this approach: RegEx based LSI. Since a multi-term expression has more context than a single term, we expect the polysemy to be handled better by the hybrid approach. By taking the example of the keyword "application" in the expressions *L1* and *L2*, we can distinguish between the two meanings of "application" if we populate the semantic space in LSI using the occurrences of two different terms "loan...application" and "application...loan", thus distinguishing between the two meanings of the keyword "application".

5.5.1 Filtering Process

In section 3.5, we describe the LSI and how it is applied to filtering log data. In this section, we describe the proposed hybrid LSI using regular expressions. The main motivation behind the proposal of a regular expression based indexing and retrieval approach (that we call regex LSI) is to improve on the BOW assumption in LSI. Regular expression (regex) LSI replaces the traditional term-document semantic space used in LSI with a regular expression-document semantic space. In the following sections we explain the process of regular expression extraction and the regular expression LSI approach.

Regular Expressions Extraction

Regular expressions represent context-independent patterns that encapsulate a specific ordering of characters. Regular expressions can be used to represent patterns of single characters or concatenations of characters. An example of a regular expression is *[bcr]at* that finds a match in texts containing *bat*, *cat* or *rat*. The regular expressions used in this work are of the form:

$$keyword_1.*keyword_2.*\ldots*keyword_n \tag{5.1}$$

where n is the number of keywords ranging from 1 (pattern reduced to a term) to n. The optimal number of keywords varies depending on the size of the documents being indexed. In the case where documents represent log data, our experimental evaluation indicates that patterns consisting of 2-3 keywords contribute to lower false positives without any increases in false negatives. The generation of the regular expression is done based on an algorithm that builds all possible regular expressions from the terms extracted from each document while keeping the order of the terms as they appear in the document. This process is concluded when all queries associated with the anti-goal model are processed. An example of a regular expression with 3 keywords is [$credit.*request.*submit$]. The generation of the regular expressions is based on an algorithm that generates permutations of keywords. Heuristics can be applied to limit the number of expressions that are generated.

Algorithm V: Regular Expression Extraction Algorithm

```
$N$ = number of keywords in a pattern
$Q$ = set of all the queries (which has a total of $3n$ where n is the number of nodes)
$K$ = number of terms
For each query $q$ in $Q$ {
  $keywords$ = tokenize $q$ into a list of $K$ terms (ordered as they appeared in $Q$)
   refine keywords by removing capitalization, applying stop list and stemming
   for each $keyword_i$ in the ordered list of keywords {
    build_pattern(keyword$_i$, i, N) }
}
build_patterns(keyword$_i$, position, size_ of_pattern)) {
  let $N$ be the $size\_of\_pattern$
  let $P$ be the set of patterns of size $N$
  let $i$ = position of keyword$_i$ in the set ordered set of keywords
  if ($N$=2) then {
   for ($j = i + 1$ to $K$) {
    $P.add(keyword_i\ .*\ keyword_j)$
   return $patterns$ }
  else
   for ($j$ = 1 to $N$) {
    $P$.add(keyword$_i$ .* build(keyword$_i$, $N-1$) }
  return $P$
}
```

Algorithm V builds all possible regular expressions from the terms extracted from each document while keeping the order of the terms as they appear in the document. This process is concluded when all queries associated with the anti-goal model are processed. An example of a regular expression with 3 keywords is $[credit. * request. * submit]$. The performance of this algorithm is of the order $O(n.N!)$, where N is the size of a pattern. Since N is a constant that typically is equal to 2 or 3, the algorithm is in the order of $O(n)$.

Indexing using the regex-document matrix

The next step is to create the regular expression-document matrix. The columns of the matrix represent a set of signature vectors each corresponding with a log entry. Rows denote the set of regular expressions generated earlier. The signature row vector for each expression r is populated using a term frequency weighting function wf (Equation 3.3). Table 5.2 is the semantic space for three log entries populated using the selected *tf-idf* weighting frequency. Note that in this example, we generated regular expressions by tokenizing keywords from the query only. We show the corresponding query as follows:

Q = ((source component is soapUI) ∧ (loan application received from soapUI on localhost))

Similarly to traditional LSI, we apply a rank lowering transformation of the regex-document matrix into a lower dimensionality matrix using the SVD algorithm. Dimension reduction is an important step to aggregate expressions that belong to the same context. We apply SVD to the semantic space in Table 5.2:

$$M = \begin{bmatrix} 0.0 & 0.0 & 0.0 \\ -0.57 & 0.0 & 0.0 \\ -0.0 & -0.0 & 1.0 \\ -0.57 & -0.0 & 0.0 \\ -0.0 & -0.0 & 0.0 \\ -0.0 & -1.0 & 0.0 \\ -0.0 & -0.0 & 0.0 \\ -0.0 & -0.0 & 0.0 \\ -0.57 & -0.0 & 0.0 \\ -0.0 & -0.0 & 0.0 \end{bmatrix} \begin{bmatrix} 1.73 & 0.0 & 0.0 \\ 0.0 & 0.71 & 0.0 \\ 0.0 & 0.0 & 0 \end{bmatrix} \begin{bmatrix} -1.0 & 0.0 & 0.0 \\ 0.0 & -0.71 & -0.71 \\ 0.0 & -0.71 & 0.71 \end{bmatrix}$$

Table 5.2: Term-document Matrix

Log Entry	application.*soapui	loan.*application	receiv.*localhost	application.*receiv	loan.*localhost	soapui.*localhost	loan.*soapui	receive.*soapui	loan.*receive	application.*localhost
2010-04-05 13:52:51.684105, Credit Service, INFORMATION, SUCCESSFUL, Loan Application received and successfully validated for applicant Hamzeh, 1111, localhost	0	1	0	1	0	0	0	0	1	0
2010-04-05 13:52:54.0, soapUI, DEBUG, Date: Mon, 05 Apr 2010 17:52:53 GMT, localhost	0	0	0	0	0	0.5	0	0	0	0
2010-04-05 13:52:50.0, soapUI, DEBUG, "http://www.hamzeh.org/uno/getCreditRating", localhost	0	0	0	0	0	0.5	0	0	0	0

In this example, the number of 2-gram regular expressions (columns of Table 5.2) is: $m = 10$ terms. The log data corpus is $n = 3$ log entries. If we set the number of concepts to $k = 2$ and applying a rank reduction to $(U_{m.n} * S_{n.n} * V^T_{n.n})$, we obtain: $(U_{m.k} * S_{k.k} * V^T_{k.n})$.

$$M^* = \begin{bmatrix} 0.0 & 0.0 \\ -0.57 & 0.0 \\ 0.0 & 0.0 \\ -0.58 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & -1.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ -0.58 & 0.0 \\ 0.0 & 0.0 \end{bmatrix} \begin{bmatrix} 1.73 & 0.0 \\ 0.0 & 0.71 \end{bmatrix} \begin{bmatrix} -1.0 & 0.0 & 0.0 \\ 0.0 & -0.71 & -0.71 \end{bmatrix}$$

The next step is to calculate a new vector representation for log entry based on the reduced concept-document matrix. Note that the original vector representation of each document shown in Table 5.2 was based on the original regular expressions-document semantic space. The new set of vector representation is given by Equation 3.4. The resulting document-concept matrix is shown as follows:

$$\begin{bmatrix} -1.0 & 0.0 \\ 0.0 & -0.71 \\ 0.0 & -0.71 \end{bmatrix}$$

We calculate the transposed query vector representation for the query Q as follows:

$$\begin{bmatrix} 0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 \end{bmatrix}$$

We calculate the query representation with respect to the reduced concept matrix based on the Equation 3.5:

$$\begin{bmatrix} -1.0 & 0.0 \end{bmatrix}$$

We calculate the similarity matrix based on Equation 3.6:

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 \end{bmatrix}$$

The similarity matrix correctly indicates that the query is very close to the first log entry in Table 5.1.

5.6 SOA/BPM Experimentation Environment

The test environment that we use to illustrate the proposed framework contains a set of off-the-shelf applications and emulates an enterprise environment. The experimentation environment includes a business process layer and a service oriented infrastructure, and is built using commercial off-the-shelf software such as IBM WebSphere Business Process, IBM WebSphere Message Broker and Microsoft SQL Server 2008 database management system.

Table 5.3: SQL and LSI False Positive and Negative Comparison Table

	Total Rows		False Pos.		False Neg.	
SQL Where Condition	SQL	LSI	SQL	LSI	SQL	LSI
description like 'Message Broker started%'	0	199	0	198	1	0
source_address = '%MB7Broker%' or description like '%start%'	193	200	192	199	0	0
source_address = '%database%' and description like '%start%'	6	22	5	21	0	0

5.6.1 LSI Filtering in Action

Before using the framework, enterprise systems are modeled using annotated goal models and then executed with event logging enabled. The framework collects diverse logs, generates queries that would filter and integrate the log data, which in turn provide useful information for RCA. To evaluate the precision and recall of filtering log data using the proposed framework, we use the test environment described in the previous section to run a set of experiments and we measure the false positive and false negative results with respect to finding log data relevant to a given query.

Our test scenario is based on a custom built financial business process *(Apply_For_Loan)* deployed on the IBM process server. The test scenario involves an online user applying for a loan and having their loan evaluated and finally a loan accept/reject decision is made based on the information supplied by the user. We apply and compare the two information retrieval techniques: SQL and LSI to the log data incoming from five different monitored applications. The log database table contained 501 log entries transformed from their original format and stored in the unified format. The filtering results are shown in Table 5.3. Queries used in this experiment are of the form *SELECT * FROM [EVENTSDB].[dbo].[LogData] where description like '%Message Broker started%'*. In Table 5.3, we only show WHERE condition of each annotation. After performing LSI, the extracted vocabulary contained 570 keywords. The rank reduction was done using a k value of 50 corresponding to a minimum of 3 for singular values. Documents with a threshold value of $\alpha > 0.5$ were considered as relevant to the evaluated queries. We note that for higher values of k, more concepts are generated, resulting in less false positive but potentially more false negatives in the filtered log data.

Note that the two log filtering methods (SQL and LSI) are presented as alternative approaches. The first approach could be characterized with low false positives and high false negatives. In fact, when we experimentally filtered log data by directly applying the generated SQL queries, we noticed that unless the SQL queries are very relaxed and well formulated, the filtered log data returned an empty set. Alternatively, LSI based log filtering is characterized by higher false positives and lower false negatives. Theoretically, it is possible to use the two approaches in a complementary fashion (after relaxing the SQL queries in the first approach) by applying them sequentially to get better results and lower false positives.

Table 5.4: Traditional LSI Precision and Recall Table

	Total(1)	Relevant(2)	(1) ∩ (2)	Precision	Recall
$L1$	26	1	1	3.85%	100%
$L2$	8	10	5	62.5%	50%
$L3$	14	1	0	0%	0%
$L4$	25	10	0	0%	0%
$L5$	30	1	1	3.33%	100%
$L6$	8	6	3	37.5%	50%
$L7$	5	1	0	0%	0%
$L8$	2	6	0	0%	0%
$L9$	2	3	0	0%	0%

5.6.2 RegEx LSI Filtering in Action

To evaluate the RegEx-based filtering, we run two sets of experiments. The objective of the first set of experiments is to measure the precision and recall for log data retrieval. The objective of the second set of experiments is to study the performance of the implemented search/indexing techniques and the impact of the size of the corpus of documents on the indexing/retrieval operations. The experiments were evaluated by applying the *database scan* attack scenario shown in Figure 3.2. This attack aims at querying the contents of the credit rating database from an unauthorized external application.

5.6.3 Precision and Recall

In order to run the first set of experiments, the credit rating service was invoked directly using the soapUI toolkit. Log data generated in a 10 minute interval before and after the attacks is collected from all the systems in the test environment (i.e., process server, broker, credit service, MQ, SQL database server and soapUI). These log entries are converted and stored in a centralized database table based on a unified schema. As part of the evaluation,

Table 5.5: Regex LSI Precision and Recall Table

	Total(1)	Relevant(2)	$(1) \cap (2)$	Precision	Recall
$L1$	0	3	0	0%	0%
$L2$	5	10	5	100%	50%
$L3$	32	1	0	0%	0%
$L4$	19	10	10	52.63%	100%
$L5$	19	3	3	15.79%	100%
$L6$	5	6	3	60%	50%
$L7$	32	1	0	0%	0%
$L8$	3	6	3	100%	50%
$L9$	66	2	2	3.03%	100%

both traditional LSI and regular expression based LSI were applied to extract log data from the database table using the logical expression that are annotated to the goal model nodes. Since the *Apply_For_Loan* business process was not invoked, the first task $a1$ of the example anti-goal tree (Figure 3.2) does not result in any log data generated. The remaining anti-goal model tasks results in at least one event when executed.

The log database table contained 1690 log entries generated from all systems in our test environment. After applying the stemming operation and using a stop-list of common words, the resulting vocabulary included 678 keywords. Using regex LSI with expressions based on Equation (5.1), the corresponding number of generated regular expressions is 877. For LSI, the chosen number of k concepts is 50 while for regex LSI, it is 5. We have listed results for collecting log data based on 9 different expressions using LSI and regex LSI in tables 5.4 and 5.5 (respectively). The results of this experiment indicate an improvement in both precision and recall when indexing/retrieving log data using regex LSI compared to traditional LSI. In particular, we note the following:

- When the same regular expression occurs concurrently in a query and a log entry, then the terms composing this expression have a higher chance of carrying the same meaning. In fact, using regular expressions assures that, whenever there is a match, the shared keywords appear in the same order in both documents, thus handling the BOW assumption of LSI. In addition, the fact that the same set of keywords co-occur in both documents indicates that they carry the same meaning, thus handling the polysemy limitation in LSI.

- The precision and recall varies from one query to another which clearly indicates the impact of the quality of the formulation of the logical expressions on the filtering quality. Experimental results indicate higher precision values for same recall levels in the regular expressions based approach compared to the traditional LSI approach.

- The weighting function to populate the semantic space in LSI is based on the count of occurrences of terms in each document. On the other hand, the weighting function for RegEx LSI is a count of the number of matches of regular expression in each document which is usually lower than the count of each term in a document. As a result, the semantic space in RegEx LSI is sparser than in the standard LSI case; hence the corresponding number of singular values is lower resulting in a lower ranked matrix than in the case of LSI (after applying the rank lowering and SVD algorithms).

5.6.4 Performance

The second experiment aimed to measure the time required (in milliseconds) to perform the most computationally expensive step in the monitoring framework for various log data corpus sizes which is the log filtering step.

The experimental results in Tables 5.4 and 5.5 indicate that for larger sizes of the log data corpus, more time is needed to initialize the semantic space and apply rank reduction on it. After the preliminary work of preparing the reduced semantic space is completed, the remaining query evaluation takes minimal computational time. In general, LSI operations took more time than the proposed regular expression LSI, except for the semantic space initialization phase where complex expressions are being matched in regular expression LSI, compared to terms matching in the case of LSI. Note that k-Concepts of 3 and 50 were used in the rank lowering phases in regular expression LSI and standard LSI (respectively).

5.7 Summary

This chapter presents a framework for log filtering in distributed systems. We inspect log data generated by six off-the-shelf software applications. We define a unified log data format that fields from both standards (WEF and the Windows Event Viewer) deemed necessary. In terms of log data reduction, we enhance the basic SQL queries recall by using the LSI which is an approach commonly used in web search algorithms. This enhancement leads to less false positives and easier formulation for queries. Our main motivation to use goal models is to make the log data extracted by our framework readily available to be used by RCA tools for proving/disproving their diagnosis. Furthermore, we propose and compare a log filtering approach based on a hybrid approach of LSI and pattern expressions.

The motivation behind this work is to improve the precision/recall of the traditional LSI approach when filtering the logs.

Chapter 6

Root Cause Analysis for Failures Caused by Internal Faults

In this chapter, we describe a framework for root cause analysis to detect failures caused by internal system faults. This RCA approach is based first, on the modeling of system requirements in the form of goal trees and second, on filtering log data and extracting an audit trail and third, on a probabilistic reasoning methodology that is based on Markov Logic Networks to guide the root cause identification process. In this respect, once the failure of a functional or non-functional requirement is observed, the corresponding goal tree is analyzed and different combinations of problems that may lead to the observed failure are considered and evaluated utilizing a probabilistic reasoning process. We accomplish this by proposing a run-time architecture that first, models system requirements as goal trees that generate first order logic formulas; second, stores log data in a centralized database using a unified format and; third, uses pattern expressions annotating goal trees to extract log data that is pertinent to the observation and; fourth, applies probabilistic reasoning utilizing Markov Logic Networks in order to generate a ranked set of diagnoses for a given observed failure. The block diagram of the diagnostic process is illustrated in Figure 6.1.

This chapter is organized as follows. Section 6.1 presents the motivation behind the work in this chapter. Section 6.2 discusses issues related to building the knowledge base from goal models. Section 6.3 describes the architecture and processes in the proposed framework. Section 6.4 describes an alternative architecture for this RCA framework but using a deterministic approach based on SAT solvers. A case study, a functional and a performance evaluation of the proposed framework are presented in sections 6.5, 6.5.2 and 6.6 respectively. The case study includes a comparison of results using the deterministic and probabilistic approaches. The conclusions are in section 6.7.

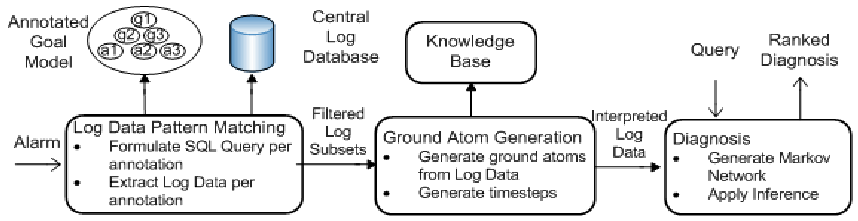


Figure 6.1: Diagnostic Framework Block Diagram.

6.1 Motivating Scenario

In this chapter we propose a probabilistic approach for a software root cause analysis framework. Similarly to the approach in Chapter 7, the proposed probabilistic approach is based on modeling the complex systems' interdependencies using requirement goal models. Goal models, along with log data generated by the monitored systems are used to construct a knowledge base that, in turn, using Markov logic Networks, help infer the root cause(s) for the observed failures. This approach improves on the deterministic SAT-solver based approach in Chapter 7 by providing a better handling of the uncertainty in the observation. In particular, the lack of an observation of an event does not necessarily mean that this event did not occur but that its occurrence is unknown. Moreover, the proposed approach allows for a more refined representation of the monitored software environment by providing the capability to assign different weights to the rules that describe the relationships between the different software components depending on the importance of each rule and as assigned by subject matter experts.

6.2 Knowledge Representation

In this section, we discuss the issues and steps related to modeling a diagnostic knowledge base from goal models.

6.2.1 Goal Model Annotations

We extend the goal models used in the proposed framework by annotating the goal model nodes with additional information on the events pertaining to each of these nodes. In particular, tasks (leaf nodes) are associated with preconditions, occurrence and postconditions, while goals (non-leaf nodes) are associated with preconditions and postconditions only. The annotations are expressed using string pattern expressions of the form, $[not]column\_name[not]like``match\_string''$ where *column_name* represents a field name in the log database and *match_string* can contain the following symbols:

- %: Matches strings of zero or many characters.
- Underscore (_): Matches one character.
- [...]: enclose sets or ranges, such as $[abc]$ or $[a-d]$.

An annotation example is the precondition for goal g_1 (in Figure 3.2) shown below:
Pre(g_1): Description like '%starting%Database%eventsDB%' AND source_component like 'eventsDB'
This annotation example matches event trace generated by the "eventsDB" database system that have a description text containing the keyword *starting*, followed by space, then followed by the keyword *Database*, then space then followed by the keyword *eventsDB*. More annotation examples are shown in Figure 3.2.

6.2.2 Goal Model Predicates

As discussed previously, goal models denote the conditions, tasks and constraints for the system requirements to be achieved. For our work, we use first order logic to represent semantic information on goal models. We represent the monitored systems/services' states and actions as first order logic predicates. A predicate is intensional if its truth value can only be inferred (i.e., cannot be directly observed). A predicate is extensional if its truth value can be directly observed. A predicate is strictly extensional if it can only be observed and not inferred for all its groundings [95]. We use the extensional predicates *ChildAND(parent_node, child_node)*, *ChildOR(parent_node, child_node)* to denote the AND/OR goal decomposition. For instance, *ChildAND(parent, child)* is true when *child* is an AND-child of *parent* (similarly for *ChildOR(parent, child)*). Examples of AND goal decomposition are goals g_1 and g_2 shown in Figure 3.2. An example of OR decomposition is goal g_3. We use the extensional predicates *Pre(node, timestep)*, *Occ(node, timestep)*, and *Post(node, timestep)* to represent preconditions, tasks' occurrences and postconditions (respectively) at a certain timestep. For our work we assume a total ordering of events according to their logical or physical timestamps [27]. Finally, we use the intensional predicates *G_Occ(node, timestep, timestep)* and *Satisfied(node, timestep)* to represent the goals occurrences and the goals/tasks satisfaction. The predicate *Satisfied* is predominantly intensional except for the top goal which satisfaction is observable (i.e., the observed system failure that triggers the RCA process). If the overall service/transaction is successfully executed, then the top goal is considered to be satisfied, otherwise it is denied.

6.2.3 Goal Model Rules

Relationships in the goal model are represented using first order logic expressions. Goals/tasks' satisfaction are expressed using the truth assignment of the *Satisfied(node)* predicate which, in turn, are inferred as follows: A task a with a precondition $\{Pre\}$ and a postcondition $\{Post\}$ is satisfied at time $t+1$ if and only if $\{Pre\}$ is true at time $t-1$ just before the task a occur at time t, and $\{Post\}$ is true at time $t+1$ (please see Equation 7.1).

$$Pre(a, t-1) \wedge Occ(a, t) \wedge Post(a, t+1) \Rightarrow Satisfied(a, t+1) \tag{6.1}$$

Unlike tasks which occur on a specific moment of time, goal occurrences span over an interval $[t_1, t_2]$ that includes the occurrences times of its children goals/tasks. Thus, a goal g with precondition $\{Pre\}$ and postcondition $\{Post\}$ is satisfied at time t_2 if and only if the goal occurrence finishes at time t_2, and $\{Pre\}$ is true when goal occurrence starts at t_1 where $(t_1 < t_2)$ and $\{Post\}$ is true when goal occurrence is completed at t_2 (please see Equation 7.2).

$$Pre(g, t_1) \wedge G\_Occ(g, t_1, t_2) \wedge Post(g, t_2)) \Rightarrow Satisfied(g, t_2) \tag{6.2}$$

The truth values of the predicate *G_Occ(goal)* (used in Equation 7.2) can only be inferred based on the satisfaction of all its children in the case of AND-decomposed goals (Equation 7.3) or at least one of its children in the case of OR-decomposed goals (Equation 7.4).

$$\forall a, Satisfied(a, t_1) \wedge ChildAND(g, a) \wedge (t_2 < t_1 < t_3) \Rightarrow G\_Occ(g, t2, t3) \tag{6.3}$$

$$\exists a, Satisfied(a, t_1) \wedge ChildOR(g, a) \wedge (t_2 < t_1 < t_3) \Rightarrow G\_Occ(g, t_2, t_3) \tag{6.4}$$

Contribution links of the form $node_1 \xrightarrow{++S} node_2$ are represented in Equation 7.5. (Similarly for *++D,--S,--D*).

$$Satisfied(node_1, t_1) \Rightarrow Satisfied(node_2, t_2) \tag{6.5}$$

6.2.4 Storing of Log Data

In a system such as the loan example application system, each component generates log data using its own native schema. We consider mappings from the native schema of each

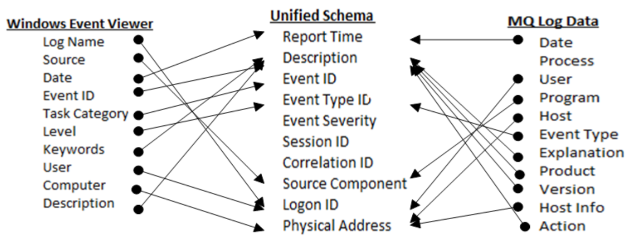


Figure 6.2: Mappings from Windows Event Viewer and MQ Log into Unified Schema.

logger into a common log schema as outlined in Table 6.1. The unified schema we utilize for this work contains ten fields classified into four categories: *general*, *event specific*, *session information* and *environment related*. This proposed schema represents a comprehensive list of data fields that we consider to be useful for diagnosis purposes. In practice, many commercial monitor environments contain only a subset of this schema. The identification of the mappings between the native log schema of each monitor component and the unified schema is outside of the scope of this thesis. Such mappings can be compiled using semi-automated techniques discussed in detail in [5] or compiled manually by subject matter experts. For the purposes of this study, we have implemented the mappings as tables using the Java programming language. Figure 6.2 illustrates mappings between the unified schema and the native schema of two systems used in the test environment.

6.3 Root Cause Analysis Framework using Markov Logic Networks

In this Section, we present the elements of the proposed framework and we illustrate its use through the running example. The framework consists of two main components: observation generation and diagnosis. This framework takes as input first, the set of annotated goal models that represent the monitored systems and second, the log data stored in unified format in a centralized database. The behavior of the framework is described in the sequence diagram in Figure 6.3 and in the following sections.

6.3.1 Observation Generation

The first component uses pattern expressions annotating the goal models and applies them to the log data in order to extract evidence for the occurrence of events associated with the goal model nodes. We describe this component in the following sections:

Goal Model Compilation

The proposed framework is built on the premise that the monitored system′s requirements goal model is available by system analysts or can be reverse engineered from source code using techniques discussed in [108]. Tasks in a goal model represent simple components in the source code. These are treated as black boxes for the purposes of monitoring and diagnosis which enable us to model a software system at different levels of abstraction.

Ground Atoms Generation

Once logged data are stored in a unified format, the pattern expressions that annotate the goal model nodes (see Figure 3.2) can be used to generate SQL queries that are applied to collect a subset of the logged data pertaining to the analysis. An example of log data pattern matching the expression (example in section 6.2.1) of the precondition of goal g_1 in Figure 3.2 is shown below:

Report_Time	**Description**	**Physical_Address**
2010-02-05 17:46:44.24	*Starting database eventsDB...*	*DATABASE*

The truth assignment for the extensional predicates is done based on the pattern matched log data. We show below a subset of the ground atoms for the goal model in Figure 3.2,

pre(g_1,1), pre(a_1,1), occ(a_1,2), post(a_1,3), pre(g_2,3), !pre(a_3,3), ?occ(a_3,4), . . . , ?post(g_1,15), !satisfied(g_1,15)

The set of literals above represents the observation of one failed loan application session. For some of the events corresponding to goals/tasks execution, there may contain no evidence of their occurrence which can be interpreted as either they did not occur or they were missed from the observation set. We use this uncertainty by preceding the corresponding ground atoms with interrogation mark (?). In cases where there is evidence that an event did not occur, the corresponding ground atom is preceded with an exclamation mark (!). For example, in Figure 3.2 the observation of the system failure is represented by *!Satisfied(g_1,15)* which indicates top goal g_1 denial at timestep 15. Note in the above

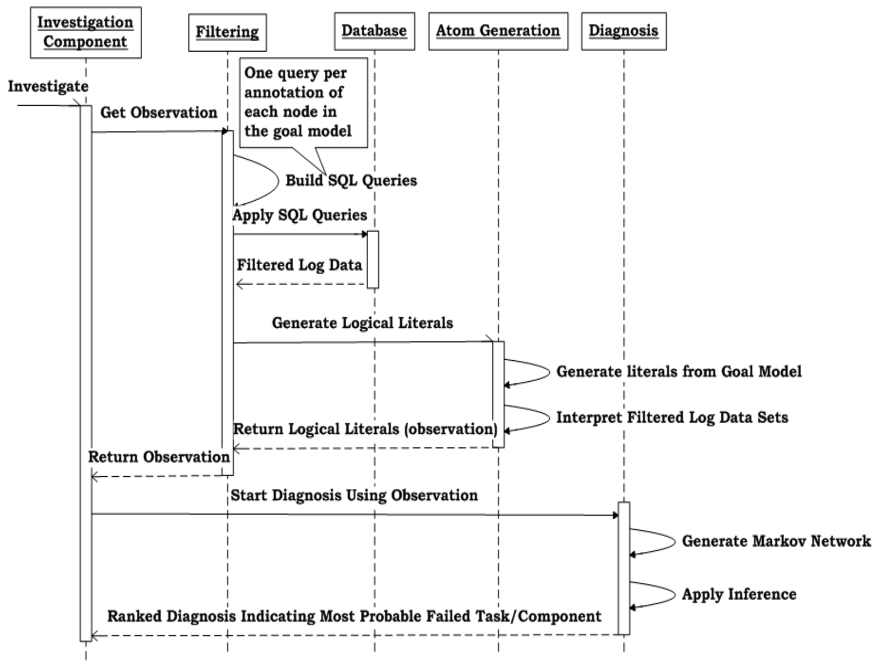

Investigation Component
Filtering
Database
Atom Generation
Diagnosis
Investigate
Get Observation
One query per annotation of each node in the goal model
Build SQL Queries
Apply SQL Queries
Filtered Log Data
Generate Logical Literals
Generate literals from Goal Model
Interpret Filtered Log Data Sets
Return Logical Literals (observation)
Return Observation
Start Diagnosis Using Observation
Generate Markov Network
Apply Inference
Ranked Diagnosis Indicating Most Probable Failed Task/Component

Figure 6.3: Sequence Diagram for the inner workings of the diagnosis Framework.

example, the precondition for task a_3 was denied and the occurrence of a_3 was not observed leading to the denial of task a_3, which led to goal g_2 not to occur and thus be denied. In turn, the denial of goal g_2 supports the observation goal g_1 did not occur and consequently satisfied.

The filtering of the predicate with timesteps of interest is done using Algorithm VI. Algorithm VI consists of two steps: first, a list of literals is generated (see example above) by left-first traversing the goal model tree and generating a list of literals from the nodes annotations (precondition, occurrences and postconditions); second, sequentially go through that list and look for evidence in the log data for the occurrence of each literal (within a certain time interval).

Algorithm VI: Observation Generation

```
Input:
 goal_model: goal model for the monitored system
 log_data: log data stored in central database
Output:
```

literals: set of literals of the form *[?,!] literal(node,timestep)*

```
Procedure generate_literals(log_subset, node, annotation) {
 Set Curr_Node = $g_1$ (Start by the top of the goal tree)
 Generate an entry for precondition of Curr_Node:
  literal = pre(Curr_Node, global_counter);
 While Curr_Node has children,
  find leftmost not visited child and call generate_literals on it
 If all children are visited or Curr_Node has no children,
  If this is a task, generate an entry for occurrence:
   literal = occ(Curr_Node, global_counter++);
  generate an entry for this node's postcondition:
   literal = post(Curr_Node, global_counter++);
  literals[].add(literal);}
 return literals[];}
}
Procedure assign_logical_values(literals[]) {
 for each literal in the set literals[]
  load the corresponding pattern expression;
  filter the log data based on the loaded pattern expression;
  if no log data found matching the pattern expression:
   literal = (?) literal;
  else leave it unchanged;
  if system is reported to have failed:
   add literal satisfied(top, max_timestep) to literals[];
 return literals[];}
}
main(goal_model, matched_subsets[])
 literals[] = generate_literals(goal_model, node, annotation);
 assign_logical_values(literals[])
return literals[];}
```

Algorithm VI is illustrated with the following example: a log entry (*2010-02-05 17:46:-44.24 Starting up database eventsDB ... DATABASE*) that matches the pattern of the precondition of goal g_1 represents an evidence for the occurrence of this event (precondition in this case). Other logical literals that do not have evidence in the log data that they occurrence, a (?) mark is assigned to these literals such as the precondition of a_1.

Table 6.1: Unified Schema for Log Data

	Field Name	**Category**
1	Report_Time	General
2	Description	General
3	Event_ID	Event Specific
4	Event_Type_ID	Event Specific
5	Event_Severity	Event Specific
6	Session_ID	Session Information
7	Correlation_ID	Session Information
8	Source_Component	Environment Related
9	Logon_ID	Environment Related
10	Physical_Address	Environment Related

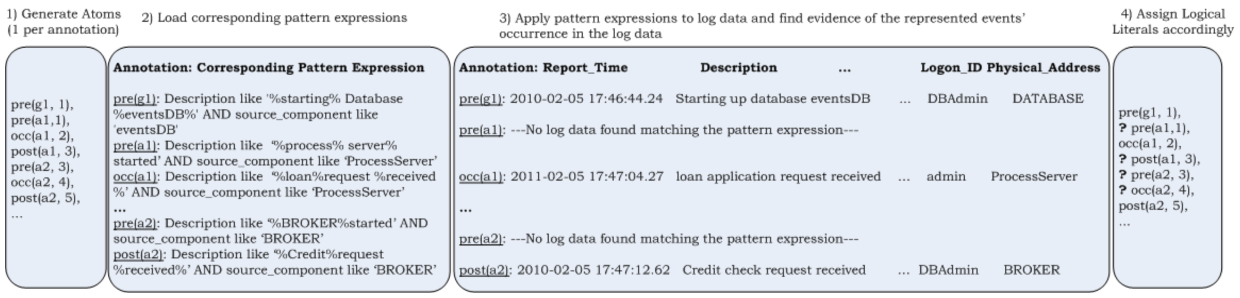
The ground atom generation process is exemplified in Figure 6.4 where the log data is filtered and used to associate the logical assignment into a set of ordered literals that was generated form the goal model.

Algorithm VI performs a depth-first traversal for the goal model and generates a literals for each of the events associated with each node. Thus the worst case performance of this algorithm is in the order of O(n) where n is the number of vertices (nodes of the goal model).

Uncertainty Representation

This framework relies on log data as evidence for the diagnostic process. The process of selecting log data (described in the previous step) can potentially lead to false negatives and false positives which in turn lead to a decreased confidence in the observation. We address uncertainty in observations using a combination of logical and probabilistic models:

- The domain knowledge representing the interdependencies between systems/services is modeled using weighted first order logic statements. The strength of each relationship is represented with a real-valued weight set based on domain knowledge and learning from a training log data set and reflects. The weight of each rule represents our confidence in this rule relative to the other rules in the knowledge base. Consequently, the probability inferred for each atom depends on the weight of the competing rules where this atom occurs. For instance, the probability of the satisfaction of task *a4* in Figure 3.2 (*Satisfied(a4,t)*) is inferred based on the Equation


1) Generate Atoms (1 per annotation)
2) Load corresponding pattern expressions
3) Apply pattern expressions to log data and find evidence of the represented events' occurrence in the log data
4) Assign Logical Literals accordingly

```
pre(g1, 1),
pre(a1,1),
occ(a1, 2),
post(a1, 3),
pre(a2, 3),
occ(a2, 4),
post(a2, 5),
...
```

```
Annotation: Corresponding Pattern Expression

pre(g1): Description like '%starting% Database
%eventsDB%' AND source_component like
'eventsDB'
pre(a1): Description like  '%process% server%
started' AND source_component like 'ProcessServer'
occ(a1): Description like  '%loan%request %received
%' AND source_component like 'ProcessServer'
...
pre(a2): Description like '%BROKER%started' AND
source_component like 'BROKER'
post(a2): Description like '%Credit%request
%received%' AND source_component like 'BROKER'
```

```
Annotation: Report_Time          Description                          ...    Logon_ID Physical_Address

pre(g1): 2010-02-05 17:46:44.24   Starting up database eventsDB        ...   DBAdmin    DATABASE

pre(a1): ---No log data found matching the pattern expression---


occ(a1): 2011-02-05 17:47:04.27   loan application request received    ...   admin      ProcessServer

...

pre(a2): ---No log data found matching the pattern expression---

post(a2): 2010-02-05 17:47:12.62  Credit check request received        ...   DBAdmin    BROKER
```

```
pre(g1, 1),
? pre(a1,1),
occ(a1, 2),
? post(a1, 3),
? pre(a2, 3),
? occ(a2, 4),
post(a2, 5),
...
```

Figure 6.4: Generating Observation from Log Data.

7.8 with weight *w1*,

$$w1 \;\; Pre(a4, t) \wedge Occ(a4, t+1) \wedge Post(a4, t+2) \Rightarrow Satisfied(a4, t+2) \qquad (6.6)$$

On the other hand, the ($--S$) contribution link (defined in section 3.2) with weight *w2* (Equation 7.9) quantifies the impact of the denial of goal *g3* on task *a4*,

$$w2 \;\; !Satisfied(g_3, t_1) \Rightarrow !Satisfied(a_4, t_2) \qquad (6.7)$$

Consequently, the probability assignment given to $Satisfied(a_4, t)$ is determined by the rules containing it as well as the weight of these rules.

- Applying an open world assumption to the observation where a lack of evidence does not absolutely negate an event's occurrence but rather weakens its possibility.

6.3.2 Diagnosis

The second component generates a Markov Network based on the goal model relationships described in section 6.2 and then uses the observation generated to provide an inference on the root causes for the system failure.

Weight Learning

Rules weight learning is done semi-automatically by first using discriminative learning based on a training set [59] and then manually being refined by a system expert. During automated weight learning, each formula is converted to CNF, and a weight is learned for each of its clauses. The weight of a clause is used as the mean of a Gaussian prior for the learned weight. The learned weight can be further modified by the operator to reflect his or her confidence in the rules. For example, in Figure 3.2, a rule indicating that the denial of the top goal g_1 implies that at least one of its AND-decomposed children (a_1, g_2 and a_2) have been denied, should be given higher weight than a rule indicating that a_1 is satisfied based on log data proving that *pre*(a_1), *occ*(a_1) and *post*(a_1) are true. In this case, the operator has without any doubt witnessed the failure of the system, even if the logged data do not directly "support" it and therefore, can adjust the weights manually to reflect this scenario.

Markov Network Construction

A Markov Network is constructed using an exhaustive scheme of rules and predicates as discussed earlier as well as, grounding predicates with all possible values, and connecting them if they coexist in a grounded formula. The choice of possible values can lead to an explosion in the number of ground atoms and network connections if not carefully designed, in particular when modeling time. For this purpose, we represent time using timesteps (integers) that denote the time interval that one session of the service described by the goal model takes to execute.

Inference

Using the constructed Markov Network, we can infer the probability distribution for the ground atoms in the KB given the observations. Of particular interest are the ground atoms for the *Satisfied(node, timestep)* predicate which represents the satisfaction or denial of tasks and goals in the goal model at a certain timestep. The Algorithm VII below is used to produce a diagnosis for failure of a top goal at timestep T. MLN inference generates weights for all the ground atoms of *Satisfied(task,timestep)* for all tasks and at every timestep. Based on the MLN rules listed in section 3.7, the contribution of a child node's satisfaction to its parent goal's occurrence depends on the timestep of when that child node was satisfied. Algorithm VII recursively traverses the goal model starting at the top goal node. Using the MLN set of rules, Algorithm VII identifies the timestep t for each task's *Satisfied(n,t)* ground atom that contributes to its parent goal at a specific timestep (*t'*). The *Satisfied* ground atoms of tasks with the identified timesteps are added to a secondary list and then ordered based on their timesteps. Finally, Algorithm VII inspects the weight of each ground atom in the secondary list (starting from the earliest timestep), and identifies the tasks with grounds atoms that have a weight of less than 0.5 as

Table 6.2: Four scenarios for the Loan Application.

Scenario	**Observed (& Missing) Events(s)**	*Satisfied (a1,3)*	*Satisfied(a3,5)*	*Satisfied(a6,7)*	*Satisfied(a7,9)*	*Satisfied(a4,11)*	*Satisfied(a5,13)*	*Satisfied(a2,15)*
1 - Successful execution	Pre(g_1,1), Pre(a_1,1), Occ(a_1,2), Post(a_1,3), Pre(g_2,3), Pre(a_3,3), Occ(a_3,4), Post(a_3,5), Pre(g_2,5), Pre(a_6,5), Occ(a_6,6), Post(a_6,7), ?Pre(a_7,5), ?Occ(a_7,6), ?Post(a_7,7), Post(g_3,7), Pre(a_4,7), Occ(a_4,8), Post(a_4,9), Pre(a_5,9), Occ(a_5,10), Post(a_5,11), Pre(a_2,11), Occ(a_2,12), Post(a_2,13), Post(g_1,13), Satisfied(g_1,13)	0.99	0.99	0.99	0.49	0.99	0.99	0.99
2 - Failed to update loan database	Pre(g_1,1), Pre(a_1,1), Occ(a_1,2), Post(a_1,3), Pre(g_2,3), Pre(a_3,3), Occ(a_3,4), Post(a_3,5), Pre(g_3,5), Pre(a_6,5), Occ(a_6,6), Post(a_6,7), ?Pre(a_7,5), ?Occ(a_7,6), ?Post(a_7,7), Post(g_3,7), Pre(a_4,7), ?Occ(a_4,8), ?Post(a_4,9), ?Pre(a_5,9), ?Occ(a_5,10), ?Post(a_5,11), ?Pre(a_2,11), ?Occ(a_2,12), ?Post(a_2,13), ?Post(g_1,13), !Satisfied(g_1,13)	0.99	0.99	0.99	0.43	0.45	0.45	0.36

3 - Credit database table not accessible	Pre(g_1,1), Pre(a_1,1), Occ(a_1,2), Post(a_1,3), Pre(g_2,3), Pre(a_3,3), Occ(a_3,4), Post(a_3,5), Pre(g_2,5), ?Pre(a_6,5), ?Occ(a_6,6), ?Post(a_6,7), ?Pre(a_7,5), ?Occ(a_7,6), ?Post(a_7,7), Post(g_3,7), Pre(a_4,7), Occ(a_4,8), Post(a_4,9), Pre(a_5,9), Occ(a_5,10), Post(a_5,11), Pre(a_2,11), Occ(a_2,12), Post(a_2,12), ?Post(g_1,13), !Satisfied(g_1,13)	0.99	0.99	0.33	0.32	0.45	0.43	0.38
4 - Failed to validate loan application WS request	?Pre(g_1,1), Pre(a_1,1), ?Occ(a_1,2), ?Post(a_1,3), Pre(g_2,3), ?Pre(a_3,3), ?Occ(a_3,4), ?Post(a_3,5), Pre(g_3,5), Pre(a_6,5), ?Occ(a_6,6), ?Post(a_6,7), Pre(a_7,5), ?Occ(a_7,6), ?Post(a_7,7), ?Post(g_3,7), Pre(a_4,7), ?Occ(a_4,8), ?Post(a_4,9), ?Pre(a_5,9), ?Occ(a_5,10), ?Post(a_5,11), ?Pre(a_2,11), ?Occ(a_2,12), ?Post(a_2,13), ?Post(g_1,13), !Satisfied(g_1,13)	0.48	0.48	0.35	0.36	0.47	0.48	0.46

the potential root cause for the top goal's failure. The tasks with ground atoms at earlier timesteps are identified as more likely to the source of failure. A set of diagnosis scenarios based on the goal model in Figure 3.2 are shown in Table 6.2. For each scenario, a set of ground atoms representing this particular instance's observation is applied to the set of MLN rules representing the goal model in order to generate the probability distribution for the *Satisfied* predicate, which in turn is used to infer the root cause for the failure.

Algorithm VII: Diagnosis Algorithm

```
Input:
  mln: weighted rules for the goal model
  Satisfied(n,t): ground atoms for predicate Satisfied
  T: timestep where top goal satisfaction is investigated
Output:
  $\Gamma$: ranked list of root causes
Diagnose(mln, Satisfied(n,t), T) {
 initialize $\Theta$ and $\Gamma$ to be empty, and set $t=T$
 add Satisfied(topgoal,T) to $\Phi$
 for each goal $g$ corresponding to an atom in $\Phi$ {
   set t = timestep in the Satisfied(g,t) ground atom
   for each task $a$ child of goal $g$ {
     look up rule $r$ in mln that exhibits the contribution of Satisfied(a,t1) to (GoalOccurrence(g,t))
     identify the timestep t1 using $r$
     add ground atoms Satisfied(a,t1) and its weight to $\Theta$}
   add Satisfied ground atoms of all subgoals of $g$ to $\Phi$}
 next, order the ground atoms in $\Theta$ based on timesteps
 for each ground atom ga in $\Theta$ {
   if weight of Satisfied(ga,t) $\leq 0.5$) {
     if $a$ is an AND child $\Rightarrow$ RETURN ($a$)
     if $a$ is an OR child {
       find siblings of $a$ in goal model
       if no sibling of $a$ is satisfied in $\Theta$ $\Rightarrow$ add $a$ to $\Gamma$
       if $a$ has at least one satisfied sibling $\Rightarrow$ CONTINUE.}}}
```

Algorithm VII iterates through all goals and their children. Thus, the performance of the algorithm is in the order of $O(n^2)$ where n is the number of nodes.

6.4 Root Cause Analysis using SAT Solvers

The RCA framework can be used in a deterministic mode by using a SAT solver based diagnostics component. This is adapted from work presented in [104], where the diagnostic component uses a SAT solver from the Artificial Intelligence (AI) theory of action and diagnosis. Similarly to the probabilistic approach, the framework takes as input the set of annotated goal models and the log data stored and it consists of two components: observation generation and diagnosis. The behavior of the framework is described in the following sections.

6.4.1 Observation Generation

Similarly to section 6.3.1, the observation generation is based on using pattern expressions annotating the goal models and applying them to the log data in order to extract evidence for the occurrence of events associated with the goal model nodes.

The process of ground atoms generation is similar to section 7.2.2, however in the deterministic case, it is uses a closed world assumption. In this case, the lack of evidence for the occurrence of an event is interpreted as evidence that the event did not occur, thus the corresponding ground atom is prefixed with an exclamation mark (!). We show below the closed world based representation of the generated ground atoms for the same example in section 7.2.2

$pre(g_1,1)$, $pre(a_1,1)$, $occ(a_1,2)$, $post(a_1,3)$, $pre(g_2,3)$, $!pre(a_3,3)$, $!occ(a_3,4)$, $\ldots$, $!post(g_1,15)$, $!satisfied(g_1,15)$

Similarly to section 6.3.1, the processing of generating the set of logical trails and associating them timesteps is done using Algorithm VI.

The domain knowledge representing the interdependencies between systems/services is modeled using (non-weighted) first order logic statements. In this context, the deterministic case can be considered as a special case of the probabilistic case where all the statements are given absolutely large weights. On the other hand, the closed world assumption regarding the observation generated based on the log data does not allow for the handling of uncertainty caused by missed log entries and not well formed queries.

6.4.2 Diagnosis using SAT Solvers

The SAT encoder transforms the parsed goal model and log data (offline) into a propositional formula in Conjunctive Normal Form (CNF). The SAT solver uses the CNF formula generated by the encoder and tries to find a set of variables in the propositional formula that makes it evaluate to *TRUE*. If found, this combination of variables represents a list of potential underlying systems that may have triggered the alert of concern. As a result, there could be no diagnosis (system running correctly), one or more diagnoses. The combination of variables in a diagnosis corresponds to goals/tasks that are denied or not. In particular, if a variable is preceded by a *!* in a diagnosis, then the corresponding goal/task is denied. If one task in the generated diagnosis represents a separate system, we can invoke the RCA framework again by loading the corresponding goal model and repeating the whole analysis. Table 6.3 contains the diagnosis for the same scenarios in Table 6.2 but now using the SAT solver based approach.

6.5 Case Studies

In this section, we discuss two case studies to demonstrate the applicability of the proposed framework in detecting the root causes for failures.

6.5.1 Loan Application

The first case study consists of a set of scenarios conducted using the loan application business process. These scenarios are conducted from the perspective of the system administrator where a system failure is reported and a root cause investigation is triggered. The 5 scenarios in this study include one success and four failure scenarios (see Table 6.2).

The normal execution scenario for the system starts upon receiving a loan application in the form of a Web Service request. The loan applicant's information is extracted and used to build another request that is sent to a credit evaluation Web Service. The credit rating of the applicant is returned as Web Service reply. Based on the credit rating of the loan applicant, a decision is made on whether to accept or reject the loan application. This decision is stored in a table before a Web Service reply is sent back to the front end application. The requirements of the Loan application process are modeled in the goal tree illustrated in Fig. 3.2. For our running example we consider that the operator observes that the top goal *g1* of the goal model has failed. The motivating scenario has been

Table 6.3: Four scenarios for the Loan Application using SAT Solver-based Diagnostics.

Scenario	**Observed (& Missing) Events(s)**	Diagnostics
1 - Successful execution	Pre(g_1,1), Pre(a_1,1), Occ(a_1,2), Post(a_1,3), Pre(g_2,3), Pre(a_3,3), Occ(a_3,4), Post(a_3,5), Pre(g_2,5), Pre(a_6,5), Occ(a_6,6), Post(a_6,7), !Pre(a_7,5), !Occ(a_7,6), !Post(a_7,7), Post(g_3,7), Pre(a_4,7), Occ(a_4,8), Post(a_4,9), Pre(a_5,9), Occ(a_5,10), Post(a_5,11), Pre(a_2,11), Occ(a_2,12), Post(a_2,13), Post(g_1,13)	No denied tasks
2 - Failed to update loan database	Pre(g_1,1), Pre(a_1,1), Occ(a_1,2), Post(a_1,3), Pre(g_2,3), Pre(a_3,3), Occ(a_3,4), Post(a_3,5), Pre(g_3,5), Pre(a_6,5), Occ(a_6,6), Post(a_6,7), !Pre(a_7,5), !Occ(a_7,6), !Post(a_7,7), Post(g_3,7), Pre(a_4,7), !Occ(a_4,8), !Post(a_4,9), !Pre(a_5,9), !Occ(a_5,10), !Post(a_5,11), !Pre(a_2,11), !Occ(a_2,12), !Post(a_2,13), !Post(g_1,13), !Satisfied(g_1,13)	a7, a4, a5, a2 are denied
3 - Credit database table not accessible	Pre(g_1,1), Pre(a_1,1), Occ(a_1,2), Post(a_1,3), Pre(g_2,3), Pre(a_3,3), Occ(a_3,4), Post(a_3,5), Pre(g_2,5), !Pre(a_6,5), !Occ(a_6,6), !Post(a_6,7), !Pre(a_7,5), !Occ(a_7,6), !Post(a_7,7), Post(g_3,7), Pre(a_4,7), Occ(a_4,8), Post(a_4,9), Pre(a_5,9), Occ(a_5,10), Post(a_5,11), Pre(a_2,11), Occ(a_2,12), Post(a_2,12), !Post(g_1,13)	a6, a7, a2 are denied
4 - Failed to validate loan application WS request	!Pre(g_1,1), Pre(a_1,1), !Occ(a_1,2), !Post(a_1,3), Pre(g_2,3), !Pre(a_3,3), !Occ(a_3,4), !Post(a_3,5), Pre(g_3,5), Pre(a_6,5), !Occ(a_6,6), !Post(a_6,7), Pre(a_7,5), !Occ(a_7,6), !Post(a_7,7), !Post(g_3,7), Pre(a_4,7), !Occ(a_4,8), !Post(a_4,9), !Pre(a_5,9), !Occ(a_5,10), !Post(a_5,11), !Pre(a_2,11), !Occ(a_2,12), !Post(a_2,13), !Post(g_1,13)	All tasks are denied

implemented as a proof of concept and includes 6 systems/services: a front end application (soapUI), a process server (IBM Process Server 6.1), a loan application business process, a message broker (IBM WebSphere Message Broker v7.0), a credit check Web Service and an SQL server (Microsoft SQL Server 2008). The framework components were implemented using Java 1.5, while the diagnosis/inference was performed using the open source Alchemy system developed at the University of Washington [59]. We used Microsoft SQL Server 2008 to host the log database.

Scenario 1 represents a successful execution of the loan application process. Please note, that the denial of task a_7 does not represent a failure in the process execution since goal g_3 is exclusive OR-decomposed into a_6 *(extract credit history for existing clients)* and a_7 *(calculate credit history for new clients)*, and the successful execution of either a_6 or a_7 is enough for g_3's successful occurrence (see Figure 3.2). The probability values (weights) of the ground atoms range 0^+ (highly denied) to 0.99 (highly satisfied). During each loan evaluation and before the reply is sent back to the requesting application, a copy of the decision is stored in a local table (a_4 *(Update loan table)*). Scenario 2 represents a failure to update the loan table leading to failure of top goal g_1. Using Algorithm VII, we identify a_4 as the root cause for failure. Note that although a_7 was denied ahead of task a_4, it is not the root cause since it is an OR child of g_3, and its sibling task a_6 was satisfied. Similarly for scenario 3, we identify task a_6 as the root cause for failure. Scenarios 4 represent the failure to validate WS requests for the loan application. Almost no events are observed in this scenario. Using Algorithm VII, we identify task a_1 *(Receive loan application WS request)* as the root cause for the observed failure.

Table 6.4: Functional Evaluation using ATM Simulation System.

# Injected Failure Scenario	**Denied Tasks**	**Ranked Causes (Top 3)**
1 - Turn on ATM	a_1	a_1
2 - Available cash	a_2,a_3,a_4,...	a_2,a_3,a_4
3 - Open connection to the bank	a_4,a_5,a_6,...	a_4,a_5,a_6
4 - Card inserted	a_5,a_6,a_7,...	a_5,a_6,a_7
5 - Enter pin number	a_7,a_9,a_{10},a_{11},a_{12},..	a_9,a_{10},a_{11}
6 - Transaction type	a_7,a_{10},a_{12},a_{13},a_{14},...	a_{12},a_{13},a_{14}
7 - Update Balance	a_7,a_{10},a_{18},a_{19},a_{20},...	a_{18},a_{19},a_{20}
8 - Print	a_7,a_{10},a_{21},a_{22},a_{23},...	a_{21},a_{22},a_{23}
9 - Eject card	a_7,a_{10},a_{22},a_{23},a_{24},...	a_{23},a_{24},a_{25}
10 - Close connection	a_7,a_{10},a_{22},a_{24},a_{25},...	a_{24},a_{25}

6.5.2 ATM Simulation

The second case study is the ATM simulation. This case study is an illustration of an OO design presented in [9]. The application is a simulation of an ATM machine where customers can perform withdraw, deposit, transfer and balance inquiry transactions. The Java original source code contains 36 Classes with 5000 LOC which is reverse engineered into a requirements goal model with 37 goals and 51 tasks. A partial goal graph with 19 goals and 23 tasks is show in Figure 6.5. The framework components were implemented using Java 1.5, while the diagnosis/inference was performed using the open source Alchemy system developed at the University of Washington [59]. We used Microsoft SQL Server 2008 to host the log database.

To evaluate the accuracy of the diagnosis generated by our framework, we ran 10 scenarios where we injected a runtime error in each scenario during the execution of tasks a_1, a_2, a_4, a_5, a_9, a_{12}, a_{18}, a_{21}, a_{23} and a_{25} respectively (Table 6.4). The original system does not generate audit log data. Thus we used Apache log4j to instrument this system. In almost all scenarios, the injected error led to the failure of all subsequent tasks. In scenarios 1 through 5, the first failed task was ranked the highest as the root cause for the failure. Due to the presence of OR goals, some tasks have failed (tasks a_7,a_{10},a_{22} or a_7) in all scenarios but were rightfully not diagnosed as the root cause for the failures by our framework. Diagnosis results in each of the scenarios ranked the failed task as the first root cause reflecting a high accuracy of the diagnostic process.

6.6 Performance Evaluation

The evaluation of this framework was conducted using Ubuntu Linux running on Intel Pentium 2 Duo 2.2 GHz machine. We used a set of extended goal models representing the loan application goal model to evaluate the performance of the framework when larger goal models are used. The 4 extended goal models contained 10, 50, 80 and 100 nodes respectively. The matching and ground atom generation algorithms performance depends linearly on the size of the corresponding goal model and log data and thus lead us to believe that the process can easily scale for larger and more complex goal models. In particular, we are interested in measuring the impact of larger goal models on the learning and inference aspects of the framework. Figure 6.7 illustrates that the number of ground atoms/clauses, which directly impacts the size of the resulting Markov model, is linearly proportional to the goal model size. Figure 6.6 shows that the running time for learning the weights of the rules in the MLN of a goal model increases linearly with the size of the goal model. The

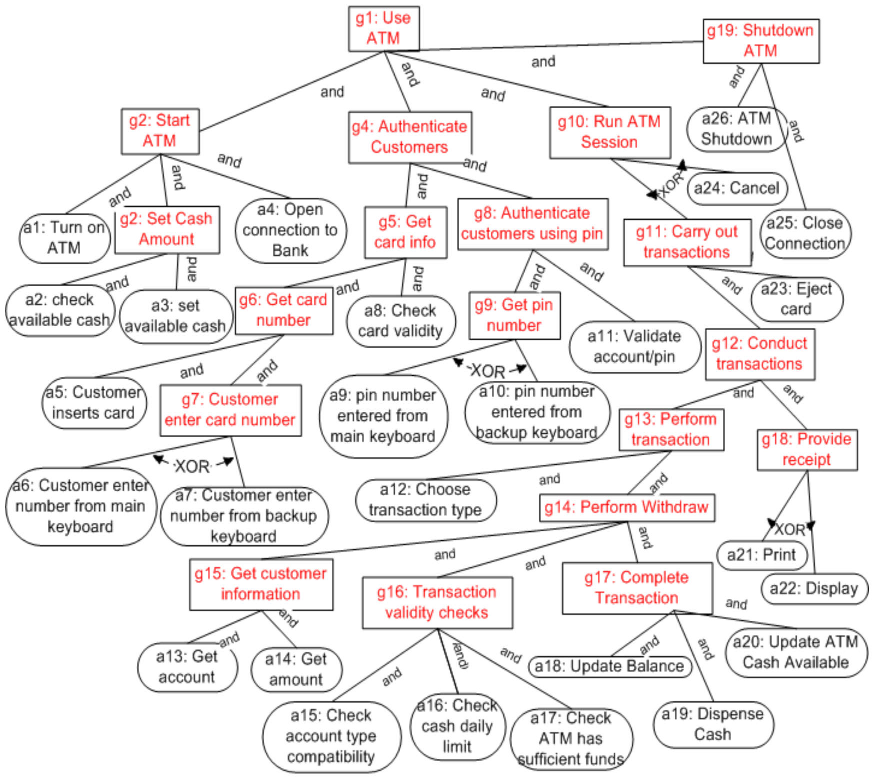

g1: Use ATM
g19: Shutdown ATM
g2: Start ATM
g4: Authenticate Customers
g10: Run ATM Session
a26: ATM Shutdown
a24: Cancel
a25: Close Connection
a1: Turn on ATM
g2: Set Cash Amount
a4: Open connection to Bank
g5: Get card info
g8: Authenticate customers using pin
g11: Carry out transactions
a2: check available cash
a3: set available cash
g6: Get card number
a8: Check card validity
g9: Get pin number
a11: Validate account/pin
a23: Eject card
g12: Conduct transactions
a5: Customer inserts card
g7: Customer enter card number
a9: pin number entered from main keyboard
a10: pin number entered from backup keyboard
g13: Perform transaction
g18: Provide receipt
a6: Customer enter number from main keyboard
a7: Customer enter number from backup keyboard
a12: Choose transaction type
g14: Perform Withdraw
a21: Print
a22: Display
g15: Get customer information
g16: Transaction validity checks
g17: Complete Transaction
a13: Get account
a14: Get amount
a18: Update Balance
a20: Update ATM Cash Available
a15: Check account type compatibility
a16: Check cash daily limit
a17: Check ATM has sufficient funds
a19: Dispense Cash
and
XOR

Figure 6.5: Goal Model for the ATM Machine.

learning time ranged from slightly over 10 minutes for a goal model of 10 nodes, to over 2 hours for a model with 100 nodes. The inference time ranged from 5 seconds for a goal model of 10 nodes, up to 53 seconds for a model of 100 nodes. As a result, our approach can be applied to industrial software applications with small to medium-sized requirement models.

6.7 Summary

This chapter presents a framework that assists operators perform Root Cause Analysis in software systems. The framework takes a goal driven approach whereby software system requirements are modeled as goal trees. Once a failure is observed the corresponding goal tree is analyzed. The analysis takes the form of first, selecting the events to be considered based on a pattern matching process, second on a rule and predicate generation process

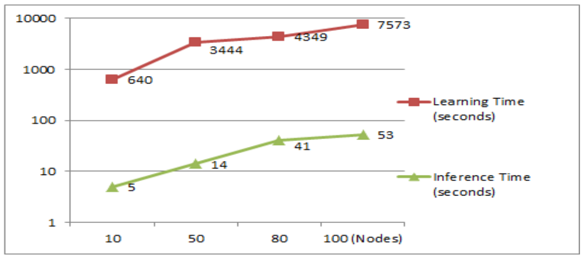


Figure 6.6: Impact of goal model size on learning/inference time.

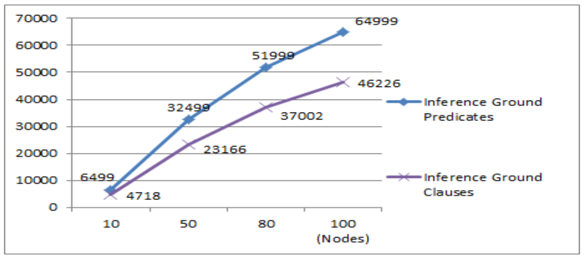


Figure 6.7: Impact of goal model size on ground atoms/clauses for inference.

where goal models are denoted as Horn Clauses and third, a probabilistic reasoning process that is used to confirm or deny the goal model nodes. The most probable combinations of goal model nodes that can explain the failure of the top goal (i.e., the observed failure) is considered the most probable root cause. Goal models for large systems can be organized in a hierarchical fashion allowing for higher tractability and modularity in the diagnostic process. Initial results indicate that the approach is tractable and allows for multiple diagnoses to be achieved and ranked based on their probability of occurrence. In the following chapter, we propose an extension to this RCA framework by handling failures caused as a result of malicious behavior.

Chapter 7

Root Cause Analysis for Failures Caused by Malicious Behavior

Root cause determination for software failures that occurred due to intentional or unintentional third party activities is a difficult and challenging task. The difficulty stems from the lack of models to represent the erroneous system behavior as well as the different malicious behavior, in addition to dealing with uncertainty and missing information regarding the system runtime behavior.

In this chapter, we extend the RCA framework described in Chapter 8 and propose a technique for identifying the root causes of system failures stemming from external interventions. The new approach consists of first, modeling the conditions by which a system delivers its functionality utilizing goal models, second on modeling the conditions by which system functionality can be compromised utilizing anti-goal models, third representing logged data as well as, goal and anti-goal models as rules and facts in a knowledge base and fourth, utilizing a probabilistic reasoning technique that is based on the use of Markov Logic Networks. The technique is evaluated in a medium size COTS based system and the DARPA 2000 Intrusion Detection data set.

The chapter is organized as follows. Section 7.1 covers the motivations behind the work in this chapter. Section 7.2 presents the overall architecture and processes of the proposed framework. Section 7.3 contains the functional and scalability evaluations for the framework. Section 7.4 presents a summary of the chapter.

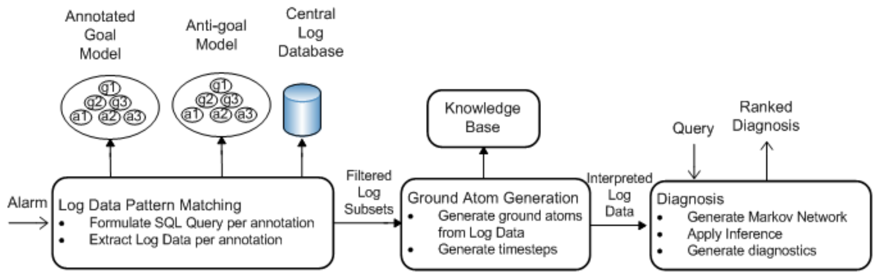


Figure 7.1: Logical Architecture of the Malicious Behavior Detection Framework.

7.1 Motivation

Monitoring and diagnosing (M&D) software based on requirement models is a research area that has recently received a lot of attention in field of Requirement Engineering. This class of frameworks is aimed at using goal models to diagnose failures in software at different levels of granularity. In this context, we classify failures caused by goals that are not satisfied due to false preconditions and/or postconditions are class I failures. Another class of goal failures that we denote class II is caused by malicious attacks. In class II failures, all system components are functioning as designed but an external agent is preventing one or more of the system's tasks from successful execution. All other goal failures including failures caused by inconsistencies between the requirement model and the runtime model are considered to be class III failures.

The framework described in Chapter 8 belongs to the class of M&D that handles class I failures. In this chapter, we describe a RCA framework that can identify class I and II failures. Thus, once the component at the root cause of the system's failure is determined, the proposed framework looks for evidence of any malicious behavior against this component. If any malicious behavior is found against the failed component, this will provide more confidence in the original diagnosis. Furthermore, the failure is classified as class II failure.

7.2 Framework Overview

The framework proposed in this chapter aims at detecting malicious behavior against software systems. As shown in the block diagram in Figure 7.1, this framework consists of three main processes: first, the preprocessing of log data which consists of storing the log data based on a unified schema as well as log filtering. Second, representation/compilation of a set of goal and anti-goal models that are used to generate a diagnostics knowledge base. Finally, using the knowledge base to generate a Markov network that in turn is used for diagnostics.

7.2.1 Log Data Representation and Storage

In a large system which is composed of many subsystems and where each subsystem is monitored by a different logging mechanism, it is possible that the logged data are represented in different formats and schemas. To address the problem of different log data formats and schemas, we consider mappings from the native schema of each logger into a common log schema. The unified schema we utilize for this work contains 10 fields (see Table 7.1) classified into four categories: general, event specific, session information and environment related. This proposed schema represents a comprehensive list of data fields that we consider to be useful for diagnosis purposes.

Schema Mappings

Even though the identification of the mappings between the native log schema of each monitor component and the unified schema is outside of the scope of this thesis, such mappings can be compiled using semi-automated techniques discussed in detail in [5] or compiled manually by domain experts. For the purposes of this study, we have implemented the mappings manually, as tables. Figure 7.2 shows the mappings between the unified schema and the schema of two systems from the test environment.

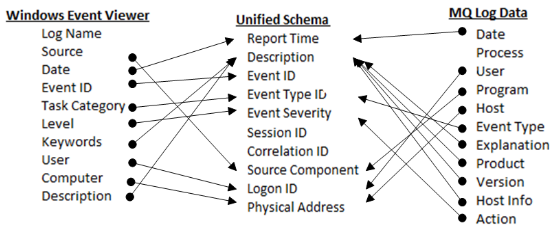


Figure 7.2: Mappings from Windows Event Viewer and IBM's MQ Log into Unified Schema.

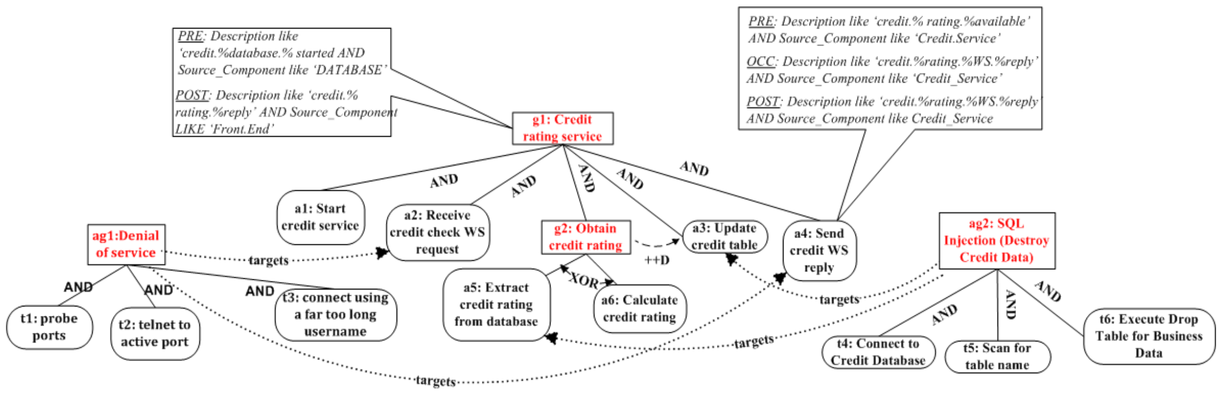

PRE: Description like 'credit.%database.% started AND Source_Component like 'DATABASE'
POST: Description like 'credit.% rating.%reply' AND Source_Component LIKE 'Front.End'
PRE: Description like 'credit.% rating.%available' AND Source_Component like 'Credit.Service'
OCC: Description like 'credit.%rating.%WS.%reply' AND Source_Component like 'Credit_Service'
POST: Description like 'credit.%rating.%WS.%reply' AND Source_Component like Credit_Service
g1: Credit rating service
AND
a1: Start credit service
a2: Receive credit check WS request
g2: Obtain credit rating
a3: Update credit table
a4: Send credit WS reply
++D
XOR
a5: Extract credit rating from database
a6: Calculate credit rating
ag1:Denial of service
t1: probe ports
t2: telnet to active port
t3: connect using a far too long username
targets
ag2: SQL Injection (Destroy Credit Data)
t4: Connect to Credit Database
t5: Scan for table name
t6: Execute Drop Table for Business Data

Figure 7.3: Goal and Anti-Goal Models for the Credit History Service

Storing Log Data

This framework uses natively generated log data for each component of the distributed system. Log data are received as sequences of events from monitored applications. Log data are transformed and stored in a unified table format based on the schema shown in Table 7.1.

Table 7.1: Unified Schema for Log Data

Field Name	**Category**
Report_Time	General
Description	General
Event_ID	Event Specific
Event_Type_ID	Event Specific
Event_Severity	Event Specific
Session_ID	Session Information
Correlation_ID	Session Information
Source_Component	Environment Related
Logon_ID	Environment Related
Physical_Address	Environment Related

In a typical industrial setup, millions of log data entries can be generated daily, thus careful attention should be done to avoid performance issues. In our evaluation data sets, we segmented log data in multiple tables where each table contains a whole day's data. Furthermore, we indexed the tables storing the log data based on the log entry's timestamp which was stored in POSIX time format (total of seconds since January 1, 1970).

Once logged data are stored in a unified format, we apply the pattern expressions that annotate the goal model nodes (Figure 7.3) as queries to collect a subset of the logged data that are relevant to the specific goal model node. An example of a pattern denoting the *occurrence* of task t_1 in the credit service in Figure 7.3 (see section 3.7 for details on the overall loan application):

Description LIKE '%credit%history%WS% request%received' AND Source_Component LIKE 'Credit Service'.

For this work, these subsets are created by utilizing a combination of SQL and Latent Semantic Indexing on the set of events using the keywords appearing in the query as described in Chapters 4, 5 and 6. In this respect, we identify the collection of events that mostly relate to the keywords in the pattern. This step is performed to limit the size of the data to consider and improve the framework's overall performance.

7.2.2 Building a Diagnostics Knowledge Base

In the following sections, we describe the steps to build the diagnostics knowledge base starting from the monitored system's requirements goal and anti-goal models.

Goal and Anti-Goal Model Annotations

This RCA framework is built on the premise that the monitored system′s requirements goal and anti-goal models are available by system analysts or can be reverse engineered from source code. As described in Chapter 3, a goal model defines a set of desired behaviors, whereas the anti-goal models define a set of undesirable behaviors. Furthermore, we consider that leafs in a goal model relate to operations of simple system components, that can be delivered as black boxes for the purposes of monitoring and diagnosis.

Goal and anti-goal models used in this framework are extended by annotations in the form of pattern expressions such as the ones illustrated in Figure 7.3. These expressions describe the conditions under which goals/anti-goals instances can be fulfilled. In particular, tasks (leaf nodes) are associated with preconditions, occurrence and postconditions, while goals (non-leaf nodes) are associated with preconditions and postconditions only.

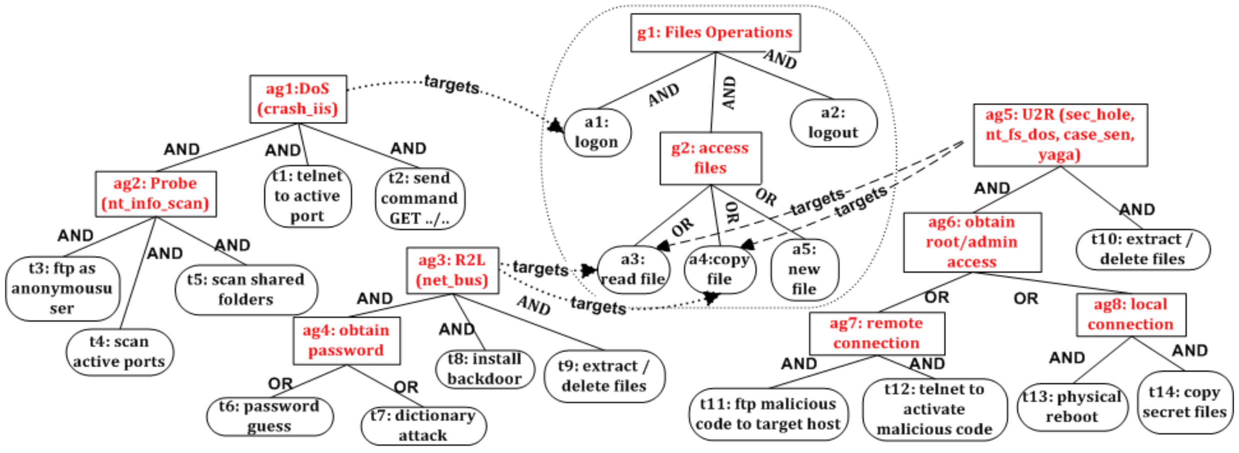


Figure 7.4: Anti-Goal Attacks against (g_1) the Goal Model for Files Services (center).

The annotations are expressed using string regular pattern expressions as described in [41]. An example of such an expression is *Windows.%starting.up* which is used to match strings that contain the word *Windows* followed by an unspecified number of characters, followed by the word *starting*, followed by a single unspecified character, followed by *up*. An annotation example for goal g_1 in Figure 7.4 is shown below:

Description like 'Windows.%starting.up' AND Source_Component like 'hume.eyrie.af.mil'

This example matches event trace generated by the "hume.eyrie.af.mil" host that have a description text containing the sentence *Windows is starting up*, and is considered to be the precondition *Pre(g_1)* for node g_1. The matching process that aims to yield a subset of the logged data according to the annotation *i.e., query* is based either on string matching or on Latent Semantic Indexing (LSI) type of retrieval and is presented in [111] (Chapters 5 and 6). The result of the matching (filtering) process is a list of *Matched_subsets[]* of logged events that relate to the specific annotation specification.

Goal and Anti-Goal Models Predicates

As presented earlier, goal models denote the conditions, tasks and constraints for the system requirements to be achieved, and anti-goal models represent obstacles to the achievement of the system requirements.

We use first order logic to represent semantic information on goal/anti-goal models. A predicate is intensional if its truth value can only be inferred (i.e., cannot be directly observed). A predicate is extensional if its truth value can be directly observed. A predicate is strictly extensional if it can only be observed and not inferred for all its groundings [95].

We use the extensional predicates *ChildAND (parent_node, child_node)*, *ChildOR (parent_node, child_node)* to denote the AND/OR goal decomposition. Goals g_1 and g_2 shown in Figure 7.4 are examples of AND decompositions. An example of OR decomposition is the goal g_2. Similarly for anti-goals, ag_1 in Figure 7.4 represents an example of AND decomposed anti-goal.

We use the extensional predicates *Pre(node, timestep)*, *Occ (node, timestep)*, and *Post (node, timestep)* to represent preconditions, tasks' occurrences and postconditions (respectively) at a certain timestep. We use the extensional predicate *Targets(anti-goal, node)* to denote the relationship between an anti-goal model and the targeted task or goal.

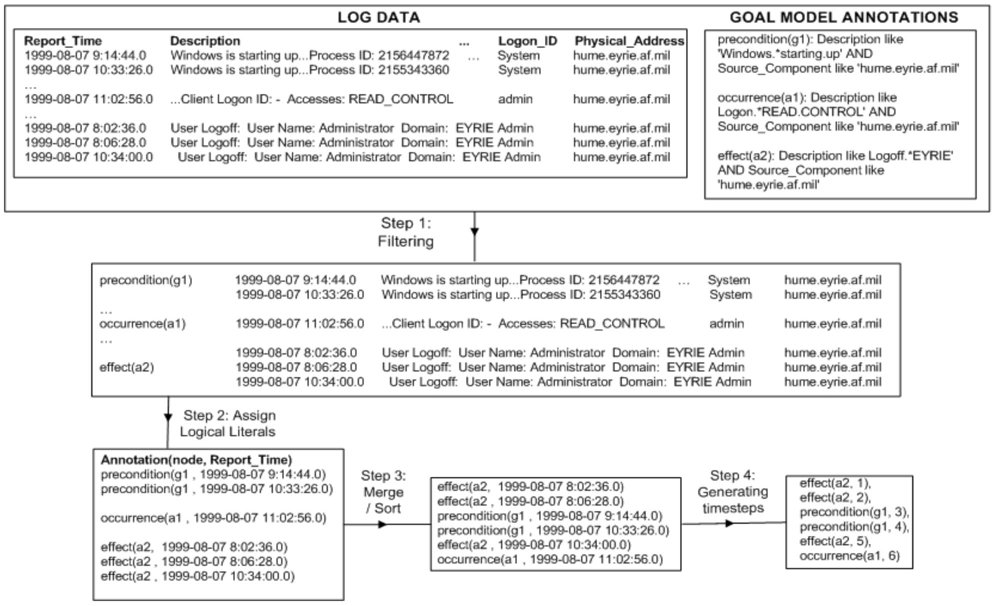
Finally, we use the intensional predicates *Occ(goal, timestep, timestep)* and *Satisfied(node, timestep)* to represent the goals occurrences and the goals/anti-goal/tasks satisfaction. The predicate *Satisfied* is predominantly intensional except for the top goal which satisfaction is observable (i.e., the observed system failure that triggers the root cause analysis process). If the overall service/transaction is successfully executed, then the top goal is considered to be satisfied, otherwise it is denied.

Ground Atoms Generation

In this section, we discuss the process to interpret and transform the (filtered) log subsets into a set of ordered ground atom literals. Figure 7.5 illustrates the basic steps of the transformation process with an example.

There are two inputs to this process: first, the log data stored in a common database as described in Section 7.2.1 and; second, the goal model for the monitored system with *precondition*, *postcondition* and *occurrence* patterns annotating each node of the model. The output is a totally timestep-ordered set of literals (ground atoms) of the form *literal(node,timestep)*.

Step 1: by applying the filtering and matching techniques described in Chapters 4 and 5, we extract from the log database, a collection (*Matched_subsets[]*) of partially ordered log entries sets (the order is based on the timestamp in the *Report_time* field in the unified schema in Table 7.1)). Each subset corresponds to one annotation in the goal model. An example of this process is shown in Figure 7.5 where each goal or task node annotation is associated with a set (can be empty) of log data produced by the matching process. In this respect, each annotation (*precondition*, *postcondition*, *occurrence*), is considered as a



```
Report_Time            Description                                             ...   Logon_ID   Physical_Address
1999-08-07 9:14:44.0   Windows is starting up...Process ID: 2156447872         ...   System     hume.eyrie.af.mil
1999-08-07 10:33:26.0  Windows is starting up...Process ID: 2155343360               System     hume.eyrie.af.mil
...
1999-08-07 11:02:56.0  ...Client Logon ID: -  Accesses: READ_CONTROL                 admin      hume.eyrie.af.mil
...
1999-08-07 8:02:36.0   User Logoff:  User Name: Administrator  Domain:  EYRIE Admin         hume.eyrie.af.mil
1999-08-07 8:06:28.0   User Logoff:  User Name: Administrator  Domain:  EYRIE Admin         hume.eyrie.af.mil
1999-08-07 10:34:00.0    User Logoff:  User Name: Administrator  Domain:  EYRIE Admin       hume.eyrie.af.mil
```

```
precondition(g1): Description like
'Windows.*starting.up' AND
Source_Component like 'hume.eyrie.af.mil'

occurrence(a1): Description like
Logon.*READ.CONTROL' AND
Source_Component like 'hume.eyrie.af.mil'

effect(a2): Description like Logoff.*EYRIE'
AND Source_Component like
'hume.eyrie.af.mil'
```

```
precondition(g1)   1999-08-07 9:14:44.0    Windows is starting up...Process ID: 2156447872   ...   System   hume.eyrie.af.mil
                   1999-08-07 10:33:26.0   Windows is starting up...Process ID: 2155343360         System   hume.eyrie.af.mil
...
occurrence(a1)     1999-08-07 11:02:56.0   ...Client Logon ID: -  Accesses: READ_CONTROL           admin    hume.eyrie.af.mil
...
                   1999-08-07 8:02:36.0    User Logoff:  User Name: Administrator  Domain:  EYRIE Admin     hume.eyrie.af.mil
effect(a2)         1999-08-07 8:06:28.0    User Logoff:  User Name: Administrator  Domain:  EYRIE Admin     hume.eyrie.af.mil
                   1999-08-07 10:34:00.0     User Logoff:  User Name: Administrator  Domain:  EYRIE Admin   hume.eyrie.af.mil
```

```
Annotation(node, Report_Time)
precondition(g1 , 1999-08-07 9:14:44.0)
precondition(g1 , 1999-08-07 10:33:26.0)

occurrence(a1 , 1999-08-07 11:02:56.0)

effect(a2,  1999-08-07 8:02:36.0)
effect(a2 , 1999-08-07 8:06:28.0)
effect(a2 , 1999-08-07 10:34:00.0)
```

```
effect(a2,  1999-08-07 8:02:36.0)
effect(a2 , 1999-08-07 8:06:28.0)
precondition(g1 , 1999-08-07 9:14:44.0)
precondition(g1 , 1999-08-07 10:33:26.0)
effect(a2 , 1999-08-07 10:34:00.0)
occurrence(a1 , 1999-08-07 11:02:56.0)
```

```
effect(a2, 1),
effect(a2, 2),
precondition(g1, 3),
precondition(g1, 4),
effect(a2, 5),
occurrence(a1, 6)
```

Figure 7.5: Transforming Filtered Log Data Subsets into Ground Atoms.

query applied to the log data base that returns a subset of the logged data that may relate to the specific annotation and goal model node. Once all the annotations for a goal model node return a non-empty result set, we consider that the corresponding goal model node has succeeded. An empty log data set means that no evidence can be found in the log data to indicate that this event has occurred during the observation period.

Step 2: for each *log_entry* in each *log_subset* in the set *Matched_subsets[]*, we create a literal in the form *annotation(node, log_entry.timestamp)*, where *annotation* represents *precondition*, *occurrence* or *postcondition*. An example based on the goal model in Figure 7.4 is to transform a log entry (*1999-08-07 1999-08-07 10:33:26.0 DARPA Security Windows is starting up ... Process ID: 2155070784*) that matches the pattern of the *precondition* of goal g_1 into an atom (*pre*(g_1*, 1999-08-07 10:33:26.0*)). As a result of this step, the subsets of log data are converted into subsets of literals (see step 2 in Figure 7.5).

Step 3: the subsets of ground atoms generated earlier are merged in one set representing all the matched log data from all the goal model nodes. A total ordering is accomplished by sorting all ground atoms based on their timestamps (see step 3 in Figure 7.5). To guarantee the uniqueness of each timestamp, each timestamp is appended with the corresponding process/session id.

Step 4: the timestamp in each atom is replaced by a timestep. The timestep is an integer initialized at 1, and incremented for each subsequent atom. Thus *precondition(g_1, 1999-08-07 10:33:26.0)* is transformed into *precondition(g_1,4)*.

The outcome of this process is a set of ordered ground atoms that are used in the subsequent inference phase as one source of system observation.

Goal Model Rules

Relationships in the goal model are also represented using first order logic expressions. Goals/tasks' satisfaction is expressed using the truth assignment of the *Satisfied(node)* predicate which, in turn, is inferred as follows: A task a with a precondition $\{Pre\}$ and a postcondition $\{Post\}$ is satisfied at time $t+1$ if and only if $\{Pre\}$ is true at time $t-1$ just before the task a occur at time t, and $\{Post\}$ is true at time $t+1$ (see Equation 7.1).

$$Pre(a, t-1) \wedge Occ(a, t) \wedge Post(a, t+1) \Rightarrow Satisfied(a, t+1) \tag{7.1}$$

A goal g with precondition $\{Pre\}$ and postcondition $\{Post\}$ is satisfied at time t_2 if and only if the goal occurrence finishes at time t_2, and $\{Pre\}$ is true when goal occurrence starts at t_1 where $(t_1 < t_2)$ and $\{Post\}$ is true when goal occurrence is completed at t_2 (see Equation 7.2).

$$Pre(g, t_1) \wedge G\_Occ(g, t_1, t_2) \wedge Post(g, t_2)) \Rightarrow Satisfied(g, t_2) \tag{7.2}$$

The truth values of the predicate *Occ(goal)* (used in Equation 7.2) can only be inferred based on the satisfaction of all its children in the case of AND-decomposed goals (Equation 7.3) or at least one of its children in the case of OR-decomposed goals (Equation 7.4).

$$\forall a, Satisfied(a, t_1) \wedge ChildAND(g, a) \wedge (t_2 < t_1 < t_3) \Rightarrow Occ(g, t2, t3) \tag{7.3}$$

$$\exists a, Satisfied(a, t_1) \wedge ChildOR(g, a) \wedge (t_2 < t_1 < t_3) \Rightarrow Occ(g, t_2, t_3) \tag{7.4}$$

Contribution links of the form $node_1 \xrightarrow{\text{++S}} node_2$ are represented in Equation 7.5. (Similarly for *++D,--S,--D*).

$$Satisfied(node_1, t_1) \Rightarrow Satisfied(node_2, t_2) \tag{7.5}$$

As shown in Equations 7.3 and ORgoaloccurrence, the contribution of a child to its parent's satisfaction is a function of time. For instance, the occurrence (and consequently the satisfaction) of gaol g_1 at (timestep = 11) in Figure 7.4 is impacted by the satisfaction of task a_1 at (timestep = 3), goal g_2 at (timestep = 5) and task a_2 at (timestep = 7). If for example task a_1 is denied or satisfied at (timestep = 2) then this has no impact on the satisfaction of parent goal g_1 at (timestep = 7).

Anti-Goal Model Rules

Similarly to goal models, anti-goal models also consist of AND-OR decompositions, therefore anti-goal model relationships with their children nodes are represented using the same first order logic expressions as for goal models (Equations 7.3 and 7.4). The satisfaction of tasks (leaf nodes) in anti-goal models follows the same rule as in Equation 7.1. The satisfaction of anti-goals is expressed using the truth assignment of the *Satisfied(node)* predicates which, in turn, is inferred as follows:

$$Pre(ag_1, t_1) \wedge Occ(ag_1, t_1, t_2) \wedge Post(ag_1, t_2) \Rightarrow Satisfied(ag_1, t_2) \tag{7.6}$$

The satisfaction of an anti-goal ag_1 at time-step t_1 presents a negative impact on the satisfaction of a target task a_i if ag_1 occurred just before the denial of the target task a_i

$$Satisfied(ag_1, t_1) \wedge Targets(ag_1, a_i) \wedge (t_1 < t_2) \Rightarrow !Satisfied(a_i, t_2) \tag{7.7}$$

Uncertainty Representation

The proposed framework relies on log data as evidence for proving the occurrence of the events of interest. However, the selection of events of interest (e.g., using LSI) is not perfect and is plagued by False Positive and False Negative entries. For this reason, we allow for user-defined weights representing the confidence a domain expert has on an observation for a given case. We address uncertainty in observations using a combination of logical and probabilistic models as follows:

A) The domain knowledge representing the interdependencies between systems/services is modeled using weighted first order logic statements. The strength of each relationship is represented with a real-valued weight set based on domain knowledge and learning from a training log data set and reflects. The weight of each rule represents our confidence in this rule relative to the other rules in the knowledge base. Consequently, the probability inferred for each atom depends on the weight of the competing rules where this atom occurs. For instance, the probability of the satisfaction of task a_2 in Figure 7.4 (*Satisfied(a_4,t)*) is inferred based on the Equation 7.8 with weight w_1,

$$w_1 : \quad Pre(a4, t) \wedge Occ(a4, t+1) \wedge Post(a4, t+2) \Rightarrow Satisfied(a4, t+2) \tag{7.8}$$

On the other hand, the contribution link based on Equation 7.9 with weight w_2 contributes to the denial of task a_4,

$$w2 : \ !Satisfied(g_2, t_1) \Rightarrow !Satisfied(a_4, t_2) \tag{7.9}$$

Consequently, the probability assignment given to $\mathit{Satisfied}(a_4, t)$ is determined by the rules containing this predicate as well as the weight of these rules in the knowledge base.

B) Applying an open world assumption to the observation where a lack of evidence does not absolutely negate an event's occurrence but rather weakens its possibility. By propagating knowledge from one part of goal model to another, we can possibly infer the truth value of missing observation. Knowledge propagation from the observation is done in three directions with respect to the goal model:

1. Bottom up: by knowing the truth values of the preconditions, postconditions and occurrence of a task we can make an assertion whether the task is satisfied (successfully executed) or not. Next, knowing the satisfaction truth value of all (at least one) of the AND (respectively OR) children subgoals of a goal g, we can infer the truth value for the satisfaction of the top goal g.
2. Top bottom: since the top goal's satisfaction is observable, an assertion can be made that all (at least one) of the AND (respectively OR) children of this goal (and recursively on its children) are satisfied or not.
3. Sideways: using the contribution links, the satisfaction (denial) of one node can be used to infer the satisfaction (denial) of others.

Logical Representation

We represent the goal model relationships in the form of production rules *(production $\rightarrow$ conclusion)*. To reduce the computational complexity, the rules in our KB are constructed using function-free Horn clauses where inference is more tractable. A Horn clause is an implication with only one literal in the consequent, and all antecedents are positive. Using Horn clauses, we define composite events from sub-events such as the case in Equation 7.1 where the precondition, postcondition and occurrence of a task imply its satisfaction. The inference algorithm used in this study is the MC-SAT. MC-SAT combines ideas from MCMC and satisability and performs better in the case when probabilistic and deterministic dependencies coexist [59].

7.2.3 Markov Logic Network Construction and Diagnostics

Weight learning for rules and observations is done semi-automatically by first using discriminative learning based on a training set of past cases as discussed in [28] and then manually refined by a system expert. During automated weight learning, each formula is converted to CNF, and a weight is learned for each of its clauses from past cases. The weight of a clause is used as the mean of a Gaussian prior for the learned weight. The learned weight can be modified to reflect our confidence in the rules. For example, in Figure 7.4, a rule indicating that the denial of the top goal g_1 implies that at least one of its AND-decomposed children (a_1, g_2, and a_2) have been denied, should be given higher weight than a rule indicating that g_1 is satisfied based on log data showing that *pre*(g_1), *occ*(g_1) and *post*(g_1) are true. This example relates to the case where he administrator has without any doubt witnessed the failure of the system, even if the observed log data do not agree (due to missing or inaccurate data).

Markov Network Construction

Consequently, the Markov Logic Network is constructed using an exhaustive scheme of rules and predicates as discussed earlier as well as, grounding predicates with all possible values, and connecting them if they coexist in a grounded formula.

Diagnostics

The flowchart in Figure 7.6 represents the process of detection of malicious behavior using the proposed framework. This process starts when an alarm is raised indicating a system failure, and completes when a list of root causes of the observed failure is identified. In order for the diagnostic reasoning process to proceed, we consider that the annotated goal/anti-goal models for the monitored systems and their corresponding Markov networks and logical predicates have been generated as described in Section 7.2. The diagnostic reasoning process is composed of seven steps as discussed in detail below:

Step 1: The investigation starts when an alarm is raised or when the system administrator observes the failure of the monitored service, which is the denial of a goal g.

Step 2: Based on the knowledge base KB that consists of all goal models and all observations (filtered log data and visual observation of the failure of the monitored service), the framework constructs a Markov Network that in turn is applied to generate a probability distribution for all the *Satisfied(node, timestep)* predicates for all nodes in the goal tree rooted at the denied goal g and the nodes of all other connected to it trees. This probability distribution is used to indicate the satisfaction or denial of tasks and goals at every timestep of the observation period. More specifically, if the probability of satisfaction of a task a_i at a timestep t is higher than α then we conclude that a_i is satisfied at t (i.e.,

$Satisfied(a_i, t) > \alpha \Rightarrow a_i$ is satisfied at t). Typically the timestep t starts at 0, and α is chosen to be 0.5. An example of the outcome of this step is the following based on Figure 7.4:

$$\begin{cases} Satisfied(a_1, 3) : 0.36 \\ Satisfied(a_3, 7) : 0.45 \end{cases} \Rightarrow \begin{cases} 1 : a_1 \\ 2 : a_3 \end{cases}.$$

Step 3: The outcome of the previous step is a ranked list of denied tasks/goals where the first task represents the node that is most likely to be the cause of the failure. In step 3, the framework iterates through the denied list of tasks identified in step 2, and loads the anti-goal models targeting each of the denied tasks.

Step 4: for each anti-goal model selected in Step 3, we generate the observation literals from the log data by applying the ground atoms generation process described in Section 7.2.2. Note that a task may have zero, one or multiple anti-goal models targeting it. The goal model in Figure 7.4 contains task a_2 which has no anti-goal models targeting it, task a_1 which has one anti-goal (ag_1) targeting it, and task a_3 which has two anti-goals (ag_3 and ag_5) targeting it.

Step 5: this step is similar to step 2 but instead of evaluating a goal model, the framework evaluates the anti-goal model identified in step 4 using the log data-based observation and determines if the top anti-goal is satisfied, i.e., has been successfully executed. This is done by constructing a Markov Network based on the anti-goal model relationships (see Section 7.2.2) and inferring the probability distribution for the *Satisfied(anti-goal, timestep)* predicate for the nodes in the anti-goal model. In particular, we are interested in the probability assigned to $Satisfied(ag_j, t)$ where t represents the timestep at which ag_j is expected to complete. Using the example in Figure 7.4, ag_1 takes 11 steps to complete execution, while ag_3 takes 7 steps. In this case, we are interested in $Satisfied(ag_1, 11)$ and $Satisfied(ag_2, 7)$.

Step 6: Based on the value of $Satisfied(ag_j, t)$ identified in step 5, we distinguish the following two cases:

1. If $Satisfied(ag_j, t) >= \alpha$ (typically $\alpha = 0.5$): this indicates that ag_j is likely to have occurred and caused a_i to be denied. In this case, we add this information (anti-goal and its relationship with the denied task) to the knowledge base KB_g. This knowledge base enrichment is done at two levels: first, a new observation is added to indicate that anti-goal ag_j has occurred; second, Equation 7.7 is added to the set of rules.

2. If $Satisfied(ag_j, t) < \alpha$: this indicates that ag_j did not occur, thus we exclude it as a potential cause of failure for a_i.

Before getting to next step, the framework checks whether the denied task a_i is targeted by any other anti-goal models. If so, it iterates through the list of other anti-goals ag_j that are targeting a_i by going back to Step 4. Once all anti-goals targeting a denied task a_i are evaluated, the framework checks whether there are more denied tasks and if so, it repeats step 2.

Step 7: Based on the new knowledge acquired from evaluating the anti-goal models, we re-evaluate the satisfaction of tasks based on the enriched knowledge based KB, and produce a new ranking of the denied tasks. The fact that an attack is likely to have occurred targeting a specific task that has failed increases the chances that this task is actually denied due to the intrusion leading to the overall system failure. This overall process helps improve the diagnosis but also provides interpretation for the denial of tasks.

7.3 Experimental Evaluations

To evaluate the proposed framework, we conducted the following set of experiments. The first case study aims examine the diagnostic effectiveness of the framework using a sample application utilizing COTS components. The objective of the second set of experiments is to evaluate the scalability and measure the performance of the framework using large data sets.

The framework's filtering and interpretation components were implemented using the Java Programming language (version 1.5). The diagnosis/inference was conducted using the Alchemy system developed at the University of Washington [59]. We also used Microsoft SQL Server 2008 to host the log database.

7.3.1 Malicious Behavior Categorization

In this study, we adopt the taxonomy of attacks based on the work in [64]. This taxonomy includes the following 4 classes of attacks:

Probe (scan attacks): active port exploitation for a known vulnerability of the corresponding service.

Denial of service (DoS): disruption of a host or network service by making a computer resource unavailable to its intended users.

Remote to Local (R2L): attempt to gain access in order to extract files and modify data in transit on the attacked machine.

User to root (U2R): elevation of privileges of a local user to ones normally reserved for the super user or the administrator.

In Figure 7.4, we show anti-goal models for attacks classified according to their category as per the categorization described above. We also show a goal model for a sample service (*files services*) being targeted by these attacks.

7.3.2 Credit Service Scenario

In the first case study for the proposed framework, we use the monitoring and detection of attacks on the execution of a credit evaluation business process implemented by a number of different services and COTS components.

Experiment Setup

The credit evaluation process starts upon receiving a Web Service request. If already available, the credit rating of the applicant is retrieved from the credit history table. For new applicants, the score is calculated and used to update the credit table. A Web Service reply containing the credit score is sent back to the front end application. In case one (or more) of the systems supporting the credit service fails, requests sent from the front end application will get no reply and will time out. The test environment (Figure 3.3 includes 4 systems/services: the front end application (soapUI), a message broker (IBM WebSphere Message Broker v7.0), the credit check Web Service and an SQL server (Microsoft SQL Server 2008). More details on the test environment are in Section 3.7.

The experiment is conducted from the perspective of the system administrator where a system failure is reported and an investigation is triggered. The case study scenario consists of running an attack while executing the credit history service (see the goal model of the credit service in Figure 7.3).The anti-goal ag_1 is executed by first probing the active ports on the targeted machine, and then the attacker attempts to login but uses a very long username (16000 characters) disrupting the service authentication process, and denying legitimate users from accessing the service. Traces of this attack can be obtained in the Windows Event Viewer.

The anti-goal ag_2 models an SQL injection attack that aims at dropping the table containing the credit scores. One way to implement such an attack is to send two subsequent legitimate credit history web service requests that contain embedded malicious SQL code within a data value. In particular, we embed an SQL command to scan and list all the tables in the data value of the field *ID* as follows:

<ID>12345; select from SYSOBJECTS where TYPE = 'U' order by NAME</ID>

The system extracts the credential data value (e.g., ID) and uses it to query the database thus inadvertently executing the malicious code. The scan is followed by a second WS request that contains a command to drop the Credit_History table:

<ID>12345; Drop Table Credit_History </ID>

Traces of the credit service sessions are found in the SQL Server audit log data and the message broker (hosting the credit service). The first anti-goal ag_1 represents the attacker's plan to deny access to the credit service to keeping busy the port used by that service. The second anti-goal ag_2 aims at injecting an SQL statement that destroys the credit history data.

Experiment Enactment

We run a sequence of credit service requests and in parallel we perform an SQL injection attack. The log database table contained 1690 log entries generated from all systems in our test environment. The interactions of the different components of the framework are described in the flowchart in Figure 7.6.

The first step in the diagnosis is to use the filtered log data to generate the observation (ground atoms) in the form of Boolean literals representing the truth values of the node annotations in the goal (or anti-goal) model during the time interval where the log data is collected. The *Planfile1* model segment below represents the observation corresponding for one credit service execution session:

Planfile1: *Pre(g_1 , 1); Pre(a_1, 1); Occ(a_1, 2); Post(a_1, 3); Pre(a_2, 3); Occ(a_2, 4); Post(a_2, 5); Post(g_2, 5); ?Pre(a_5, 5); ?Occ(a_5, 6); ?Post(a_5, 7); Pre(a_6, 5); ?Occ(a_6, 6); ?Post(a_6, 7); ?Post(g_2, 7); ?Pre(a_3, 7); ?Occ(a_3, 8); ?Post(a_3, 9); ?Pre(a_4, 9); ?Occ(a_4, 10); ?Post(a_4, 11); ?Post(g_1, 11), !Satisfied(g_1, 11)*

In the case where we do not find evidence for the occurrence of the events corresponding to goals/anti-goals/tasks execution (precondition, postcondition, etc.), we do not treat this

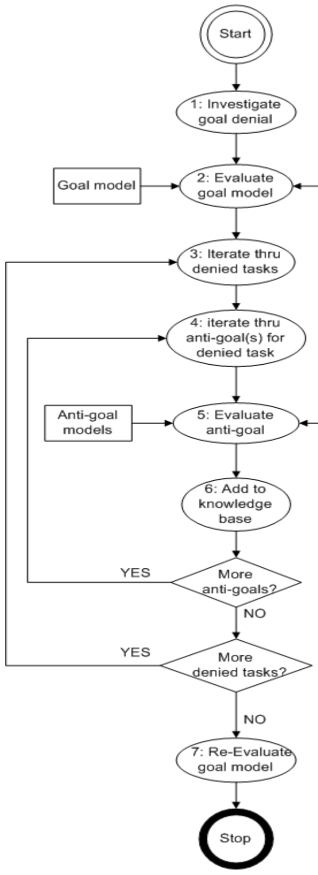


Figure 7.6: Malicious Behavior Detection Process

as a proof that these events did not occur but we consider them as rather uncertain. We represent this uncertainty by preceding the corresponding ground atoms with interrogation mark (?). In cases where there is evidence that an event did not occur, the corresponding ground atom is preceded with an exclamation mark (!).

By applying inference based on the observation in *Planfile1* (Step 2 in flowchart in Figure 7.6), we obtain the following probabilities for the ground atoms using the Alchemy tool:

$Satisfied(a_1, 3)$:0.99, $Satisfied(a_2, 5)$:0.99, $Satisfied(a_5, 7)$:0.30, $Satisfied(a_6, 7)$:0.32, $Satisfied(a_3, 9)$:0.34, $Satisfied(a_4, 11)$:0.03

Using step 2 in the diagnostics flowchart (Figure 7.6), we deduce that a_4, a_5, a_6 and a_3 are the root causes for the failure of the top goal g_1.

Using step 3 and 4 in the diagnostics flowchart, we first generate the observation for anti-goal ag_1 (*Planfile2*) and apply inference based on the anti-goal model ag_1 relationship and the generated observation.

Planfile2: *Pre(ag_1, 1), ?Pre(t_1, 1), ?Occ(t_1, 2), ?Post(t_1, 3), ?Pre(t_2, 3), ?Occ(t_2, 4), ?Post(t_2, 5), ?Pre(t_3, 5), ?Occ(t_3, 6), ?Post(t_3, 7), ?Post(ag_1, 7)*

In Step 5, the outcome of the inference indicates that ag_1 is denied (satisfaction probability is 0.0001). We iterate through step 3, and the next denied task a_5 is targeted by anti-goal ag_2. The observation generated for the anti-goal ag_2 is in *Planfile3* below:

Planfile3: *Pre(ag_2, 1), Pre(t_4, 1), Occ(t_4, 2), Post(t_4, 3), Pre(t_5, 3), Occ(t_5, 4), Post(t_5, 5), Pre(t_6, 5), Occ(t_6, 6), Post(t_6, 7), Post(ag_2, 7)*

The outcome of the inference indicates that ag_2 is satisfied (satisfaction probability is 0.59). Using step 6, we add the result to the goal model knowledge base. In step 7, we re-evaluate the goal model based knowledge base and generate a new diagnosis as follows:

$Satisfied(a_1, 3) : 0.95$, $Satisfied(a_2, 5) : 0.98$, $Satisfied(a_5, 7) : 0.23$, $Satisfied(a_6, 7) : 0.29$, $Satisfied(a_3, 9) : 0.11$, $Satisfied(a_4, 11) : 0.26$

The new diagnosis ranks a_3 as the most likely root cause for the failure of the top goal g_1. Next, a_5, a_4 and a_6 are also possible root causes but less likely. This new diagnosis is an improvement over the first one generated in step 2 since it accurately indicates a_3 as the most probable root cause. We know that this is correct since the SQL injection attack ag_2 targeted and dropped the credit table.

Inference Case Study

A case study of the inference was performed using Ubuntu Linux running on Intel Pentium 2 Duo 2.2 GHz machine. We used 3 sets of extended goal models representing the credit service goal model. The 3 extended goal models contained 50, 80 and 100 nodes respectively. In addition, we used a set of 5 anti-goal models with a total of 40 anti-goals nodes. In particular, we are interested in measuring the impact of the size of goal and anti-goal models on the inference component. Figure 7.7 illustrates that the number of ground atoms/clauses, which directly impacts the size of the resulting Markov model, is linearly proportional to the goal/anti-goal models total number of nodes.

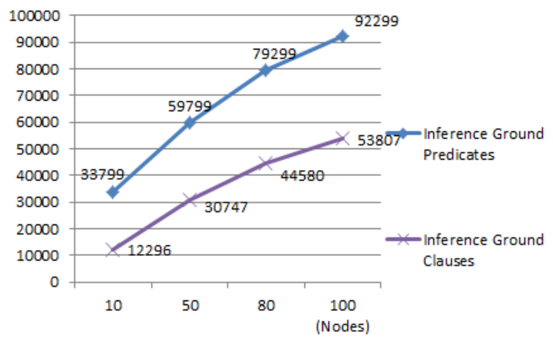


Figure 7.7: Impact of goal model size on ground atoms/clauses for inference.

Figure 7.8 shows that the inference time ranged from 42 seconds for a goal model of 50 nodes (10 goal nodes and 40 anti-goal nodes), up to 124 seconds for a model of 140 nodes (100 goal nodes and 40 anti-goal nodes). These results show that our approach can be applied to industrial software applications with small to medium-sized requirement models.

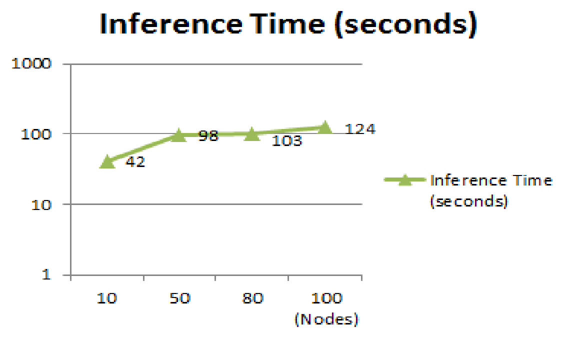


Figure 7.8: Impact of goal model size on inference time.

Threshold Alpha Impact

In the experiment above, we set the threshold value α as 0.5. In this experiment, we vary the value of the threshold α and analyze the impact on the diagnostic process, in particular, on steps 2 and 5 (Figure 7.6).

In the first case, we set α to 0.75. This has no impact on step 2, but it impacts step 5. Since both the anti-goals ag_1 and ag_2 have a probability of satisfaction below 0.75 and then they are considered to have not occurred and have no impact on the goal model evaluation. the result is that step 7 does not cause any modification to the original diagnosis.

In the second case, we set α to 0.25. The α value change impacts the decision on what tasks are denied (step 2). In this case, the diagnosis in step 2 indicates that only a_4 is denied. Similarly, a_4 is targeted by ag_1 which has no trace in the observation and thus has no impact on the overall diagnosis process. The final result is that the diagnosis in step was not modified in step 7.

As shown in the two experiments above, the threshold parameter α affects the overall diagnosis process and has to be tuned accordingly either by the user or by considering past cases.

7.3.3 Performance and Scalability

The second experiment evaluates the performance and scalability of the framework. We use the publicly available DARPA (Advanced Research Projects Agency) intrusion data sets, which is collected and provided by the IST Group at MIT [64]. In particular, we use the Windows NT Data Set for DARPA 2000 [60]. When stored in our common log database, the data set amounted to 153829 entries and took up to 600 MB of disk space. We use annotated anti-goal models to model the set of the attacks that are known to have traces in the DARPA 2000 Windows NT data set (see Figure 7.4).

Filtering Performance

This experiment measures the performance of log filtering. To do so, we apply the detection framework on the NT event viewer log data in the DARPA dataset, and measure the time required (in milliseconds) to perform the computationally demanding steps mainly the normalization and filtering.

First, the normalization process parses the log data files which is in the comma separated value format (CSV) which was exported from the Windows NT event viewer (evtx) format. Next, the log data is transformed into the unified schema format and stored in the log database. Last step consists of indexing the log table based on the timestamp field.

Second, the filtering was done using two techniques: Grep and LSI. As expected, the performance of the filtering process varied depending on the technique used. The first step in the filtering is common for the two approaches and it consists of parsing the goal/anti-goal models stored as local XML files, and collecting the nodes and their associated queries.

Table 7.2: Performance Measures for Normalization and Filtering (Milliseconds)

	Normalization	LSI Filtering	Grep Filtering
Processing Time (145855 entries)	178422	708876	181299
Average Time per Log Entry	1.22	4.86	1.24

For Grep based filtering, the second step is to transform the collected queries into regular expressions and find matching log entries.

Based on the processing rates shown in Table 7.2, the framework can handle few hundred log entries per second thus it scales up to an industrial environment.

7.4 Summary

This chapter discusses a root cause analysis technique that can be used to identify the causes of failures resulting from third party malicious actions. The technique is based first, on modeling with goal models the conditions, constraints, tasks, and actions that are required for the system to deliver its functionality, second on modeling with anti-goal models the different ways an intrusive third party action can be manifested, third on a knowledge base that represents as rules and facts the aforementioned models and the logged events and, fourth a probabilistic reasoning technique that is based on Markov Logic Networks, that allows for the evaluation of possible and alternative ways to explain the failed behavior of the system. More specifically, the proposed approach upon the observation of a failure not only computes the alternative ways this failure is supported but also, computes whether these alternative ways can be manifested as results of known intrusive actions. Consequently, the proposed approach best fits to detect the class of known (as opposed to mutating or new) intrusive behavior that can be documented using anti-goal AND-OR tree model. The probabilistic reasoning framework increases the probability of an identified root cause when the corresponding root cause can be supported by an intrusive behavior.

Chapter 8

Practical Considerations for Log Schema in Root Cause Analysis

In this chapter, we describe a set of industrial logging formats and propose a logging standard that streamline the process of logging between the monitored applications and the monitoring requirements. In addition, the size of the average log entries using this standard is minimal compared to some of the existing logging data.

The chapter is organized as follows. Section 8.1 covers the motivations behind the work in this chapter. Section 8.2 presents the guidelines and the details of the proposed logging schema. Section 8.3 describes and compares a set of industrial logging standards. Section 8.4 contains functional and performance evaluations for a set of industry logging standards. Section 8.5 presents a summary of the chapter.

8.1 Motivation

RCA systems are designed to respond to raised alarms by tracing software changes and generating a diagnostics in a timely manner. This is achieved by monitoring the software systems and collecting the generated log data; the collected log data are then analyzed in order to identify failures and pinpoint problematic components.

One of the challenges for collecting evidence for RCA systems from log data is in the lack of widely adopted standard in event logging [58]. More specifically, most logs are missing crucial information for the correct operation of autonomic systems. A study by Oliner et al. suggests that existing logging approaches lack an operational context that captures the system's expected behavior, which in turn lead to a lower confidence in the automated log analysis. Moreover, these logs may contain large amount of redundant information about some events while no information on others leading to asymmetrical reporting. Finally, logs commonly generated by computer systems suffer from the lack of explicit correlation information, having heterogeneous structures and containing corrupted log records. In addition, the sheer size of the generated log data, often make it intractable for it to be used in order to determine the root cause of a specific problem. These challenges make the interpretation of the log data more challenging which makes it impractical or at best reduces its usefulness for RCA systems.

Another challenge for logging systems stems from the exponential increase of the log data generated by commercial systems over the last years. Systems such as web Servers, database management systems, etc., usually generate large amounts of log data during short time periods. In addition to the cost of storage, these large amounts of log data can fill up the local storage space and potentially crash the host systems. In addition to its use for RCA systems, log data may sometimes be considered as part of the business data and is required to be archived for a long period of time due to financial, health or governmental regulations. In fact, North American communications operators are required to keep call records (CDR's) for periods up to 2 years. HIPAA, the Health Insurance Portability and Accounting Act in the US, requires the retention of patients' health information for their lifetime plus two years. In essence, concise logging is preferred for conservation of storage space, but some reality factors favor long logs due to its completeness. Therefore, it is important to optimize the design of the log data records so that it includes enough information to help in the subsequent analysis or auditing phases but reduce any redundancy.

More motivation for proposing a standard logging schema for software systems stems from the software development lifecycle itself. Decisions about instrumentation are usually left to the late stage of development, and typically are made individually by developers who may or may not be aware of how this log data will be used later. Due to the lack of clear guidelines on when to log, the generated log data is largely asymmetric and redundant. The asymmetrical reporting can be observed in most existing industrial systems where some events are logged multiple times while other events are not reported at all.

8.2 Standard Logging Schema

Most existing logging approaches focus on the contents of the log data and rarely provide guidelines on what events should be logged. Log data generated by existing commercial systems is generally incomplete and/or largely asymmetrical and thus is of little use for monitoring purposes. A more structured and commonly used logging approach is log events signaling the start/end of processes and services, as well as failure events. Log data generated using this approach is usually directed at developers and administrators but in general does not provide enough information in order to provide a reliable and automated root cause analysis.

A better approach is to design log data based on the type of information needed for monitoring purposes. A good criterion to use is based on requirement goal models. Enriching goal models with annotations describing precondition, occurrence and effect events allows us to find evidence of the execution for these tasks in the software system. Such evidence can be help show whether the monitored system is adhering to its requirements specifications and if not assist in pinpointing the root cause for the failure.

An example of a goal model annotation based on OCL is the *a15 (Retrieve Credit Rating)* goal node of Figure 3.7 which has a precondition of,

$context\ a15$
$inv : self.precondition- > forAll(e|$
$e.Description = \text{“}Database\ LOGS\ started'' \ AND\ e.Source\_Component = \text{“}MS\ SQL\ Server''$
$AND\ e.Severity = SEVERITY.NORMAL\_OPERATION$

and an effect of,

$context\ a15$
$inv : self.effect- > forAll(e|$
$e.Description = \text{“}SOAP\ Web\ Service\ Loan\ Reply\ Submitted\ for\ applicant\ 'Jane\text{”}'\ AND\ e.Source$
$\_Component = \text{“}IBM\ Process\ Server''\ AND\ e.Severity = SEVERITY.NORMAL\_OPER-$
$ATION\ AND\ e.Physical\_Address = \text{“192.168.0.1''}$

8.2.1 Guideline for Logging Schema

The level of sophistication of techniques applied to analyze log data is of little relevance if the quality of the log data being analyzed is poor. In order to ensure a high level quality log data that contains information useful for autonomic computing and monitoring, we design a schema for logging using the following guidelines:

- Expressiveness: a log entry should clearly identify the reported event by including enough context information about this event such as event name, event type ID, source component, etc. Most existing logging schema suffer from being either overly complicated (e.g., Common Base Event (CBE)) or too simplistic (e.g., Apache HTTP Server error log), subject specific (e.g., Intrusion Detection Message Exchange (IDMEF)) or vendor specific (e.g., Windows Event Viewer). The logging schema should contain enough details for it to be useful in subsequent analyses, but not too complicated to drive away developers.

- Uniqueness: the log entry should unambiguously point to one specific event instance. This can be achieved by including temporal and geographical information about when and what system generated this event instance. Furthermore, session and correlation information are necessary to distinguish similar events that could be generated by the same system during short time intervals.

- Minimal log record size: careful attention should be paid to the size of each log entry to avoid a large redundant log data corpus that may result in higher storage costs and delays in indexing/retrieval times.

- Comprehensiveness: to ensure a good quality in reporting of events of interest, instrumentation goals can be captured and formally documented ahead of the development phase. This will ensure a comprehensive coverage of events.

- Ease of use: log data should include direct information about the reported event. Information about an event should be explicit and the occurrence of an event should be inferred without the need of further complex processing.

On the other hand, the proposed log format has two facets: physical and logical. The proposed physical data format consists of using space as a field delimiter, and carriage return as a records terminator. Using tag delimited records is more flexible than fixed size fields based formats since it sets no limit on the size of data field. It also leads to smaller log records compared to equivalent XML based documents. The proposed data types for the log fields are the following: String, hexadecimal, integer, decimal and percentage. Strings and timestamps are identified with quotes, while hexadecimal and percentage values are prefixed with “0x” and “%”.

The proposed schema contains 13 fields that cover 5 categories: general (Report_Time, Description), event specific (event_type, event_ID), diagnostic related (Cause, Hypothesis, Prescription), session specific (Session_ID, Correlation_ID), environment related (Logon_ID, Source_Component, Physical_Address). Below we describe in detail each field:

1. **General:**

   (a) *Report_Time*: is represented as a string of the form *yyyy-mm-dd hh:mm:ss.[fff...]*. For cases where log data is stored in a database, a more efficient approach, but less human readable, is to store the timestamp as the number of seconds since 00:00:00 GMT on Jan 1, 1970 (UNIX or POSIX time). Typical log data querying involves the use of the creation time; thus, using a long integer representation is more efficient than using Strings. The following timestamp is an example using the proposed format: *'2009-08-11T21:21:22.000'*. When converted to UNIX time, the equivalent timestamp is *1217971282*.

   (b) *Description*: this is a free format field that, along with the timestamp field, appears in almost every currently generated log data. It is traditionally the most useful field for developers and administrators. Similarly to existing approaches, the proposed logging schema includes a description field that consists of static and dynamic sections. The static component is a template that is common for all log entries of the same "class". The dynamic part contains business and instance specific information that are specific to the current session or transaction such as the applicant name, branch name, etc. For security and privacy reasons, information such as credit card, social insurance numbers, etc. should not be included here. An example of the description field is the following: *The HTTP Listener has started listening on port "7800" for "http" connections*. Finally, since this field is the only free formatted field in the log data, this is the only place to include application and business specific information that can be helpful in formulating queries and extracting relevant log data at runtime.

2. **Event Specific**

   (a) *Event_ID*: is a unique identifier automatically generated for each event instance. The value assigned for this field is a randomly generated number that uses the timestamp as a seed. The uniqueness of the field is only needed at the local machine level. When combined with the host name or address, this id can be

Table 8.1: Unified Schema for Log Data

Field Name	**Category**	**Description**
Report_Time	General	A string of the form *yyyy-mm-dd hh:mm:ss.[fff...]*. For example: *'2011-02-11T21:21:22.000'*.
Description	General	Consists of static and dynamic sections. The static part is a template that is common for all log entries of the same "class". The dynamic part contains business and instance specific information that are specific to the current session/transaction such as the applicant name, branch name, etc. Example: *The HTTP Listener has started listening on port "7800" for "http" connections*
Event_ID	Event Specific	A unique identifier for each event instance. The uniqueness of the field is usually at the level of the machine generating the event. When combined with the host name or address, this id can be safely assumed as unique across the environment. Example: *12331111*
Event_Type_ID	Event Specific	Represents a type or a class of events. Example *1.2.11*
Event_Severity	Event Specific	Indicates whether this event is an audit event (normal operation) or a failure. In case it is a failure, a level of severity is assigned to it. The enumeration values are: *normal_operation*, *low_severity*, *medium_severity* and *high_severity*
Hypothesis	Diagnosis Information	Contains information that describes potential causes of the problem that has resulted in the system to be in the current state. This information is unverified but conceived at design time to help diagnose the root cause problem

Cause	Diagnosis Information	A description of the internal fault that immediately led to the observed failure
Prescription	Diagnosis Information	Contains suggested actions that can be used to help rectify the current problem. This information is customized based on a protocol (template) that includes a list of actions that are to be applied to certain entities. The following are examples prescriptions: *Restart web server Alpha* or *Unlock logon id for user Jane*
Session_ID	Session Information	Used to uniquely identify a business or technical transaction. Events belonging to the same transaction have the same session id and can be randomly generated by using the session start time of the first event in this session as a seed
Correlation_ID	Session Information	This field is commonly confused with the session identifier. A correlation identifier correlates one session to another, such as the correlation of a reply session to its request session. The reply session ID can be assigned the same value of the request session correlation ID
Source_Component	Environment Related	Name of the resource experiencing the event such as the name of the application server, ...
Logon_ID	Environment Related	User logon ID under which the application experiencing the event is authenticated to run.
Physical_Address	Environment Related	Contains either the host name or the IP address of the resource generating the event

safely assumed as unique across the environment.

(b) *Event_Type_ID*: represents a type or a class of events. In fact, events that belong to the same class of events have the same static text in the description field. The values used in this field are assigned based on a hierarchical numbering scheme as suggested by [83]. Salfner et al. (2004) propose a hierarchical numbering scheme based on the SHIP (i.e., software, hardware, interoperability, people) tree model as shown in Figure 8.1. SHIP classification is useful and can be easily used to support clustering algorithms. An example of an Event_Type_ID field is the following: 1.2.1. This class of events represents an error caused by an invalid data value during the execution of a software program.

(c) *Event_Severity*: this field is used to indicate whether this event is an audit event (normal operation) or a failure. In case it is a failure, a level of severity is assigned. The enumeration values are: *normal_operation*, *low_severity*, *medium_severity* and *high_severity*. The severity data field is common in existing logging systems; however, most of the existing loggers use enumerations such as *warning, critical*, etc., where the severity of the reported event/problem cannot be directly inferred without knowing the whole list of enumerations values, which may not be easily accessible. The proposed set of enumeration values is concise and self-explanatory.

3. **Diagnosis Information** The following 3 fields must not be used when the severity field is set to *normal_operation*.

   (a) *Cause*: contains a description of the internal fault that immediately led to the observed failure and caused the system to be in the current failed state.

   (b) *Hypothesis*: contains information that describes potential causes of the problem that has resulted in the system to be in the current state. This information is unverified but conceived at design time to help diagnose the root cause problem.

   (c) *Prescription*: contains suggested actions that can be used to help rectify the current problem. This information is customized based on a protocol (template) that include a list of actions that are to be applied to certain entities such as restart Alpha web server, or unlock user Jane logon id, etc.

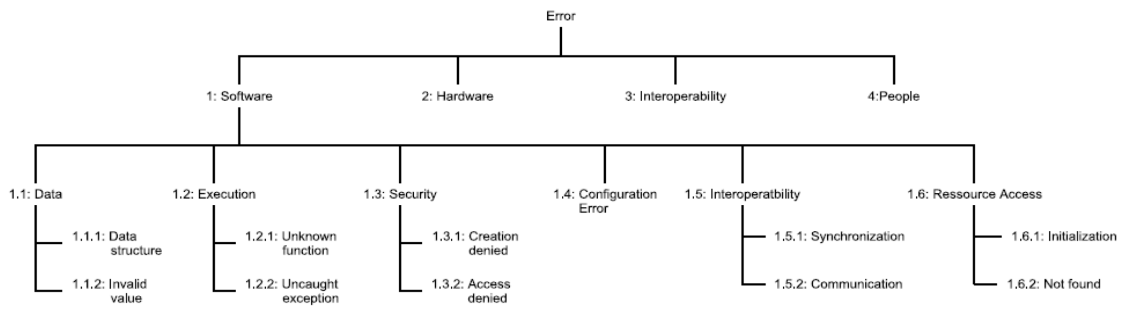


Figure 8.1: SHIP Based Hierarchical Numbering Scheme for Event Types' Identifiers.

4. **Session Information**

   (a) *Session_ID*: used to uniquely identify a business or technical transaction. Events belonging to the same transaction have the same session id. The session ID can be randomly generated by using the session start time of the first event in this session as a seed.

   (b) *Correlation_ID*: this field is commonly confused with the session identifier. In fact, the correlation identifier correlates one session to another. An example of such correlation is to correlate a request session with its reply session. In this case, the reply session ID can be assigned the value in the correlation ID of the request session.

5. **Environment Related**

   (a) *Source_Component*: name of the resource experiencing the event. For example the name of the application server, web service, database name, etc.

   (b) *Logon_ID*: user logon ID under which the application experiencing the event is authenticated to run.

   (c) *Physical_Address*: this field contains either the host name or the IP address of the resource generating the event.

8.3 Industry Case Studies

In this section, we discuss the quality of log data generated using two common industry logging standards: the Windows Event Viewer, and the Common Base Event. In addition, we analyze the log data generated by two commercial software applications: WebSphere MQ Series 7.0, and Microsoft SQL Server 2008. The analysis involves the quality of the log entries (expressiveness) as well as the coverage of the event reporting (comprehensiveness). In each of the case studies, we calculated the average size of generated log data records using a sample size of 100 records, with a 95% confidence level, and an 8.5% confidence interval.

8.3.1 Industrial Logging Standards

Windows Event Viewer

Log data generated by applications running under Windows can be captured and viewed in the *Windows Event Viewer*. The *Event Viewer* allows a separation between the software that generates log messages, the system that stores these messages and the tool that reports them. Event Viewer tracks events in several categories: Application (program), security-related (audits), setup, system and forwarded events by other computers. Log data incoming from Windows applications are viewed in the application folder in the Windows Event Viewer. The Event Viewer provides two data views: a human friendly view and an XML view. The size of the XML version is typically 3 times the non-XML view. The schema contains direct information related to session or diagnosis. On the other hand, the reported events are typically the startup and shutdown (normal and unexpected) of services and processes. An example of an event viewer based log entry is shown below:

Log Name: Application
Source: MSSQL$ENTERPRISESERVER
Date: 20/12/2010 2:41:29 PM
Event ID: 5084
Task Category: Server
Level: Information
Keywords: Classic
User: Jnoubi-PC/Jnoubi
Computer: Jnoubi-PC
Description:

Setting database option COMPATIBILITY_LEVEL to 100 for database ReportServer$ENTERPRISESERVERTempDB.

Common Base Event

The Common Base Event (CBE hereafter) is one of the early efforts to standardize log data format for the purpose of autonomic computing. CBE development was led by IBM and endorsed by of the OASIS Web Services Distributed Management Technical Committee, and was first released in 2003 [58]. In fact, the motivation for developing CBE is not only limited to autonomic computing but also aims to support enterprise management and information exchange between e-business applications. A CBE event is designed to represent several types of events such as logging, tracing, management, and business events [50].

The CBE logging schema is a strongly typed and an extensible approach. Using CBE terminology, an event represents message data sent as the result of an occurrence of a situation (failure, state change, etc.) Using CBE, the completeness of a situation reporting is considered to be achieved by providing information on the following:

1. Component reporting the situation,
2. Component affected by the situation,
3. The situation itself.

The three types of information described above are referred to as the 3-tuple information. The CBE schema contains 18 fields including 4 that are mandatory. The mandatory fields are: creation time, situation, source component identification, and reporting component identification if different from source component. CBE includes two optional fields (*localInstanceID* and *globalInstanceID*) for event identification. Session information can be captured using a field that captures a list of IDs representing the events that are associated with the current event (*associatedEvents*), and another to capture the order of the current event in the events sequence (*sequenceNumber*). CBE does not contain detailed diagnosis information in case the event is a failure. Instead, it contains a complex field (*situation*) that contains the event description and category based on an enumeration (start, stop, connect, configure, request, etc.)

Since the CBE schema contains a large number of optional fields, the average size of a log entry largely depends on the implementing application. On the other hand, the

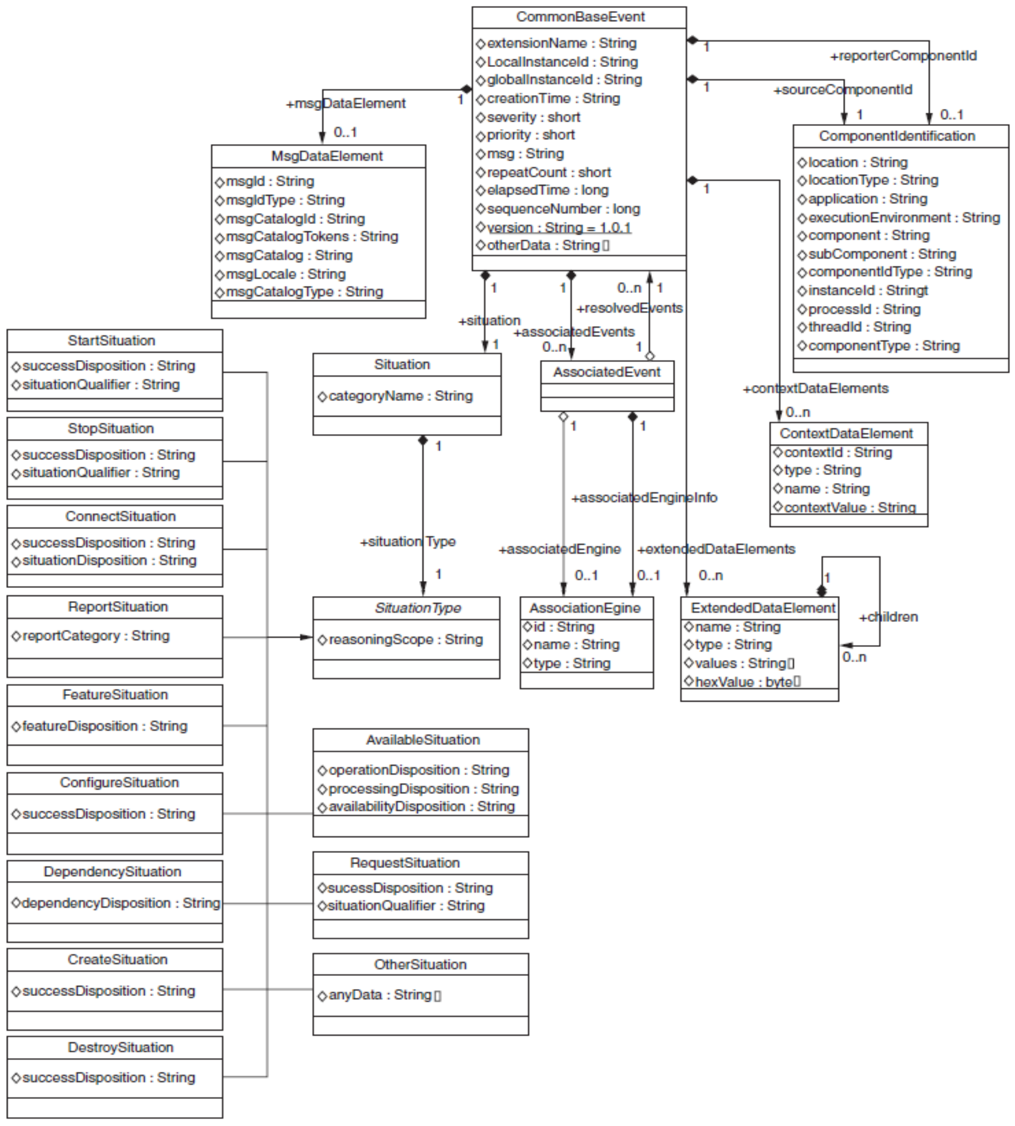


Figure 8.2: The Common Base Event Schema

format of the CBE event is expressed as an XML document using UTF-8 or UTF-16 encoding. In general, serialization using XML is less space efficient compared to custom wire formats and tag delimited strings. CBE data generated by the IBM WebSphere Process Server contained on average 1854 characters per log entry record where only 385 characters contained data values. Furthermore, the global event id can be easily replaced by combining the host name with the local as described in section 8.2.1.

8.3.2 Commercial Software Studies

WebSphere MQ Series 7.0

WebSphere MQ is an IBM product for providing a universal messaging backbone across diverse platforms and supporting the service-oriented architecture strategy. It consists of a set of queue managers and their related processes (listeners, channels, queues, etc.) An example log entry generated by MQ is the following:

```
14/01/2011 13:04:19 - Process (3876.3) User (MUSR_MQADMIN) Program (amqzmur0.exe)
Host (JNOUBI-PC)
AMQ6287: WebSphere MQ 7.0.1.0.

EXPLANATION:
WebSphere MQ system information:
Product :- WebSphere MQ for Windows
Version :- 7.0.1.0
Host Info :- Windows Ver 6.1 (1) Unknown x64 Edition, Build 7600
ACTION: None.
```

The MQ generated log is missing session information. In addition, it does not contain information regarding the severity of the reported event. It also does not assign unique identifiers for each reported event. On the other hand, the MQ log data contains *process ID*. The process ID is platform specific field is dynamically generated which we consider to be of little use for administrator or to be included in queries in automated analysis. In terms of MQ log data entries size, the average size is relatively small (350 characters per log entry). Log entries are multi-line records which can complicate parsing the log data. On the other hand, each field value is preceded with the name of that field making it more human readable. The contents of the description field are well designed and contain technical transaction information which can be used to distinguish events from the same class, and automate the retrieval of the log data. As an example, *The Queue Manager task 'APP-SIGNAL' has ended* contains the name of the specific task that has ended. In terms of comprehensiveness, the MQ logging generates a log entry when a process (queue manager, channel, listener, etc.) is started or ended (normally or abnormally). Thus, the reporting is symmetrical to the number of events. However, MQ does not provide further granularity or capability to customize logging in order to capture events such as preconditions and effects.

Microsoft SQL Server 2008

The Microsoft SQL Server 2008 provides multiple levels of logging. The high level logging is based on the standard Windows event viewer format, where all events related to the database instances and processes' startup or shutdown are captured. The SQL Server provides a comprehensive and granular auditing at the SQL commands execution level such as SQL Insert, Update, etc. The logging schema used in this level logging contains 33 data fields, and has an average of 530 characters per log entry.

The schema for the audit log of the Microsoft SQL Server 2008 contains some redundancy. For example, the date field contains the time of the reported event; however, there is another field that also contains the time. There are two fields reporting the result of the operation (Success and Succeeded). The diagnosis information is very brief and contains binary type of information (either success or failure). The following is an example of a log entry in the SQL Audit:

```
08/22/2010 22:11:27,,Success,22:11:27.0217319,JNOUBI-PC\ENTERPRISESERVER, SELECT,TABLE,1,True,1,True,65,259,1,0,0,2121058592, NT AUTHORITY\SYSTEM, NT AUTHORITY\SYSTEM,0x1100000518000,dbo,,NULL,,LOGDATA,dbo,LogData,select event_id<c/>report_time<c/>source_component<c/>event_severity<c/>situation<c/>correlation_id<c/>description<c/>event_type<c/>address from LogData where 1=2,,C:\PhD\DatabaseAudit\LogData%5Table%5Audit_8E006B41-4744-4C4C-AFC4-A5F8A65D7099_0_129269884845020000.sqlaudit,3072
```

8.4 Experimental Evaluation

8.4.1 Evaluation of Industrial Log Data Size

In this section, we compare and evaluate the log data schema for four commercial products based on the criteria and guidelines in section 8.2.1. Table 8.2 contains the comparison results. We measure the expressiveness of each schema design based on whether it contains fields describing the event name or ID, event type name or ID and source component. We measure the uniqueness aspect of a schema design based on whether an event can be identified uniquely within a transaction or a session; hence, we look for fields representing process ID, session ID and correlation ID (to correlate request/response). In order to estimate the average size of log data entries, we evaluate the average number of letter per log entry by using log samples that contained between 100 to 1000 log entries. We assess the comprehensiveness of a schema by examining if it allows the representation of the start/end of the system as well as if it is successful or not. Note that these metrics focus on the structured part of the schema and do not evaluate the contents of the description field in each schema. Finally, we do not provide a metric for the ease of the use since it varies greatly depending on the domain of use.

Table 8.2: Log Data Schema Comparison

System	**Expressive**	**Uniqueness**	**Average Size (letter)**	**Comprehensive**
Windows Event Viewer	Missing event entry ID	Missing process ID, session ID and correlation ID	385	Missing start/end states
Common Base Event	Satisfactory	Satisfactory	384	Satisfactory
WebSphere MQ	Missing event type and ID	Missing correlation and session ID	302	Missing success/failure states
Microsoft SQL Server Audit	Satisfactory	Missing process ID	439	Missing start/end states

8.4.2 Events Generation Impact on Performance

In this experiment, we show the impact of the event generation on the overall performance of the system. In order to ensure that our experimental results are statistically significant, we conducted a two-sample t-test. First, we computed the sample size needed for a two-sample t-test with the assumption of the normality of the sample population. We choose a typical power of 0.84, i.e., we seek 84% chance of finding statistical significance if the specified effect exists. We also choose a typical significance level of 0.05 in order to reduce the probability of having conclusions due to chance alone. With this power and significance level, we study the relationship between the response time of loan applications processing using the test environment used in this study, and the generation of events. We run the loan processing and measured the response times. The first set included 100 runs without events generation and then another 100 runs where the loan business process and its supporting applications generated events. When event and logging generation is disabled, the standard response time is [0; 0.063] seconds.

When logging/event generation is enabled, the standard response time jumps to [0.156; 0.622] seconds. The difference in mean response time between the case when events are not generated and the case when events are generated, with a confidence interval of 95% interval, is [0.315; 0.407] seconds. This indicates that the difference in the means is not 0 and thus the impact of the events generation is not negligible. In order to determine the likelihood of the role of chance in explaining this difference, we ran a paired t-test. Our null hypothesis is that there is no impact of events generation on the response time of the loan application, i.e., there is no significant difference in the means of response times between the case when events are generated and when events are not generated. The result of the t-test is much less than 0.0001 which indicates that the difference in response time averages is very unlikely to be due to chance. This supports our conclusion that event generation have a significant impact on the performance and response times of the software systems in a distribute environment.

8.5 Summary

In this chapter, we propose a logging approach that aims at addressing the shortcomings in the existing industrial logging approaches. This approach aims at generating log data that is minimal in size, comprehensive in representing events of interest, and expressive by including enough information that uniquely and directly identifies the generating event, and reduces the information redundancy.

Chapter 9

Conclusions

The automation of root cause analysis for detecting service degradations and failures in software systems has been a challenging yet an attractive concept for researchers and practitioners alike. The interest in automation stems from the increased dependency of businesses on their IT infrastructure. The increase in the business demand for IT resources has led to unprecedented escalation in the sheer numbers of servers and data storage devices required to support the business operations. In addition, the business competitiveness and higher consumer expectations lead to stricter quality of service agreements such as high availability and shorter problem resolution times in IT systems.

The challenges in the automation of root cause analysis for IT environments are mainly due to the complexity in modeling the monitored applications as well as extracting evidence that can be used in the investigation process. In particular, the wide adoption of recent paradigms such as cloud computing and service oriented architectures resulted in highly dynamic but also highly complex systems which required more sophisticated and adaptable models to represent them. On the other hand, an intrinsic step of root cause analysis is to gather observation data and use it as evidence to support any conclusions. This process of collecting evidence is typically be done by adding monitoring devices, instrumenting the monitored systems and/or the extracting natively generated log data. The use of external monitoring devices or instrumenting the monitored system code can be expensive, intrusive and/or may cause performance degradation. Alternatively, relying on natively generated log data as a source of observation poses many challenges due to the sheer size of the log data, its heterogeneity and incompleteness.

In this thesis, we address all these RCA challenges and propose effective solutions. More specifically, after presenting the introduction, related work and research baseline in in Chapters 1, 2 and 3, we propose in Chapter 4 a framework for reducing log data and making the evidence collection from log data a more tractable process. Log reduction is achieved by applying a probabilistic noise reduction process which discards log data that are deemed not relevant based on a criteria guided by a goal model. In Chapters 5 and 6 we describe log filtering using a set of techniques including SQL, Grep, latent semantic indexing and a hybrid semantic approach that incorporates the use of regular expressions. Similarly to log reduction, the process of log filtering is guided by criteria that uses annotated goal model in order to highlight the events of interest. The outcome of this process is a set of log data subsets that are used to generate an audit trail which in turn can be used as an observation for the monitored environment. Chapter 7 describes an approach for root cause analysis for distributed software systems based on SAT solvers. Chapter 8 addresses some of the shortcomings of the framework in Chapter 7 and describes a probabilistic approach for root cause analysis for distributed software systems based on Markov Logic Networks. Chapter 9 extends the work in Chapter 8 by handling failures caused by malicious behavior rather than internal faults. These RCA frameworks use requirement goal trees to model the runtime behavior of the monitored applications. The log reduction and filtering approaches presented in earlier chapters are complementary to the RCA frameworks by providing means to generate a set of observations that can be used to support or deny the diagnosis. Chapter 10 proposes a logging approach based on our experimental evaluation of a set of industrial logging schema.

9.1 Conclusions

In this section, we present the conclusions of the thesis as follows:

1. The experimental results indicate the feasibility and scalability of the proposed log reduction and filtering approaches.

2. The log reduction approach based on PLSI is experimentally shown to be capable of significantly reducing the size of the log data to be considered for a given task/objective with minimal loss of true positives.

3. The experimental results of the LSI-based log filtering shows improved recall when compared to traditional techniques such as Grep. Alternatively, the precision of the filtering using traditional techniques such as SQL and Grep proved to be superior to LSI based filtering when the query formulation was trained.

4. The query formulation for LSI based filtering is much simpler than in the case of Grep or SQL since it allows for queries to build using synonymous keywords and without any dependency on the order of the keywords in the query.

5. The experimental evaluation of the hybrid RegEx-LSI approach shows a significant improvement in the quality and performance of log indexing and extraction compared to the traditional LSI-based log filtering.

6. Another factor that affects the RCA is the availability of the logging system itself. As an example, one of our experiments with the RCA frameworks involved the injection of some faults. Although the failure in the application did occur as planned, this failure impacted in some instances the logging subsystem itself causing it to fail as well, which potentially lead to missing log data. The design of the logging system should emulate the design of black box in the airline industry.

7. The initial results show that the proposed deterministic and probabilistic RCA frameworks are scalable and feasible. In particular, goal models for large systems can be organized in a hierarchical fashion allowing for higher tractability and modularity in the diagnostic process. The scalability of the other components in the RCA frameworks such as the LSI based filtering, SAT solvers and MLN is well studied and established. Experimentally, the RCA frameworks have been evaluated in a medium size system that is composed of COTS components. Furthermore, the RCA framework for detecting malicious behavior described in Chapter 9 was successfully used in the identification of known intrusions in the DARPA intrusion dataset.

8. Initial experimental results of the probabilistic based RCA framework shows that it can handle the inaccuracies in observation much better than the deterministic approach based on the SAT solver. Moreover, the probabilistic based allows for multiple diagnoses to be achieved and ranked based on their probability of occurrence which is an advantage over the SAT solver based approach where all diagnostics are equiprobable.

9.2 Limitations

In this section, we present the limitations of the approaches and frameworks described in this thesis as follows:

- Prior to applying the RCA frameworks, there is phase of preparatory work that consists of developing the goal/anti-goal models and formulating queries to capture the conditions for the individual goals/tasks to occur. The queries formulation process is an iterative process of training where multiple scenarios are run (success and/or failure), the corresponding collected data are inspected and queries updated until satisfactory results are achieved.
- Precision and recall of the extracted and filtered log data is shown to be strongly influenced by the formulation of the goal model annotations. Therefore, in order to improve the log filtering and reduction processes, an iterative process of training the generation of the logical expressions that are annotating the anti-goal models must be considered prior to deployment.
- The quality of the filtered/reduced log data (and eventually the diagnosis based on this data) depends on the quality and availability of the natively generated log data. Low quality log data such as log data with too few fields or with generic description can lead to a degradation of the framework's precision/recall.

9.3 Future Work

In this section, we describe our ideas for the future work to handle the limitations of our work as follows:

1. Further experimentation using larger log data sets should be considered to generalize the performance of the log filtering and reduction frameworks.
2. An area of research and improvement in the domain of log data reduction concerns the development of heuristics for the selection of an optimal value for the parameter α that represents the JS distribution divergence threshold. We have experimentally set the value of α to 0.25, 0.5 and 0.75 but we have noticed that the optimal value varied depending on the log data corpus.
3. More experimentation is required on the RCA frameworks to identify the performance of the framework when multiple failures are involved. An example in SOA environment is when the average response time of a business process increases. This could be caused by multiple business processes using common services and data sources. The degradation of the performance of one process can be caused by a combination

of reasons such as the database response time degradation or increased demand on other process which is putting pressure on common shared services.

4. The RCA frameworks can be applied to perform root cause defect analysis. In this respect, the RCA framework can be applied as part of regression testing to detect any introduced bugs. For new software versions, the corresponding goal models can be updated with the new requirements that were added as part of the new release, and the RCA framework can be applied on the new system, thus contributing the software maintenance and evolution lifecycle.

5. In this thesis, we used a classification of system failures based on the following criteria: first, failures due to internal faults, failures due to external malicious agents, and system failures due to domain assumption failure. The framework in this thesis addressed the detection of failures of the first two types (internal faults and external agents). Future work can extend this framework to handle domain assumption failures. For this, one approach can be by enriching the goal models representing the monitored systems in order to capture the domain assumptions.

6. In order to improve the quality of the log filtering and reduction, iterative training for the query annotating the goal model can be applied using relevance feedback algorithms such as Rocchio algorithm. Using relevance feedback, we can train the LSI set of matrices that represent the queries by iteratively applying a training set and having the user rate the retrieved results as relevant/not relevant. In every iteration, the feedback algorithm uses the user's feedback to update the queries' vectors accordingly. Further experimentation can be done in this area to show the applicability of such an approach.

7. In this thesis, we used the learning algorithms of MLN to assign weights to the rules of the diagnostic knowledge base that represent the relationships between the monitored applications. Further investigation can be done to evaluate the structure learning capability of MLN in discovering the rules of the diagnostics knowledge base using a training set.

8. As described in Chapters 4, 5 and 6, matrix manipulations and calculations are central to the LSI-based log filtering and reduction which can be intractable when handling large amounts of log data. Further experimentation can be done in this area by using multi-processing frameworks such as Map-Reduce.

9. Chapter 7 describes our approach to handle failures caused to external malicious behavior which is modeled using anti-goal models. Thus, this approach is limited

to detecting known and documented types of malicious behavior. In the future, the range of the types of attacks that can be detected by this framework can be extended to unknown attacks that has similar behavior to known ones but with different file signature. One way to do this is by using techniques such as latent semantic indexing when proving an anti-goal model.

10. The approach used in this thesis to detect the root causes for the monitored systems failures is based on generating an audit trail from the log data. In fact, this audit trail is a set of literals that represent the success and failure events for the monitored components ordered based on the time of their occurrence. As a future work, this feature can be exploited to discover not only the root cause failure but also any cascaded failures.

Appendix A - Markov Logic Network Code

In this appendix, we include the knowledge base built and used as part of the experimental evaluation for the probabilistic based root cause analysis approach described in Chapter 7 and 8. The knowledge base contains a set of variables, predicates and weighted rules. In addition, we include one set of observations that corresponds to one of the failure scenarios. The specific failure scenario is the credit database failure (task A6) with an evidence of an SQL injection attack.

```
//Variables
node = G1,G2,AG1,AG2,A1,A2,A3,A4,A5,A6,T1,T2,T3,T4,T5,T6
int = 1,...,10

//predicate declarations
ChildAND(node,node)
Pre(node,int)
Satisfied(node,int)
Task(node)
ChildOR(node,node)
Post(node,int)
OccurrenceGoal(node,int,int)
Occ(node,int)
Succeed(int,int,int)

5 !Pre(A1,t1) v !Occ(A1,t2) v !Post(A1,t3) v Satisfied(A1,t4) v !Succeed(t4,t2,1) v !Succeed(t4,t1,2) v !Succeed(t4,t3,0)
5 !Pre(A2,a1) v !Occ(A2,a2) v !Post(A2,a3) v Satisfied(A2,a4) v !Succeed(a4,a2,1) v !Succeed(a4,a1,2) v !Succeed(a4,a3,0)
```

5 !Pre(A3,a1) v !Occ(A3,a2) v !Post(A3,a3) v Satisfied(A3,a4) v !Succeed(a4,a2,1) v !Succeed(a4,a1,2) v !Succeed(a4,a3,0)
5 !Pre(A4,a1) v !Occ(A4,a2) v !Post(A4,a3) v Satisfied(A4,a4) v !Succeed(a4,a2,1) v !Succeed(a4,a1,2) v !Succeed(a4,a3,0)
5 !Pre(A5,a1) v !Occ(A5,a2) v !Post(A5,a3) v Satisfied(A5,a4) v !Succeed(a4,a2,1) v !Succeed(a4,a1,2) v !Succeed(a4,a3,0)
5 !Pre(A6,a1) v !Occ(A6,a2) v !Post(A6,a3) v Satisfied(A6,a4) v !Succeed(a4,a2,1) v !Succeed(a4,a1,2) v !Succeed(a4,a3,0)

5 !Pre(T1,a1) v !Occ(T1,a2) v !Post(T1,a3) v Satisfied(T1,a4) v !Succeed(a4,a2,1) v !Succeed(a4,a1,2) v !Succeed(a4,a3,0)
5 !Pre(T2,a1) v !Occ(T2,a2) v !Post(T2,a3) v Satisfied(T2,a4) v !Succeed(a4,a2,1) v !Succeed(a4,a1,2) v !Succeed(a4,a3,0)
5 !Pre(T3,a1) v !Occ(T3,a2) v !Post(T3,a3) v Satisfied(T3,a4) v !Succeed(a4,a2,1) v !Succeed(a4,a1,2) v !Succeed(a4,a3,0)
5 !Pre(T4,a1) v !Occ(T4,a2) v !Post(T4,a3) v Satisfied(T4,a4) v !Succeed(a4,a2,1) v !Succeed(a4,a1,2) v !Succeed(a4,a3,0)
5 !Pre(T5,a1) v !Occ(T5,a2) v !Post(T5,a3) v Satisfied(T5,a4) v !Succeed(a4,a2,1) v !Succeed(a4,a1,2) v !Succeed(a4,a3,0)
5 !Pre(T6,a1) v !Occ(T6,a2) v !Post(T6,a3) v Satisfied(T6,a4) v !Succeed(a4,a2,1) v !Succeed(a4,a1,2) v !Succeed(a4,a3,0)

1 !Task(A1) v Post(A1,a1) v !Satisfied(A1,a2) v !Succeed(a2,a1,0)
1 !Task(A2) v Post(A2,a1) v !Satisfied(A2,a2) v !Succeed(a2,a1,0)
1 !Task(A3) v Post(A3,a1) v !Satisfied(A3,a2) v !Succeed(a2,a1,0)
1 !Task(A4) v Post(A4,a1) v !Satisfied(A4,a2) v !Succeed(a2,a1,0)
1 !Task(A5) v Post(A5,a1) v !Satisfied(A5,a2) v !Succeed(a2,a1,0)
1 !Task(A6) v Post(A6,a1) v !Satisfied(A6,a2) v !Succeed(a2,a1,0)

1 !Task(T1) v Post(T1,a1) v !Satisfied(T1,a2) v !Succeed(a2,a1,0)
1 !Task(T2) v Post(T2,a1) v !Satisfied(T2,a2) v !Succeed(a2,a1,0)
1 !Task(T3) v Post(T3,a1) v !Satisfied(T3,a2) v !Succeed(a2,a1,0)
1 !Task(T4) v Post(T4,a1) v !Satisfied(T4,a2) v !Succeed(a2,a1,0)
1 !Task(T5) v Post(T5,a1) v !Satisfied(T5,a2) v !Succeed(a2,a1,0)
1 !Task(T6) v Post(T6,a1) v !Satisfied(T6,a2) v !Succeed(a2,a1,0)

1 !Task(A1) v Pre(A1,a1) v !Satisfied(A1,a2) v !Succeed(a2,a1,2)
1 !Task(A2) v Pre(A2,a1) v !Satisfied(A2,a2) v !Succeed(a2,a1,2)

1 !Task(A3) v Pre(A3,a1) v !Satisfied(A3,a2) v !Succeed(a2,a1,2)
1 !Task(A4) v Pre(A4,a1) v !Satisfied(A4,a2) v !Succeed(a2,a1,2)
1 !Task(A5) v Pre(A5,a1) v !Satisfied(A5,a2) v !Succeed(a2,a1,2)
1 !Task(A6) v Pre(A6,a1) v !Satisfied(A6,a2) v !Succeed(a2,a1,2)

1 !Task(T1) v Pre(T1,a1) v !Satisfied(T1,a2) v !Succeed(a2,a1,2)
1 !Task(T2) v Pre(T2,a1) v !Satisfied(T2,a2) v !Succeed(a2,a1,2)
1 !Task(T3) v Pre(T3,a1) v !Satisfied(T3,a2) v !Succeed(a2,a1,2)
1 !Task(T4) v Pre(T4,a1) v !Satisfied(T4,a2) v !Succeed(a2,a1,2)
1 !Task(T5) v Pre(T5,a1) v !Satisfied(T5,a2) v !Succeed(a2,a1,2)
1 !Task(T6) v Pre(T6,a1) v !Satisfied(T6,a2) v !Succeed(a2,a1,2)

1 !Task(A1) v Occ(A1,a1) v !Satisfied(A1,a2) v !Succeed(a2,a1,1)
1 !Task(A2) v Occ(A2,a1) v !Satisfied(A2,a2) v !Succeed(a2,a1,1)
1 !Task(A3) v Occ(A3,a1) v !Satisfied(A3,a2) v !Succeed(a2,a1,1)
1 !Task(A4) v Occ(A4,a1) v !Satisfied(A4,a2) v !Succeed(a2,a1,1)
1 !Task(A5) v Occ(A5,a1) v !Satisfied(A5,a2) v !Succeed(a2,a1,1)
1 !Task(A6) v Occ(A6,a1) v !Satisfied(A6,a2) v !Succeed(a2,a1,1)

1 !Task(T1) v Occ(T1,a1) v !Satisfied(T1,a2) v !Succeed(a2,a1,1)
1 !Task(T2) v Occ(T2,a1) v !Satisfied(T2,a2) v !Succeed(a2,a1,1)
1 !Task(T3) v Occ(T3,a1) v !Satisfied(T3,a2) v !Succeed(a2,a1,1)
1 !Task(T4) v Occ(T4,a1) v !Satisfied(T4,a2) v !Succeed(a2,a1,1)
1 !Task(T5) v Occ(T5,a1) v !Satisfied(T5,a2) v !Succeed(a2,a1,1)
1 !Task(T6) v Occ(T6,a1) v !Satisfied(T6,a2) v !Succeed(a2,a1,1)

//G1 - AND Goal
50 OccurrenceGoal(G1,a1,a2) v !Satisfied(A1,a3) v !Satisfied(A2,a4) v !Satisfied(G2,a5) v !Satisfied(A3,a6) v !Satisfied(A4,a2) v !Succeed(a2,a1,10) v !Succeed(a3,a1,2) v !Succeed(a4,a1,4) v !Succeed(a5,a1,6) v !Succeed(a6,a1,8)
1 Satisfied(A1,a3) v !OccurrenceGoal(G1,a1,a2) v !Succeed(a2,a3,8) v !Succeed(a3,a1,2)
1 Satisfied(A2,a3) v !OccurrenceGoal(G1,a1,a2) v !Succeed(a2,a3,6) v !Succeed(a3,a1,4)
1 Satisfied(G2,a3) v !OccurrenceGoal(G1,a1,a2) v !Succeed(a2,a3,4) v !Succeed(a3,a1,6)
1 Satisfied(A3,a3) v !OccurrenceGoal(G1,a1,a2) v !Succeed(a2,a3,2) v !Succeed(a3,a1,8)
1 Satisfied(A4,a2) v !OccurrenceGoal(G1,a1,a2) v !Succeed(a2,a1,10)

0.5 !Satisfied(A1,a3) v OccurrenceGoal(G1,a1,a2) v !Succeed(a2,a3,8) v !Succeed(a3,a1,2)
0.5 !Satisfied(A2,a3) v OccurrenceGoal(G1,a1,a2) v !Succeed(a2,a3,6) v !Succeed(a3,a1,4)

0.5 !Satisfied(G2,a3) v OccurrenceGoal(G1,a1,a2) v !Succeed(a2,a3,4) v !Succeed(a3,a1,6)
0.5 !Satisfied(A3,a3) v OccurrenceGoal(G1,a1,a2) v !Succeed(a2,a3,2) v !Succeed(a3,a1,8)
0.5 !Satisfied(A4,a2) v OccurrenceGoal(G1,a1,a2) v !Succeed(a2,a1,10)

5 !Pre(G1,a1) v !Post(G1,a2) v !OccurrenceGoal(G1,a1,a2) v Satisfied(G1,a2) v !Succeed(a2,a1,10)
1 Pre(G1,a1) v !Satisfied(G1,a2) v !Succeed(a2,a1,10)
0.5 OccurrenceGoal(G1,a1,a2) v !Satisfied(G1,a2) v !Succeed(a2,a1,10)
1 !OccurrenceGoal(G1,a1,a2) v Satisfied(G1,a2) v !Succeed(a2,a1,10)
1 Post(G1,a1) v !Satisfied(G1,a2) v !Succeed(a2,a1,0)

// G2 - OR Goals
3 !Satisfied(A5,a2) v !Succeed(a2,a1,2) v OccurrenceGoal(G2,a1,a2)
3 !Satisfied(A6,a2) v !Succeed(a2,a1,2) v OccurrenceGoal(G2,a1,a2)
3 Satisfied(A5,a2) v Satisfied(A6,a2) v !Succeed(a2,a1,2) v !OccurrenceGoal(G2,a1,a2)

3 !Pre(G2,a1) v !Post(G2,a2) v !OccurrenceGoal(G2,a1,a2) v Satisfied(G2,a2) v !Succeed(a2,a1,2)
1 Pre(G2,a1) v !Satisfied(G2,a2) v !Succeed(a2,a1,2)
1 OccurrenceGoal(G2,a1,a2) v !Satisfied(G2,a2) v !Succeed(a2,a1,2)
1 Post(G2,a1) v !Satisfied(G2,a2) v !Succeed(a2,a1,0)

//AG1 - AND Anti-Goal
5 OccurrenceGoal(AG1,a1,a2) v !Satisfied(T1,a3) v !Satisfied(T2,a4) v !Satisfied(T3,a2) v !Succeed(a2,a1,6) v !Succeed(a3,a1,2) v !Succeed(a4,a1,4)
1 !OccurrenceGoal(AG1,a1,a2) v Satisfied(T1,a3) v !Succeed(a2,a3,4) v !Succeed(a3,a1,2)
1 !OccurrenceGoal(AG1,a1,a2) v Satisfied(T2,a3) v !Succeed(a2,a3,2) v !Succeed(a3,a1,4)
1 !OccurrenceGoal(AG1,a1,a2) v Satisfied(T3,a2) v !Succeed(a2,a1,6)

0.5 !Satisfied(AG1,a3) v OccurrenceGoal(T1,a1,a2) v !Succeed(a2,a3,4) v !Succeed(a3,a1,2)
0.5 !Satisfied(AG1,a3) v OccurrenceGoal(T2,a1,a2) v !Succeed(a2,a3,2) v !Succeed(a3,a1,4)
0.5 !Satisfied(AG1,a2) v OccurrenceGoal(T3,a1,a2) v !Succeed(a2,a1,6)

5 !Pre(AG1,a1) v !Post(AG1,a2) v !OccurrenceGoal(AG1,a1,a2) v Satisfied(AG1,a2) v !Succeed(a2,a1,6)
1 Pre(AG1,a1) v !Satisfied(AG1,a2) v !Succeed(a2,a1,6)
0.5 OccurrenceGoal(AG1,a1,a2) v !Satisfied(AG1,a2) v !Succeed(a2,a1,6)
1 !OccurrenceGoal(AG1,a1,a2) v Satisfied(AG1,a2) v !Succeed(a2,a1,6)

1 Post(AG1,a1) v !Satisfied(AG1,a2) v !Succeed(a2,a1,0)

//AG2 - AND Ant-Goal
5 OccurrenceGoal(AG2,a1,a2) v !Satisfied(T4,a3) v !Satisfied(T5,a4) v !Satisfied(T6,a2) v !Succeed(a2,a1,6) v !Succeed(a3,a1,2) v !Succeed(a4,a1,4)
1 !OccurrenceGoal(AG2,a1,a2) v Satisfied(T4,a3) v !Succeed(a2,a3,4) v !Succeed(a3,a1,2)
1 !OccurrenceGoal(AG2,a1,a2) v Satisfied(T5,a3) v !Succeed(a2,a3,2) v !Succeed(a3,a1,4)
1 !OccurrenceGoal(AG2,a1,a2) v Satisfied(T6,a2) v !Succeed(a2,a1,6)

0.5 !Satisfied(AG2,a3) v OccurrenceGoal(T4,a1,a2) v !Succeed(a2,a3,4) v !Succeed(a3,a1,2)
0.5 !Satisfied(AG2,a3) v OccurrenceGoal(T5,a1,a2) v !Succeed(a2,a3,2) v !Succeed(a3,a1,4)
0.5 !Satisfied(AG2,a2) v OccurrenceGoal(T6,a1,a2) v !Succeed(a2,a1,6)

5 !Pre(AG2,a1) v !Post(AG2,a2) v !OccurrenceGoal(A21,a1,a2) v Satisfied(AG,a2) v !Succeed(a2,a1,6)
1 Pre(AG2,a1) v !Satisfied(AG2,a2) v !Succeed(a2,a1,6)
0.5 OccurrenceGoal(AG2,a1,a2) v !Satisfied(AG2,a2) v !Succeed(a2,a1,6)
1 !OccurrenceGoal(AG2,a1,a2) v Satisfied(AG2,a2) v !Succeed(a2,a1,6)
1 Post(AG2,a1) v !Satisfied(AG2,a2) v !Succeed(a2,a1,0)

// Obstacle Rules
2 !Satisfied(AG1, t1) v !(t2 ¿ t1) v !Satisfied(A2, t2)
2 !Satisfied(AG1, t1) v !(t2 ¿ t1) v !Satisfied(A4, t2)
2 !Satisfied(AG2, t1) v !(t2 ¿ t1) v !Satisfied(A3, t2)
2 !Satisfied(AG2, t1) v !(t2 ¿ t1) v !Satisfied(A5, t2)

The set of observations indicating the failure of A6 and the occurrence of an SQL injection attack is shown below:

ChildAND(G1, G2) ChildAND(G1, A1) ChildAND(G1, A2) ChildAND(G1, A3) ChildAND(G1, A4)
ChildOR(G2, A5) ChildOR(G2, A6)
ChildAND(AG1, T1) ChildAND(AG1, T2) ChildAND(AG1, T3)
ChildAND(AG2, T4) ChildAND(AG2, T5) ChildAND(AG2, T6)
Task(A1) Task(A2) Task(A3) Task(A4) Task(A5) Task(A6)
Task(T1) Task(T2) Task(T3) Task(T4) Task(T5) Task(T6)
Pre(G1,1) Pre(A1,1) Occ(A1,2) Post(A1,3) Pre(A2,3) Occ(A2,4) Post(A2,5) Pre(G2,5) ?Pre(A5,5) ?Occ(A5,6) ?Post(A5,7) Pre(A6,5) ?Occ(A6,6) ?Post(A6,7) Post(G2,7)

?Pre(A3,7) ?Occ(A3,8) ?Post(A3,9) ?Pre(A4,9) ?Occ(A4,10) ?Post(A4,11)
?Post(G1,11) !Satisfied(G1,11)
Pre(AG1,1) ?Pre(T1,1) ?Occ(T1,2) ?Post(T1,3) ?Pre(T2,3) ?Occ(T2,4) ?Post(T2,5) ?Pre(T2,5)
?Occ(T2,6) ?Post(T2,7) ?Post(AG1,7)
Pre(AG2,1) Pre(T4,1) Occ(T4,2) Post(T4,3) Pre(T5,3) Occ(T5,4) Post(T5,5) Pre(T6,5)
Occ(T6,6) Post(T6,7) Post(AG2,7)

References

[1] mltool4j - google project hosting. http://code.google.com/p/mltool4j/.

[2] *Encyclopedia of philosophy.* Detroit [u.a.], 2. ed. edition, 2006. 10 Bde. - Auch als Online-Ausgabe auf dem Markt. - Frhere Aufl. in 8 Bdn. u.d.T.:: Encyclopedia of philosophy. Ed.: Paul Edwards, 1967.

[3] Concurrent dynamic analysis framework for multicore hardware. In *OOPSLA'09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 155–174, New York, NY, USA, 2009. ACM.

[4] Safaa O. Al-Mamory and Hongli Zhang. Intrusion detection alarms reduction using root cause analysis and clustering. *Comput. Commun.*, 32(2):419–430, 2009.

[5] Bogdan Alexe, Laura Chiticariu, Renee J. Miller, and Wang Chiew Tan. Muse: Mapping understanding and design by example. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancn, Mxico*, pages 10–19, 2008.

[6] James H. Andrews and Yingjun Zhang. Broad-spectrum studies of log file analysis. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 105–114, New York, NY, USA, 2000. ACM.

[7] John Bambenek and Angieszka Klus. *grep - Pocket Reference: the Basics for an Essential Unix Content-Location Utility.* O'Reilly, 2009.

[8] BEA. Wldf query language. http://download.oracle.com/docs/cd/E13222_01/wls/-docs90/wldf_configuring/appendix_query.html, 2009.

[9] R. Bjork. An example of object-oriented design: an atm simulation, department of computer science, gordon college, wenham, ma. http://www.math-cs.gordon.edu/courses/cs211/ATMExample, 2007.

[10] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, March 2003.

[11] S. Borghetti and A. Sgro. Dynamic software tracing. United States Patent Application, 20090241096, 2009.

[12] MIT Brandeis University, Brown University. Distributed stream processing engine. http://www.cs.brown.edu/research/borealis/public/, 2008.

[13] MIT Brandeis University, Brown University. The aurora project. http://www.cs.brown.edu/research/aurora/, 2011.

[14] Michael Cammert, Christoph Heinz, Jurgen Kramer, Alexander Markowetz, and Bernhard Seeger. Pipes: A multi-threaded publish-subscribe architecture for continuous queries over streaming data sources. Technical report, 2003.

[15] Mike Chen, Alice X. Zheng, Jim Lloyd, Michael I. Jordan, and Eric Brewe. Failure diagnosis using decision trees. *Autonomic Computing, International Conference on*, 0:36–43, 2004.

[16] Sung-bae Cho. Incorporating soft computing techniques into a probabilistic intrusion detection system. *IEEE Trans. on Systems, Man and Cybernetics-Part C:Applications and Reviews*, pages 154–160, 2002.

[17] M. Cinque, D. Cotroneo, and A. Pecchia. A logging approach for effective dependability evaluation of complex systems. In *Proceedings of the 2009 Second International Conference on Dependability*, pages 105–110, Washington, DC, USA, 2009. IEEE Computer Society.

[18] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. In *Science of Computer Programming*, pages 3–50, 1993.

[19] Anirban Dasgupta, Ravi Kumar, Prabhakar Raghavan, and Andrew Tomkins. Variable latent semantic indexing. In *KDD '05: Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 13–21, New York, NY, USA, 2005. ACM.

[20] C. J. Date. *SQL and Relational Theory: How to Write Accurate SQL Code*. O'Reilly Media, 1 edition, 2009.

[21] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[22] Johan de Kleer, Alan K. Mackworth, and Raymond Reiter. Characterizing diagnoses and systems. *Artif. Intell.*, 56(2-3):197–222, 1992.

[23] H. Debar, D. Curry, and B. Feinstein. The Intrusion Detection Message Exchange Format (IDMEF). RFC 4765 (Experimental), March 2007.

[24] S. Deerwester, S. T. Dumais, G. Furnas, Landauer T. K., and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41:391–407, 1990.

[25] Dorothy E. Denning. An intrusion-detection model. *IEEE Trans. Softw. Eng.*, 13(2):222–232, 1987.

[26] Xiaoning Ding, Hai Huang, Yaoping Ruan, Anees Shaikh, and Xiaodong Zhang. Automatic software fault diagnosis by exploiting application signatures. In *LISA'08: Proceedings of the 22nd conference on Large installation system administration conference*, pages 23–39, Berkeley, CA, USA, 2008. USENIX Association.

[27] Jean Dollimore, Tim Kindberg, and George Coulouris. *Distributed Systems: Concepts and Design (4th Edition) (International Computer Science Series)*. Addison Wesley, May 2005.

[28] Pedro Domingos. Real-world learning with markov logic networks. In Jean-Franois Boulicaut, Floriana Esposito, Fosca Giannotti, and Dino Pedreschi, editors, *Machine Learning: ECML 2004*, volume 3201 of *Lecture Notes in Computer Science*, pages 17–17. Springer Berlin / Heidelberg, 2004.

[29] C. Dukes. Logzilla, http://nms.gdd.net/index.php/logzilla.

[30] Economist. "data, data everywhere". a special report on managing information. http://www.economist.com/node/15557443, 2010.

[31] Hans-Erik Eriksson and Magnus Penker. Business modeling with uml: Business patterns at work, an introduction to the unified model language, and lessons and examples of practical business applications for software developers. John Wiley & Sons, 2000.

[32] EsperTech. Event stream intelligence. http://www.espertech.com/, 2011.

[33] M. S. Feather, S. Fickas, A. Van Lamsweerde, and C. Ponsard. Reconciling system requirements and runtime behavior. In *IWSSD '98: Proceedings of the 9th international workshop on Software specification and design*, page 50, Washington, DC, USA, 1998. IEEE Computer Society.

[34] Avigdor Gal, Ateret Anaby-Tavor, Alberto Trombetta, and Danilo Montesi. A framework for modeling and evaluating automatic semantic reconciliation. *The VLDB Journal*, 14(1):50–67, 2005.

[35] Paolo Giorgini, John Mylopoulos, Eleonora Nicchiarelli, and Roberto Sebastiani. Reasoning with goal models. pages 167–181. Springer, 2002.

[36] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, 2003.

[37] E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat-solver. *Design, Automation and Test in Europe Conference and Exhibition*, 0:0142, 2002.

[38] Eugene Goldberg and Yakov Novikov. Berkmin: A fast and robust sat-solver. *Discrete Appl. Math.*, 155(12):1549–1561, 2007.

[39] G. Golub and C. Reinsch. Singular value decomposition and least squares solutions. *Numerische Mathematik*, 14(5):403–420, April 1970.

[40] Jim Gray. Designing for 20tb disk drives and enterprise storage. research.microsoft.com/ Gray/talks/NSIC_HighEnd_Gray.ppt, 2000.

[41] The Open Group. "regular expressions". the single unix specification, version 2. http://pubs.opengroup.org/onlinepubs/007908799/xbd/re.htm, 1997.

[42] Dan Gunter, Brian L. Tierney, Aaron Brown, Martin Swany, John Bresnahan, and Jennifer M. Schopf. Log summarization and anomaly detection for troubleshooting distributed systems. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, GRID '07, pages 226–234, Washington, DC, USA, 2007. IEEE Computer Society.

[43] A. Guven, O. Bozkurt, and O. Kalpsz. Advanced information extraction with ngram based lsi. In *World Academy of Science, Engineering and Technology 17 2006*, 2006.

[44] Maggie Hamill and Katerina Goseva-Popstojanova. Common trends in software fault and failure data. *IEEE Trans. Softw. Eng.*, 35(4):484–496, 2009.

[45] Andreas Hanemann. A hybrid rule-based/case-based reasoning approach for service fault diagnosis. In *AINA '06: Proceedings of the 20th International Conference on Advanced Information Networking and Applications*, pages 734–740, Washington, DC, USA, 2006. IEEE Computer Society.

[46] Joe Hicklin, Cleve Moler, Peter Webb, Ronald F. Boisvert, Bruce Miller, Roldan Pozo, and Karin Remington. Jama. http://math.nist.gov/javanumerics/jama/, 2005.

[47] Mamoun Hirzalla, Jane Cleland-Huang, and Ali Arsanjani. Service-oriented computing — icsoc 2008 workshops. pages 41–52, Berlin, Heidelberg, 2009. Springer-Verlag.

[48] Thomas Hofmann. Probabilistic latent semantic indexing. In *SIGIR '99: Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 50–57, New York, NY, USA, 1999. ACM.

[49] Hai Huang, Raymond Jennings, III, Yaoping Ruan, Ramendra Sahoo, Sambit Sahu, and Anees Shaikh. Pda: a tool for automated problem determination. In *LISA'07: Proceedings of the 21st conference on Large Installation System Administration Conference*, pages 1–14, Berkeley, CA, USA, 2007. USENIX Association.

[50] IBM. Common base event. http://www.ibm.com/developerworks/library/specification/ws-cbe/, 2009.

[51] IBM. Eclipse modeling framework project (emf). http://www.eclipse.org/modeling/emf/,, 2009.

[52] IBM. Websphere software. http://www.ibm.com/websphere, 2009.

[53] Pankaj Jalote, Rajesh Munshi, and Todd Probsting. The when-who-how analysis of defects for improving the quality control process. *J. Syst. Softw.*, 80:584–589, April 2007.

[54] Weihang Jiang, Chongfeng Hu, Shankar Pasupathy, Arkady Kanevsky, Zhenmin Li, and Yuanyuan Zhou. Understanding customer problem troubleshooting from storage system logs. In *FAST '09: Proccedings of the 7th conference on File and storage technologies*, pages 43–56, Berkeley, CA, USA, 2009. USENIX Association.

[55] Weihang Jiang, Chongfeng Hu, Shankar Pasupathy, Arkady Kanevsky, Zhenmin Li, and Yuanyuan Zhou. Understanding customer problem troubleshooting from storage system logs. In *FAST '09: Proccedings of the 7th conference on File and storage technologies*, pages 43–56, Berkeley, CA, USA, 2009. USENIX Association.

[56] Weihang Jiang, Chongfeng Hu, Shankar Pasupathy, Arkady Kanevsky, Zhenmin Li, and Yuanyuan Zhou. Understanding customer problem troubleshooting from storage system logs. In *Proceedings of the 7th conference on File and storage technologies*, pages 43–56, San Diego, California, 2009. USENIX Association.

[57] Zhen Ming Jiang, Ahmed E. Hassan, Parminder Flora, and Gilbert Hamann. Abstracting execution logs to execution events for enterprise applications (short paper). In *QSIC '08: Proceedings of the 2008 The Eighth International Conference on Quality Software*, pages 181–186, Washington, DC, USA, 2008. IEEE Computer Society.

[58] Jeffrey O. Kephart. Research challenges of autonomic computing. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 15–22, New York, NY, USA, 2005. ACM.

[59] S. Kok, M. Sumner, M. Richardson, P. Singla, H. Poon, D. Lowd, and P. Domingos. The alchemy system for statistical relational ai. technical report, department of computer science and engineering, university of washington, seattle, wa. http://alchemy.cs.washington.edu, 2007.

[60] Jonathan Korba and Arthur C. Smith. Windows nt attacks for the evaluation of intrusion detection systems. Master's thesis, MIT, 2000.

[61] D. LaBerre. A satisfiability library for java, sat4j 1.6. http://www.sat4j.org/, 2009.

[62] Alexei Lapouchnian and John Mylopoulos. Modeling domain variability in requirements engineering with contexts. In *ER '09: Proceedings of the 28th International Conference on Conceptual Modeling*, pages 115–130, Berlin, Heidelberg, 2009. Springer-Verlag.

[63] Sangno Lee, Jeff Baker, Jaeki Song, and James C. Wetherbe. An empirical comparison of four text mining methods. In *HICSS '10: Proceedings of the 2010 43rd Hawaii International Conference on System Sciences*, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[64] Richard Lippmann, Joshua Haines, David Fried, Jonathan Korba, and Kumar Das. Analysis and results of the 1999 darpa off-line intrusion detection evaluation. In Herv Debar, Ludovic M, and S. Wu, editors, *Recent Advances in Intrusion Detection*, volume 1907 of *Lecture Notes in Computer Science*, pages 162–182. Springer Berlin / Heidelberg, 2000.

[65] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[66] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval.* Cambridge University Press, Cambridge, UK, 2008.

[67] Sheila A. Mcilraith. Explanatory diagnosis: Conjecturing actions to explain observations. In *In Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR&apos;98)*, pages 167–177, 1998.

[68] Masanori Miyazawa and Kosuke Nishimura. Scalable root cause analysis assisted by classified alarm information model based algorithm. In *Proceedings of the 7th International Conference on Network and Services Management*, CNSM '11, pages 371–374, Laxenburg, Austria, Austria, 2011. International Federation for Information Processing.

[69] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. pages 530–535, 2001.

[70] John Mylopoulos, Lawrence Chung, and Brian Nixon. Representing and using non-functional requirements: A process-oriented approach. *IEEE Transactions on Software Engineering*, 18:483–497, 1992.

[71] Sarah Nadi, Ric Holt, Ian Davis, and Serge Mankovskii. Draca: Decision support for root cause analysis and change impact analysis for cmdbs. University of Waterloo, CA Labs, Canada, 2009.

[72] M. Natu and A.S. Sethi. Using temporal correlation for fault localization in dynamically changing networks. *International Journal of Network Management*, 18:4, 2008.

[73] OASIS. Wsdm event format. http://docs.oasis-open.org/wsdm/wsdm-muws1-1.1-spec-os-01.pdf, 2007.

[74] Adam Oliner and Jon Stearley. What supercomputers say: A study of five system logs. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, pages 575–584, Washington, DC, USA, 2007. IEEE Computer Society.

[75] Object Management Group (OMG). Object constraint language, version 2.0. http://www.omg.org/spec/OCL/2.0/, 2006.

[76] S. Pankanti, Q. Fan, Y. Zhai, R. Bobbitt, A. Yanagawa, S. Miyazawa, R. Kjeldsen, and A. Hampapur. Multi-media compliance: A practical paradigm for managing business integrity. In *Multimedia and Expo, 2009. ICME 2009. IEEE International Conference on*, pages 1562 –1563, 28 2009-july 3 2009.

[77] Denys Poshyvanyk, Andrian Marcus, Vaclav Rajlich, Yann-Gael Gueheneuc, and Giuliano Antoniol. Combining probabilistic ranking and latent semantic indexing for feature identification. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 137–148, Washington, DC, USA, 2006. IEEE Computer Society.

[78] R Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987.

[79] Raymond Reiter. The frame problem in situation the calculus: a simple solution (sometimes) and a completeness result for goal regression. pages 359–380, 1991.

[80] Matthew Richardson and Pedro Domingos. Markov logic networks. *Mach. Learn.*, 62:107–136, February 2006.

[81] L. Ryan. *Efficient algorithms for clause-learning SAT solvers.* M. eng. thesis, Simon Fraser University, 2004.

[82] Reza Sadoddin and Ali Ghorbani. Alert correlation survey: framework and techniques. In *PST '06: Proceedings of the 2006 International Conference on Privacy, Security and Trust*, pages 1–10, New York, NY, USA, 2006. ACM.

[83] F. Salfner, S. Tschirpke, and M. Malek. Comprehensive Logfiles for Autonomic Systems. pages 211b+. IEEE Computer Society, April 2004.

[84] Emanuel Santos, Jaelson Castro, Juan Sánchez, and Oscar Pastor. A goal-oriented approach for variability in bpmn. In *WER*, 2010.

[85] Sawmill. Log analysis tool, http://www.sawmill.net.

[86] Shiuh-Pyng Shieh and Virgil D. Gligor. On a pattern-oriented model for intrusion detection. *IEEE Trans. on Knowl. and Data Eng.*, 9(4):661–667, 1997.

[87] Parag Singla and Pedro Domingos. Discriminative training of markov logic networks. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 868–873. AAAI Press / The MIT Press, 2005.

[88] Alex Skaletsky, Tevi Devor, Nadav Chachmon, Robert Cohn, Kim Hazelwood, Vladimir Vladimirov, and Moshe Bach. Dynamic program analysis of microsoft windows applications. In *International Symposium on Performance Analysis of Software and Systems*, ISPASS, pages 2–12, White Plains, NY, April 2010.

[89] SLCT. Simple log clustering tool (slct). http://www.estpak.ee/ risto/slct/, 2009.

[90] soapUI. soapui 2.0.2. http://www.soapui.org/, 2009.

[91] Vitor Estevao Silva Souza and John Mylopoulos. Monitoring and diagnosing malicious attacks with autonomic software. In Alberto H. F. Laender, Silvana Castano, Umeshwar Dayal, Fabio Casati, and Jos Palazzo Moreira de Oliveira, editors, *ER*, volume 5829 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2009.

[92] Splunk. Splunk, http://www.splunk.com.

[93] M. Steinder and A. S. Sethi. Probabilistic fault diagnosis in communication systems through incremental hypothesis updating. *Comput. Netw.*, 45(4):537–562, 2004.

[94] Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera, and Vishaka Nanayakkara. Siddhi: a second look at complex event processing architectures. In *Proceedings of the 2011 ACM workshop on Gateway computing environments*, GCE '11, pages 43–50, New York, NY, USA, 2011. ACM.

[95] Son Dinh Tran and Larry S. Davis. Event modeling and recognition using markov logic networks. In *European Conference on Computer Vision*, pages 610–623, 2008.

[96] Cornell University. Cornell database group - cayuga. http://www.cs.cornell.edu/bigreddata/cayuga/, 2011.

[97] Stanford University. Stream: Stanford stream data manager. http://infolab.stanford.edu/stream/ora/, 2011.

[98] Risto Vaarandi. A data clustering algorithm for mining patterns from event logs. In *IEEE IPOM03 Proceedings*, pages 119–126, 2003.

[99] A. Van Lamsweerde, R. Darimont, and P. Massonet. Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learnt. In *RE '95: Proceedings of the Second IEEE International Symposium on Requirements Engineering*, page 194, Washington, DC, USA, 1995. IEEE Computer Society.

[100] A. van Lamsweerde and E. Letier. Handling obstacles in goal-oriented requirements engineering. *Software Engineering, IEEE Transactions on*, 26(10):978–1005, 2000.

[101] Axel van Lamsweerde. Elaborating security requirements by construction of intentional anti-models. *Software Engineering, International Conference on*, 0:148–157, 2004.

[102] Axel van Lamsweerde. Requirements engineering: from craft to discipline. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 238–249, New York, NY, USA, 2008. ACM.

[103] Yiqiao Wang, Sheila A. McIlraith, Yijun Yu, and John Mylopoulos. An automated approach to monitoring and diagnosing requirements. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 293–302, New York, NY, USA, 2007. ACM.

[104] Yiqiao Wang, Sheila A. Mcilraith, Yijun Yu, and John Mylopoulos. Monitoring and diagnosing software requirements. *Automated Software Engg.*, 16(1):3–35, 2009.

[105] C. Wilke. Java code generation for dresden ocl2 for eclipse. Master's thesis, Technische Universitt Dresden, February 2009.

[106] Lihua Wu, JianPing Feng, and Yunfen Luo. A personalized intelligent web retrieval system based on the knowledge-base concept and latent semantic indexing model. *Software Engineering Research, Management and Applications, ACIS International Conference on*, 0:45–50, 2009.

[107] Liudong Xing and Wendai Wang. Probabilistic common-cause failures analysis. In *Proceedings of the 2008 Annual Reliability and Maintainability Symposium*, pages 354–358, Washington, DC, USA, 2008. IEEE Computer Society.

[108] Yijun Yu, Alexei Lapouchnian, Sotirios Liaskos, John Mylopoulos, and Julio Leite. From goals to high-variability software design. pages 1–16, 2008.

[109] Yijun Yu, Yiqiao Wang, John Mylopoulos, Sotirios Liaskos, Alexei Lapouchnian, and Julio C. Leite. Reverse engineering goal models from legacy code. In *RE '05: Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE'05)*, pages 363–372, Washington, DC, USA, 2005. IEEE Computer Society.

[110] Chun Yuan, Ni Lao, Ji-Rong Wen, Jiwei Li, Zheng Zhang, Yi-Min Wang, and Wei-Ying Ma. Automated known problem diagnosis with event traces. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 375–388, New York, NY, USA, 2006. ACM.

[111] Hamzeh Zawawy, Kostas Kontogiannis, and John Mylopoulos. Log filtering and interpretation for root cause analysis. In *ICSM '10: Proceedings of the 26th IEEE International Conference on Software Maintenance*, 2010.

[112] Hamzeh Zawawy, Kostas Kontogiannis, John Mylopoulos, and Serge Mankovski. Towards a requirements-driven framework for detecting malicious behavior against software systems. In *CASCON '11: Center for Advanced Studies on Collaborative Research*, 2011.

[113] Hamzeh Zawawy, Kostas Kontogiannis, John Mylopoulos, and Serge Mankovski. Requirements-driven root cause analysis using markov logic networks. In *CAiSE '12: 24th International Conference on Advanced Information Systems Engineering*, 2012.

[114] Hamzeh Zawawy, John Mylopoulos, and Serge Mankovski. Requirements driven framework for root cause analysis in soa environments. In *MESOA '10: Proceedings of the 4th International Workshop on Maintenance and Evolution of Service-Oriented Systems*, 2010.

[115] X. Zhou. A goal-oriented instrumentation approach for monitoring requirements. Master's thesis, University of Toronto, 2008.