Policy Driven Software Monitoring

by

Yat Fai Alfred Wong

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

Waterloo, Ontario, Canada, 2007

Abstract

Software monitoring and logging is one of the most important tools a software engineer has when faced with the challenge of auditing or analysing a software system. However, the difficulty in effectively monitoring a system, managing its logs and cross referencing them with source code makes software re-engineering a rigorous and complex task. This thesis aims to address this issue by providing a framework that enables pattern matching between a software log and an event pattern expression that is based on a monitoring policy. The framework consists of parsers and annotators that facilitates transformation of a monitoring policy into a Petri Net as well as source code annotation for gathering data through logged events. It further expands upon this work by proposing an adaptive logging framework that will greatly improve the quality of log management by autonomically adjusting the amount of information logged based on the application's operational environment. Finally, a prototype system of the policy driven monitoring framework is implemented and tested with applications of different scales as a proof of concept for the proposed framework.

Acknowledgments

To God be all glory and honour. He is my source of strength that made the completion of this thesis possible.

I would like to thank Kostas Kontogiannis, whose advice and guidance was instrumental to this thesis; and my readers, Ladan Tahvildari and Paulo Alencar, for their valuable comments and suggestions.

Words cannot express my gratitude to my immediate family — my parents and my brother, and my brothers and sisters at North Toronto Chinese Alliance Church, whose love and prayer has provided an immense amount of encourangement throughout this process. Special thanks to Terry Ha, who was there for me in my darkest hours — I know I'm who I am today because I knew you; and to Jon Chiu, whose hospitality and friendship has provided a sanctuary from the weathers of life — because I knew you, I have been changed for good.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the last decade we have witnessed the evolution of large industrial software systems from a single server application to component-based collaborating applications deployed over both local and wide-area networks. The distributed nature of these component-based systems provides significant benefits with respect to software quality, development time and deployment costs. However, such systems also pose significant challenges with respect to system monitoring, logging, auditing, and diagnosing. Specifically, these collaborating applications often consist of components that were developed independently, and thus utilize heterogeneous event logging and diagnostic techniques as well as diverse log management and system monitoring policies. In this context, an interesting challenge is to devise software engineering techniques to amalgamate and integrate the heterogeneous logging and monitoring processes so that such software systems may still be analyzed, audited and diagnosed in an effective and efficient manner.

1.1 Motivation

A log is a record of the events, operations and other associated data that occurred within a software system. Logs can contain a variety of information, such as data related to system performance, security, resource usage and even personally identifiable information. While traditionally logs have been primarily used for debugging and troubleshooting, nowadays logs are one of the most important tools a software engineer uses to optimize system performance, recording user's actions, and providing information for investigating and preventing security breaches and other malicious activities. These logs can be generated by many different sources, including:

- performance monitoring software such as statistical profilers, task managers, CPU and memory usage analyzers;
- security software such as firewalls, antivirus softwares, worm detection and phishing protection systems;
- operating systems on servers, workstations and networking equipment;
- other applications that need to keep track of user data and history, such as an Internet browser.

Because of the widespread deployment of distributed applications on networked servers, workstations and other devices, the size of logs generated for a system has multiplied many-fold. Coupled with the many sources and purposes that logs serve as mentioned above, the volume of logs for a software system has increased greatly. This calls for the need for a system monitoring framework which:

1. analyzes a software system and gathers the appropriate logging data in sufficient detail pertaining to a particular problem or investigation;
2. models appropriately these logged data; and
3. interprets the logged data according to some policies as guidelines to determine whether a threat or a deviation from the system's functional or non-functional requirements has been observed.

1.2 Problem Description

While many organizations recognize the need of logging in their software systems, many fail to manage, analyze and make use of the information logged. One of the biggest causes of this may be that log analysis — the study of log entries to identify events of interest — has become a non-trivial task for the software engineer due to the sheer volume of logs as well as the lack of software tools to effectively analyze the logs for events of interest.

Currently, most software monitoring tools use the concept of logging levels, which is typically a parameterized input to the software system, to dictate the amount of information the software system records in its logs. At arbitrarily defined levels, the amount of information logged may easily be too little or too specific for the issue under investigation. Furthermore, even with the appropriate amount of information in the logs, correlating the sequence of logged events to their origins in the source code proves to be a challenging problem.

Traditionally, most logs have not been analyzed in a dynamic or real-time manner. As such, many software engineers regard log analysis as a reactive rather than a proactive solution to problems. They see log analysis as something that has to be done after a problem is identified via other means rather than using logging and system monitoring to identify ongoing activity and look out for signs of imminent problems. Without the proper tools and framework, a software engineer that wishes to perform log analysis will have to manually parse through source code and log files to correlate the cause and effect of the events logged. In large distributed software systems, this task becomes gargantuan and next to impossible for a human to perform. Thus, the lack of proper tools and framework to monitor a system and analyze its logs significantly reduces the role of logging in investigating and identifying issues within a software system.

1.3 Thesis Contribution

This thesis aims to address the aforementioned problem by proposing a framework that consists of tools and techniques to facilitate the analysis, design and deployment of logging and monitoring processes for legacy component-based applications. The framework proposed identifies spots in source code to be logged, defines and formulates monitoring policies, and captures events from the source code to match against the monitoring policies. The framework assumes that these applications have well-defined system requirements, architecture and logging requirements. System requirements define the purpose of the application and the functionalities it supports. Logging requirements define the purposes for logging (e.g., security, diagnosis, auditing) as well as other constraints. The major contributions of this thesis are:

1. A conceptual model and language for specifying logging and monitoring requirements — logging and monitoring requirements can be specified in a policy language which conforms to a specific domain model presented in UML and is transformable to a Petri Net for application and analysis;

2. A mechanism for parsing, analysing and annotating source code for monitoring — the source code is parsed to instantiate a domain model that parallels that of the policy domain model. Lines of code that are of potential monitoring interest are annotated for comparison;

3. A pattern matching framework that locates the logging and monitoring requirements within the annotated source code — the pattern matcher compares the policy with the annotated source code and applies concepts from dynamic programming to identify the best match of the source code to the monitoring policy;

4. A plug-and-play architecture for insertion and modification of monitoring probes that takes advantage of Artificial Intelligence theories to facilitate adaptive logging — the aforementioned process is applied to an architecture where monitoring probes can be defined and revised on-the-fly, and using a blackboard design pattern, the monitoring framework can dynamically adjust

and adapt to these new changes and requests.

1.4 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 provides a survey of existing research in areas related to the work presented in this thesis, which includes a discussion on system monitoring, source code analysis and instrumentation, pattern matching and adaptive logging. Chapter 3 presents a high-level overview of the proposed policy-driven software monitoring system. Chapter 4 provides a conceptual model and a language for specifying the monitoring requirements. Chapter 5 describes the parsing of the monitoring policy and the tree representation it generates. Chapter 6 discusses the parsing and analysis of source code in order to annotate it for instrumentation. Chapter 7 presents a pattern matching algorithm to locate areas in the source code that matches the monitoring requirements. Chapter 8 presents an architecture for adaptive logging that allows monitoring probes to be dynamically inserted. Chapter 9 describes the experiments performed with a prototype system that implements the presented framework and the feasibility studies performed. Finally, Chapter 10 presents the conclusions and discusses avenues for future work.

Chapter 2

Related Work

The area of software logging and monitoring has been investigated over the years by numerous researchers. However, the constant change of software platforms, operating environments, software system deployment topologies and component communication protocols require continuous work in this area. In light of such work, this chapter presents existing research and papers related to this dissertation. This chapter is divided into three subsections. Section 2.1 will present an overview of existing monitoring systems found in existing literature. Section 2.2 will discuss logging tools and frameworks currently in use in the industry. Section 2.3 will survey well known Artificial Intelligence techniques in pattern matching and dynamic programming that can be applied to software monitoring.

2.1 Monitoring

System monitoring is the observation of specific activities or events that occur in an information system as specified by a predefined set of rules or policies. Related work in the area of monitoring information systems can be classified into five major areas, namely monitoring frameworks, monitoring to facilitate program understanding, monitoring through code instrumentation, monitoring for quality assurance and dynamic monitoring.

In the area of monitoring frameworks, Mansouri-Samani and Sloman presents in [1] a flexible technique that can be used to process and disseminate events for many different applications such as networked, distributed and event based applications. The technique models composite events as rules. A dissemination technique allows for the appropriate monitoring of components to be distributed close to where events are generated. This allows for network traffic reduction caused by event generation in large systems. In [2], Sosnowski and Poleszak propose a collection of monitoring systems that occur at three levels of abstraction, namely the operational, low architecture and high architecture level. These are integrated with available hardware and software mechanisms for commercial-off-the-shelf (COTS) based systems. Depending on monitoring objectives, the appropriate level of abstraction for logging is selected. Gwadera, Atallah and Szpankowski describes in [3] a statistical technique to minimize the number of false alarms when a pattern of suspicious events is detected. The technique aims to quantify the probability of such a pattern occurring in a large log database within a specific window size. Given this probability, the technique determines an alarm threshold so that the probability of false alarms is bounded by a given constant. Finally, Bertino, Ferrari and Guerrini proposes in [4] an event-based temporal object model that keeps track of selected values within the history of a data object. The paper also proposes techniques for querying a database with incomplete temporal information. The objective of this work is to support software monitoring applications so that events causing changes to particular objects are recorded. Moreover, the proposed technique stores the history of data items only when certain specified events occur.

In the area of monitoring for program understanding, Goldman describes a monitoring program called Smiley in [5] that can selectively log calls to functions exported from shared libraries of the operating system as they are made by an application. By selecting library calls to monitor, software engineers may obtain an understanding of an application's implementation. This logging technique can also be used as an aid in comprehending the implementation of COTS software. Sefika, Sane and Campbell discusses in [6] a monitoring approach that integrates logic-based static analysis and dynamic visualizations such as multiple code views and perspectives to validate a software system. This approach determines a system's design-implementation congruence by studying coding guidelines, design patterns, connectors and subjective design principles like high cohesion and low coupling. Lastly, Ulrich et al. in [7] presents a tracing-based analysis technology to test the trustworthiness of the requirements of a distributed system. The technique is based on a model checker that compares a model of the event logs collected during a system run with the specification of the system. The result of such a comparison is a visualization of fulfilments and violations of intended system behavior.

In the area of monitoring through code instrumentation, Kishon, Hudak and Consel introduces in [8] the concept of monitoring semantics, a model of program exeuction that captures monitoring activity as those found in debuggers, profilers, tracers, daemons etc. Monitoring semantics forms a practical basis for instrumenting code into programs by embedding monitoring actions into the program. Moore, McGregor and Breen compares in [9] two system monitoring methods: passive network monitoring and kernel instrumentation. The comparison points out that kernel instrumentation has the potential to give an exact record of what occured in the system kernel. Passive network monitoring, however, is being used as a replacement for kernel instrumentation in some situations, despite the fact that it performs poorly as a direct replacement. This is partially due to the non-intrusive nature of passive network monitoring, which is particularly valuable when the source code for the kernel that needs to be monitoried is unavailable. Robinson presents in [10] a general framework that monitors the requirements of software as it executes. The framework relies on instrumented code such as assertions and model checking to inform the monitor of the software system's execution.

In the area of monitoring for quality assurance, Lin and Siewiorek offers in [11] a proactive approach to failure-prediction by a heuristic called the Dispersion Frame Technique (DFT). The DFT is based on the shape of the inter-arrival time function of intermittent errors observed from actual error logs. The DFT aims to minimize error-log entry points required by statistical methods for failure prediction in distributed applications. With regards to monitoring deployed software to allow for a prompt reaction to failures, Bowring, Orso and Harrold presents the technique of software tomography in [12], which splits monitoring tasks across many instances of the software by collecting partial information from users through light-weight instrumentation and then merging these data to get the complete monitoring information. Peng, Wu and Sun discusses in [13] an online monitoring method that uses a distributed event-driven hybrid monitor system with a synchronous clock system. The resulting prototype online monitoring program OM proves to be useful in efficiently testing and debugging parallel and distributed applications and systems.

In the area of dynamic monitoring, Robinson presents in [14] an implementation of rule-based monitors derived from system requirements. Robinson's ReqMon research project uses the KAOS language and the Dwyer temporal patterns to define the monitoring requirements. ReqMon's monitoring is dynamic in the sense that a requirement's satisfaction status is dynamically updated after an event occurs, but is not adapative in the sense that monitors do not change dynamically on which level and what to monitor according to received events.

2.2 Logging

Logging is the creation of a record of events occuring within an organization's systems and networks [15]. There are a number of existing logging tools currently in use in the industry. Sun Microsystems develop a Solaris-based tracing facility known as DTrace [16]. DTrace provides an infrastructure for dynamically logging a system's behaviour which allows software engineers and system administartors to observe, debug and optimize system behaviour. To facilitate dynamic tracing, DTrace uses the D language for DTrace to specify the variables for tracing. DTrace programs work by setting up and enabling probes; whenever a probe fires because the condition for the probe is met, the action associated with the probe in the DTrace program is executed. Similarly, the Unicenter tool [17] by Computer Associates supports tracking and monitoring system configurations and events. Unicenter consists of an Event Management System (EMS), an Advanced Event Correlation (AEC), and an Alert Management System (AMS). These components allow for monitoring and consolidating event activity from a variety of sources, grouping associated events for further processing, initiating actions based on specific policies and managing the highest severity events. Microsoft also produces its own logging framework known as Microsoft Operations Manager (MOM) [18]. It monitors multiple servers in an enterprise environment by placing MOM agents on the computer to be monitored. The MOM agent collects events from several sources on that computer, such as the Windows Event Log, and forwards them to the MOM management server. The server places those events into the MOM database for further processing by using rules to identify issues that affect the operations of the whole system. These rules can trigger a variety of actions, such as sending notification to a human via e-mail or a pager message, generating a network support ticket, or even trigger a workflow that resolves the error causing the alerting event in the first place.

There are also a number of open source logging frameworks available. The Business Intelligence Reporting Tool (BIRT) [19] is an Eclipse-based open source reporting system, intended primarily for web applications based on Java and J2EE. BIRT has two main components: a report designer based on Eclipse, and a runtime component that can be added to an application server. An interesting application of BIRT is the creation of monitoring reports by obtaining data from different logging systems applied to a given software application. Another open source logging framework also related to Eclipse is the Eclipse Test and Performance Tools Platform (TPTP) Project [20]. TPTP contains a number of tracing, monitoring and profiling tools, including the Log and Trace Analyzer (LTA) [21] which is an Eclipse plug-in editing tool that can build symptom catalogs for an application. The information specified in the symptom catalogs is used to assist problem analysts in debugging and resolving problems occurring during deployment and operation of an application. This package also includes an Eclipse plug-in editing tool for Generic Log Adapter parser rules, which operates on both Linux and Windows. IBM also offers a Linux kernel debugging tool called Kprobes [22] that logs debugging information such as the kernel stack trace, kernel data structures and registers by dynamically inserting breakpoint instructions at a given address in the running kernel. Execution of the instruction results in a breakpoint fault and Kprobes hooks on to the breakpoint handler to collect the appropriate information. Another open source logging framework is Systemtap [23]. Systemtap allows software engineers to create scripts to name events such as entering or exiting a subroutine or a timer expiring etc. and give them handlers that specify the actions to be taken when the event occurs. It works by translating the scripts to C and compiling it into a kernel module. Loading the module activates all the probes by hooking them to the kernel, and thus when an event occurs the compiled handlers will run. The handlers normally extracts contextual data from the event, store them into other variables, or outputs them to a file or to the screen.

Finally, with respect to event languages for logging, IBM has contributed the

Common Base Event [24] specification language in this facet for defining mechanisms for managing events in business enterprise applications. The purpose of the Common Base Event is to establish a new standard to facilitate the effective intercommunication among disparate enterprise components through log record management, problem determination and repair within an enterprise. An interesting application of the Common Base Event specification is seen in the communication of self-healing actions within IBMs autonomic computing model [25].

Several logging frameworks and protocols have also been proposed from academic research as well. Barga, Chen and Lomet proposed in [26] an enhanced logging framework prototype for component-based software systems. Rather than force logging all events from intercomponent method calls and returns, the enhanced logging prototype only logs information required to remove nondeterminism, and only force log when an event commits a component's state to the other parts of the system. Kongmunvattana and Tzeng proposed in [27] the lazy logging protocol for software distributed shared memory (SDSM) systems. The lazy logging protocol works by minimizing failure-free overhead by logging only the data necessary for recovery, hence effectively reducing the number of meassges logged and the amount of log data. They further expanded on this concept in [28] by introducing a similar logging protocol for adaptive software distributed shared memory (ADSM) systems known as adaptive logging.

In the area of logging for system verification, Andrews and Zhang reports in [29] the application of log file analysis techniques to check test results for a broad range of testing tasks, such as unit-level and system-level testing, testing against requirements for critical and non-critical systems, and the use of log file analysis techniques with other conventional testing methods. To perform these different testing tasks, a logging framework that can adhere to a well-defined logging policy which details what the software under test should log under what precise conditions must exist. This can be facilitated either through standard code review and inspection procedures or through automatic code instrumentation as this dissertation shall discuss.

In a related area of log file analysis, Vaarandi presented in [30] a Simple Logfile Clustering Tool (SLCT) that uses a data clustering algorithm for log data sets to determine frequent patterns from log files, build log file profiles and identify anomalous lines from log files. By comparing against an expected profile of operation, faults in a system can be detected as they would show up as anomalies in the log files. Stearley shows in [31] a novel apporach of using the bioinformatic-inspired Teiresias algorithm to automatically classify system log messages. The occruence statistics and results are found to be comparable to those from Vaarandi's SLCT mentioned above.

Finally, there are a number of other applications of logging and log analysis. A common scenario is the study of logs to determine user patterns for a website or a web service. For example, Lin and Hadingham use log analysis to track the frequently traversed spots in a web site [32]. Similarly, Koch, Ardö and Golub presents in [33] a case study of using log analysis techniques to improve a distributed web service Renardus by analyzing usage patterns and activities as well as other parameters such as entry points, exit points, referrals and points of failures within a session. Logging and log analysis is also important in studies of user behaviour and human computer interaction. Maeda, Sugiyama and Mase presents a log analysis of human behaviour on interactive amusement media in [34]. Lastly, log analysis techniques can also be applied in the analysis of audit trails for access logs. Rostad and Edsberg present in [35] a study of access control requirements for healthcare systems. Since access logs are often huge in size and the use of uninformative reasons for access is frequent, it is difficult to manually audit the log for misuse. With the aid of logging analysis techniques and a proper logging framework, access logs can be better analyzed to reduce exceptional accesses and minimize misuse. By studying logs of the user's manipulations and the protocols used during their manipulations of the interactive media, better user interfaces can be designed.

2.3 Artificial Intelligence Techniques

To facilitate code instrumentation, the source code must be first parsed and transformed into a model of code representation, such as an Abstract Syntax Tree (AST). To pinpoint the location where the code should be instrumented, the AST needs to be matched against a pre-defined policy which can also be represented as a tree. In the area of tree representation, Knuth presents in his classical work [36] several representations of trees that is helpful in tree comparison. Tree representation techniques such as sequential memory, preorder sequential, family-order sequential, level-order sequential and postorder with degrees are all useful representations for tree comparison, manipulation and transformation.

Single keyword matching refers to finding all occurrences of a specific pattern in a given input text string [37]. There are four major approaches to solving the single keyword matching problem. Davies and Bowsher presents in [38] a comparison of these algorithms in terms of theoretical and experimental time and deteremined there is no overall 'best' algorithm. Their results for each of the algorithms are briefly discussed below.

The simplest technique for single keyword matching is the brute-force or naive algorithm, which scans the text and checks the pattern against the input text string character by character. If the length of the pattern is m and the lenght of the input text string is n, then the brute-force method has a runtime of O(mn). Knuth, Morris and Pratt presents an improvement of this techinque in [39]. Their method, commonly known as the KMP algorithm, scans the text progressively and uses knowledge of the already compared characters to determine the next position to compare the pattern against. It does this by constructing a next function table that determines the number of characters to slide the pattern to the right in case of a mismatch during the pattern matching process. The KMP algorithm has a theoretical behaviour of O(n+m) while the next function table occupies O(m) space. The fastest pattern matching algorithm for single keyword matching is proposed by Boyer and Moore [40] and is known as the BM algorithm. The BM algorithm compares the pattern against the text in the reverse direction from right to left. If a mismatch occurs, a pattern shift is computed. To speed up the process, the pattern is preprocessed to produce the shift tables. While the theoretical behaviour of the BM algorithm is identical to that of the KMP algorithm, experimental results show that the BM algorithm is practically faster than the KMP algorithm. Lastly, Karp and Rabin presents the KR algorithm in [41]. The KR algorithm takes up memory space by treating each possible m-character sections of the text as a keyword in a hash table, computing the hash function of it and checking whether it equals the hash function of the pattern. While the KR algorithm theoretically behaves linearly, the substantially higher running time of hash function computation makes it unfeasible for pattern matching in strings.

In the area of matching sets of keywords, Aho and Corasick proposes the well known AC algorithm in [42] by constructing a finite state pattern matching machine from a given set of keywords and then using the finite state machine to process the text string in a single pass. Aoe, Yamamoto and Shimada further improves this algorithm in [37] by detecting and removing redundant operations from the AC finite state machine.

While the techniques discussed above deal with exact string pattern matching problems, an extension to these problems is the approximate string matching problem. The approximate string matching problem arises in a number of areas in artificial intelligence research like speech recognition and image processing such as the work listed in [43]. It is also seen in areas like molecular biology, spelling correction, file comparison etc. Hall and Dowling presents a survey of this problem and its solutions in [44]. The simplest approach to solving the exact string pattern matching problem as discovered independently by many researchers [37] is to use a simple dynamic programming algorithm to find the edit distance, which is the smallest number of editing transformations required to change one file to match another. There are two common edit distance measures, namely the Hamming distance and the Levenshtein distance. The Hamming distance measure the number of positions with mismatching characters in two strings of equal lenght. The Levenshtein distance, on the other hand, measures the minimum number of character changes, insertions and deletions required to transform one string to another, where both strings do not have to be of identical length.

Applying these techniques discussed to broader fields, string pattern matching algorithms can be used in multidimensional matching problems from pictures and graphs to protein structures and nucleic acids. Of particular interest in this area is the tree pattern matching problem, as it is a recurring problem in many programming tasks such as interpreter and compiler design, code optimization and symbolic computation. Hoffmann and O'Donnell in [45] reviews the relationship between string and tree pattern matching techniques and evalutes a number of tree matching techniques for linear patterns. Just as the brute force method in string pattern matching takes $O(mn)$ time, the naive tree pattern matching approach, which involves traversing a tree in preorder and recursively comparing against the pattern at each node visited, also executes at the order of $O(mn)$ for a pattern tree of size m and an input text tree of size n. For linear patterns, there are two general pattern matching techniques, namely bottom-up and top-down. In the bottom-up approach, the subject tree is traversed and all patterns and parts of the patterns are matched at each point. A table is built to determine whether a matching node's children also match with the matching pattern node's children. Once the tables are precomputed, the matching time is $O(n)$; but the preprocessing time and memory space required to create these tables are intensive. In contrast to the bottom-up approach, the top-down approach treats each root-leaf path of the pattern as a string representation and then preprocesses these strings using the AC approach to determine all instances of path strings in the input subject tree. The preprocessing time of this top-down approach requires only O(m) time. Wuu in [46] extend this top-down approach by further reducing this tree matching problem to a string matching problem and applies the KMP string-matching algorithm to tree patterns.

A number of optimizations have also be performed on these tree pattern matching algorithms. Kosaraju in [47] explores the possibility of finding a better algorithm than the naive approach to the classic open problem on tree pattern matching, where don't care symobls and linear string max-min convlution are also treated. Dubiner, Galil and Magen improves upon Kosaraju's algorithm in [48] and designed an even faster simple tree pattern matching algorithm. Valiente in [49] presents the Berztiss Tree Pattern Matching algorithm and establishes its correctness and efficiency. Ejnioui and Ranganathan explores two systolic algorithms using VLSI approaches in [50] which shows a significant improvement over previous implementations.

Just as the exact string matching problems can be applied to tree pattern matching, the approximate string matching techniques can be used to apply to approximate tree pattern matching problems. Wang et. al demonstrates an implemented solution to this approximate tree pattern matching problem with the Approximate-Tree-By-Example (ATBE) system in [51]. Given a pattern, the ATBE system allows users to retrieve approximately matching trees to the pattern from a database, using a two-dimensional query language.

A noteworthy application of these tree pattern matching techniques is demonstrated by Aho, Ganapathi and Tjiang in [52]. They present a tree-mainpulation language known as *twig* to construct efficient code generators such as lexical analyzer generators and parser generators. The key idea of twig is that it combines the top-down tree pattern matching approach as presented by Hoffmann in [45] with dynamic programming to transform a tree-translation scheme into a code generator. The dynamic programming algorithm is a simplication of Aho and Johnson's optimal code-generation algorithm [53] used in several compilers. On a similar line of research, Chen, Lai and Shang demonstrate in [54] how a simple tree pattern matching algorithm can be used to optimize compile time. Inherent in the compiler is a tree pattern matching problem that matches the intermediate code with the tree-rewriting rules of the instruction description which describes the target system architecture to generate the corresponding assembly code. By using a hashing function, the tree pattern matching problem becomes a simple number comparison, which significantly speeds up the compilation time.

Furthermore, Ramesh and Ramakrishnan discusses in [55] methods of matching non-linear tree patterns. If a pattern contains variables and each variable occurs only once, then the pattern is linear; otherwise the pattern is said to be non-linear. Similarly to the top-down approach, a finite state machine is constructed for matching, but the automaton may contain cycles and extra pre-processing is required.

The application of pattern matching to system monitoring is not new. A number of intrusion detection systems (IDS) have been proposed over the past few years which identify patterns of security threats and vulnerabilities and monitors system activites to recognize when a system is under attack or when the vulnerabilities are being exploited. Denning in [56] presents a model of a real-time intrusion detection expert system that identifies security violations by monitoring a system's audit records for abnormalities in system usage. The model includes profiles and statistical models of normal system usage so that anomalous behaviour can be picked out. Similarly, Moore, Ellilson and Linger illustrates an approach for documenting system attacks in a structured and reusable format in [57]. Security analysts can use this information to identify and prevent common recurring attack patterns in information systems. For example, Shieh and Gilgor proposes in [58] a pattern-oriented intrusion detection model that tracks data flow through a software system to detect intrusions and security violations. Their model is reliant on a predefined representation of various types of intrusion patterns that [57] and [56] may have identified. Dharmapurikar and Lockwood suggests in [59] a hardware implementable pattern matching algorithm for content filtering applications such as a network IDS. Their approach is particularly suitable for implementation on network equipment such as routers. Lastly, Xu and Nygard in [60] proposes a threat-driven approach of software design that models security threats and functions as Petri Nets. The threat mitigations are modeled by Petri Net-based aspects to facilitate the crosscutting nature of security solutions. Their approach is applicable in securing a software system by reducing design-level vulnearabilities in a software system.

Chapter 3

Architectural Overview

This section discusses how the proposed policy driven software monitoring system can be broken down into different components and demonstrates the interactions between them. From a high level perspective, the system can be broken down into four major components, namely the Source Code Parser, Source Code Annotator, Monitoring Policy Parser and the Pattern Matcher. The block diagram presented in Figure 3.1 outlines the interaction of these different components and artifacts.

The use case diagram presented in Figure 3.2 illustrate at a high level what actions an user can perform and how they relate to each other in satisfying the user's end goal of locating source code statements that correspond to a sequence of events defined in a monitoring policy.

The policy-driven software monitoring takes two objects as inputs - the source code file and the monitoring policy file. For the purpose of the prototype presented

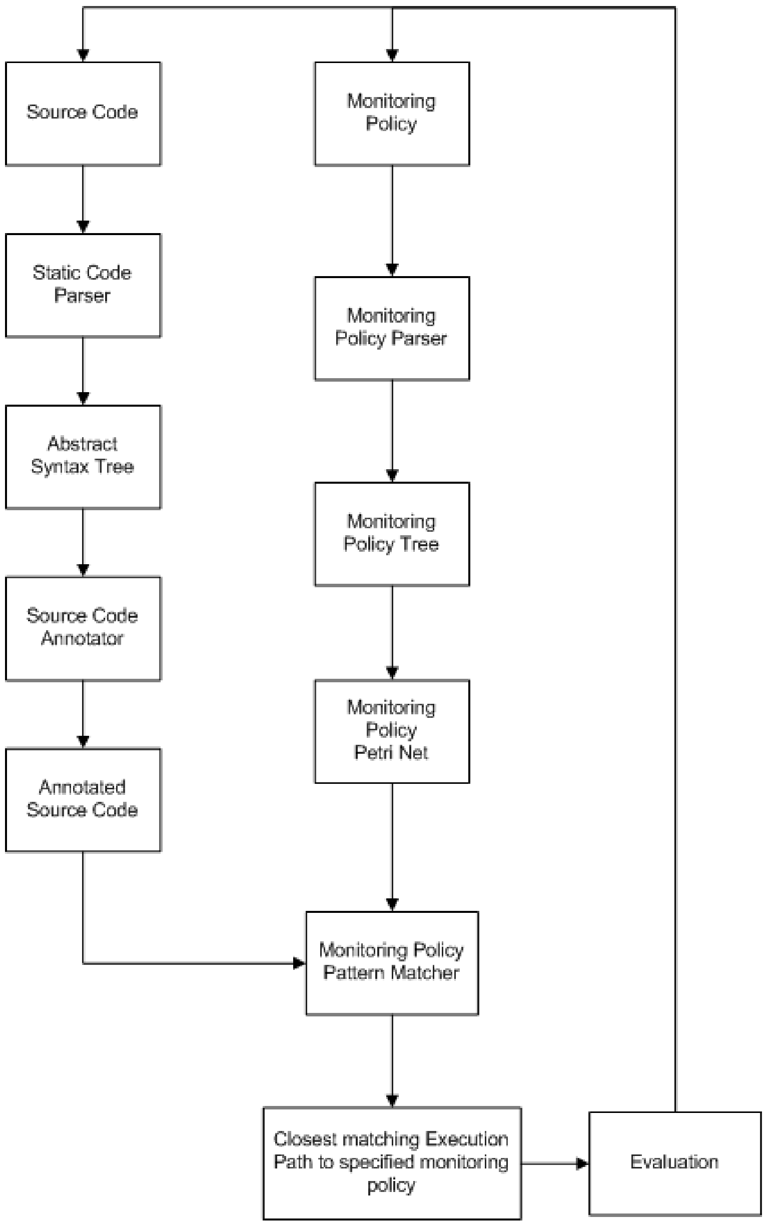


Figure 3.1: Block Diagram Outlining the Workflow of the Policy Driven Software Monitoring System

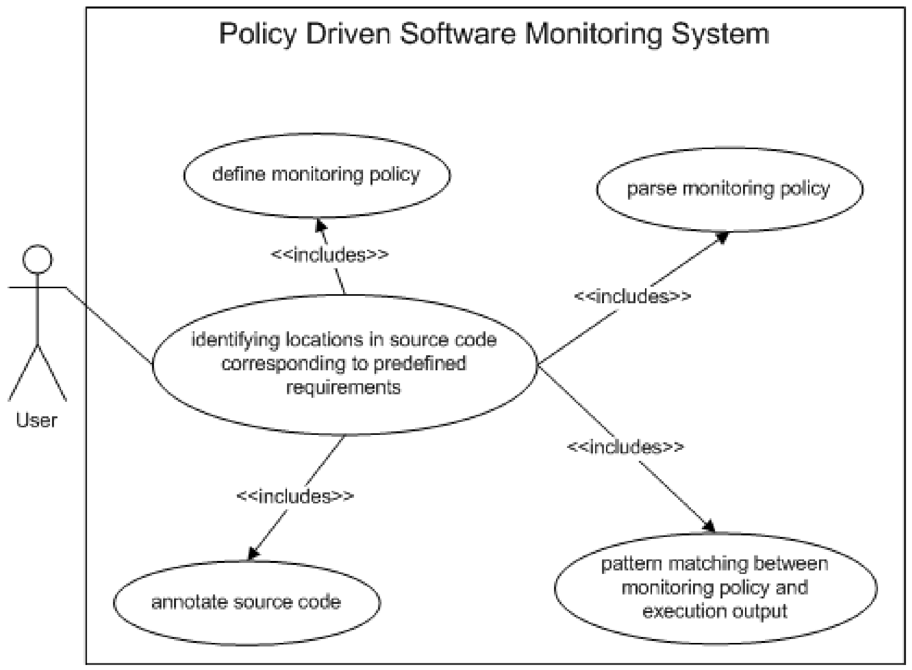

Policy Driven Software Monitoring System
define monitoring policy
parse monitoring policy
<<includes>>
<<includes>>
User
identifying locations in source code corresponding to predefined requirements
<<includes>>
<<includes>>
annotate source code
pattern matching between monitoring policy and execution output

Figure 3.2: Use Case Diagram of the Policy Driven Software Monitoring System

in this paper, the source code file is written in C; however, the architecture presented in this paper is general and can be easily modified for different programming languages. The monitoring policy file provides the specification on what to monitor in the software system according to the monitoring policy domain model presented in Chapter 4. These two input objects go through two separate processing paths, produce different artifacts along the way, and merge at the end to generate the final output.

The source code file first goes through the static code parser which generates a modified abstract syntax tree as an intermediate output. This modified abstract syntax tree is passed to the code annotator where C source code is annotated directly into the source code to produce an annotated source code file. This annotated source code file is compiled and executed and its output is used to determine the dynamic runtime path of the system. The static code parser and code annotator are described in detail in Chapter 6.

On the other processing path, the monitoring policy goes through the monitoring policy parser, which generates a monitoring policy tree. This tree representation is then transformed into a Petri Net representation as discussed in Chapter 5, which is used to match against the output of the annotated source code to determine the location of lines of code that matches the sequence of events specified by the monitoring policy. The workings of the pattern matcher are discussed in Chapter 7.

Figure 3.3 presents the activity diagram that pictorially describes these two processing paths.

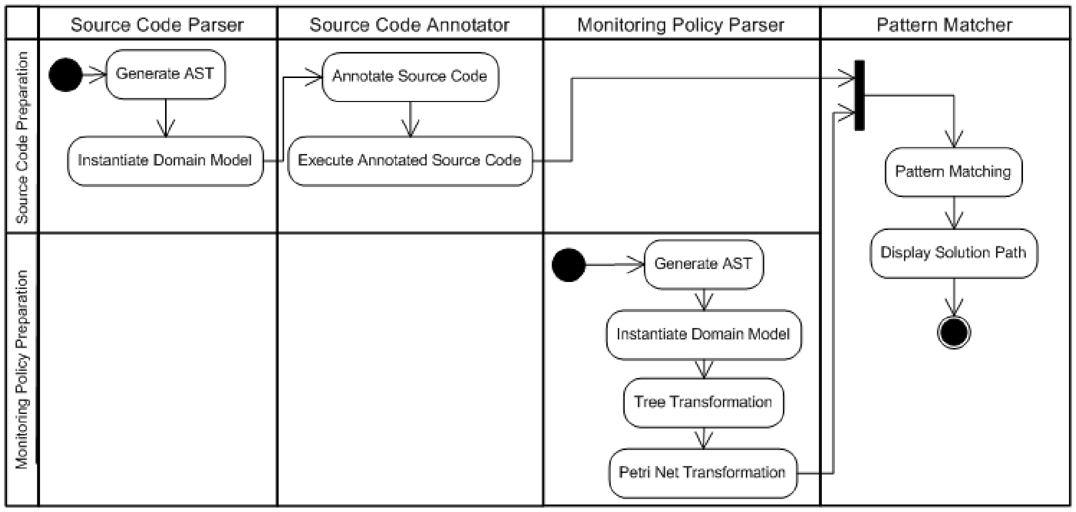


Figure 3.3: Activity Diagram for the Policy Driven Software Monitoring System

Figure 3.4 presents the sequence diagram that demonstrates how the artifacts generated by the different components interact with each other.

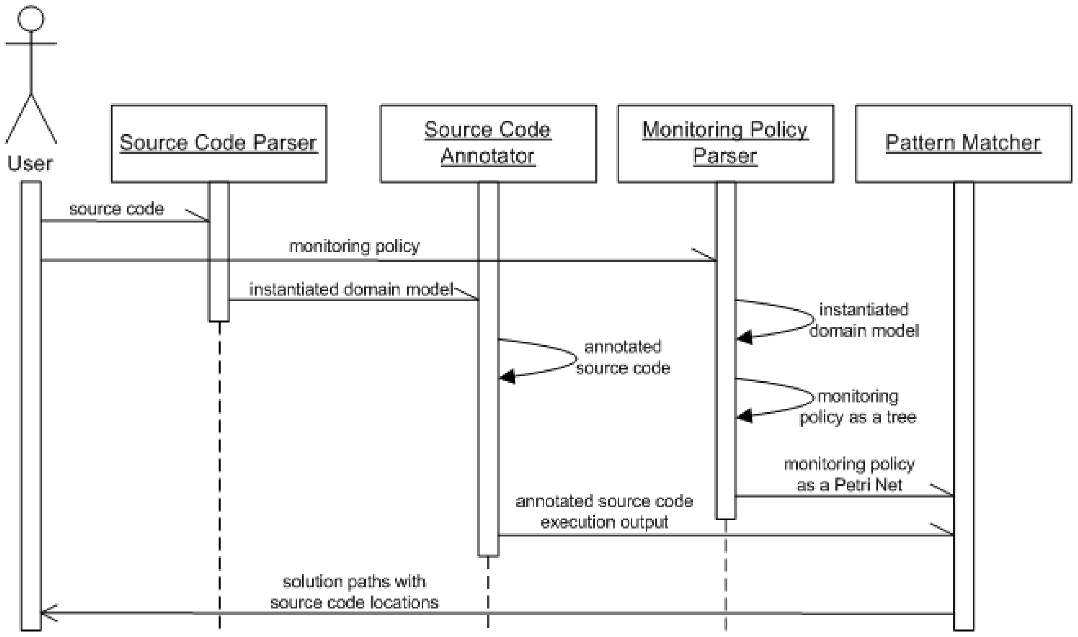

User
Source Code Parser
Source Code Annotator
Monitoring Policy Parser
Pattern Matcher
source code
monitoring policy
instantiated domain model
instantiated domain model
annotated source code
monitoring policy as a tree
monitoring policy as a Petri Net
annotated source code execution output
solution paths with source code locations

Figure 3.4: Sequence Diagram for the Policy Driven Software Monitoring System

Chapter 4

Monitoring Policy Event Language

The first step in the policy-driven software monitoring process is to define the policy that dictates what needs to be monitored. This chapter discusses the structure and syntax of the language used to create a policy for the proposed framework by presenting the semantics, the domain model and the grammar for the policy language. This chapter will conclude with a few example policies demonstrating the language in application.

4.1 Monitoring Policy Model Definitions

This section will provide formal definitions to some important terms used throughout this chapter with regards to the monitoring policy event language. Definitions will be provided for the terms atomic events, composite events, monitoring policy, log, logging system and monitoring system for the purpose of this thesis.

DEFINITION 1: **Atomic Event**

An atomic event represents the occurrence of an action, such as an user initiating a login procedure, or a state, such as a TCP/IP socket is open on port 80.

DEFINITION 2: **Composite Event**

A composite event consists of several atomic events that constitute a certain behavioural or logical pattern. For example, three unsuccessful logins from the same computer within one minute may constitute a composite event *SuspiciousLogin*.

DEFINITION 3: **Monitoring Policy**

A monitoring policy is a meaningful sequence of events that signifies a predefined behaviour worthy of monitoring, such as a performance requirement or a security breach. For example, a monitoring policy on a brute force attack of an user authentication system may be compromised of repeated *SuspiciousLogin* events due to incorrect passwords of increasing lengths.

DEFINITION 4: **Log**

A log is a record of an event that is generated by the instrumentation of a probe in a logging system.

Definition 5: **Logging System**

A logging system comprises of probes that are instrumented when triggered by an event. The instrumentation of a probe generates log records.

Definition 6: **Monitoring System**

A monitoring system gathers data through a logging system and interprets the log data to recognize significant atomic or composite events for diagnostic and auditing purposes.

4.2 Monitoring Policy Domain Model

The monitoring policy domain model specifies the concepts and relations between different objects found in a policy. A monitoring policy can be simplified by breaking it down into two major components – patterns and operators. A *pattern* is a common code structure that is of monitoring interest due to security or peformance reasons. Some common code constructs include:

- File Operations – Code that opens, closes, reads from or writes to a file. In C, these operations are characterized by the keywords `fopen, fclose, fscanf` and `fprintf`.
- Function Call – Code that invokes another function in the same file or code module with or without parameters.
- Memory Allocation – Code that performs dynamic memory allocation from the heap by explicitly specifying the size of the block of memory required. In

C, these operations are characterized by the keywords `malloc, calloc` and `realloc`.

- Memory Deallocation – Code that explicitly releases a block of memory previously allocated. In C, this operation is characterized by the keyword `free`.
- Socket Opening – Code that opens a socket for communication (eg. a TCP/IP socket). In C, this operation is characterized by the keyword `socket`.
- Socket Closing – Code that closes a socket previously opened for communication. In C, this operation is characterized by the keywords `close` and `shutdown`.
- Socket Data Operations – Code that reads or writes data to a socket previously opened for communication. In C, this operation is characterized by the keywords `read` and `write`.
- Wildcard – Any of the above common code structures.

On the other hand, an *operator* defines the temporal and logical ordering of two patterns. There are three operators available:

- Sequence – this operator defines a temporal ordering of two patterns, specifically the first pattern occurs before the second pattern. This operator is represented by the ";" symbol.
- Choice – this operator defines a logical choice between two patterns and is typically stated as the boolean OR operator in literature. This operator is represented by the "+" symbol.

- Concurrency – this operator defines a temporal ordering of two patterns, in that the two patterns should occur simultaenously as parallel threads or processes. This operator is represented by the "|" symbol.

The combination of two patterns and one operator forms a simple *atomic expression*. Multiple atomic expressions and pattern can be further combined to form *aggregate expressions*. The formal structure and relationships of these different elements of the policy domain model is presented in Figure 4.1 as a formal UML class diagram. Table 4.1 summarizes the attributes of each object presented in Figure 4.1.

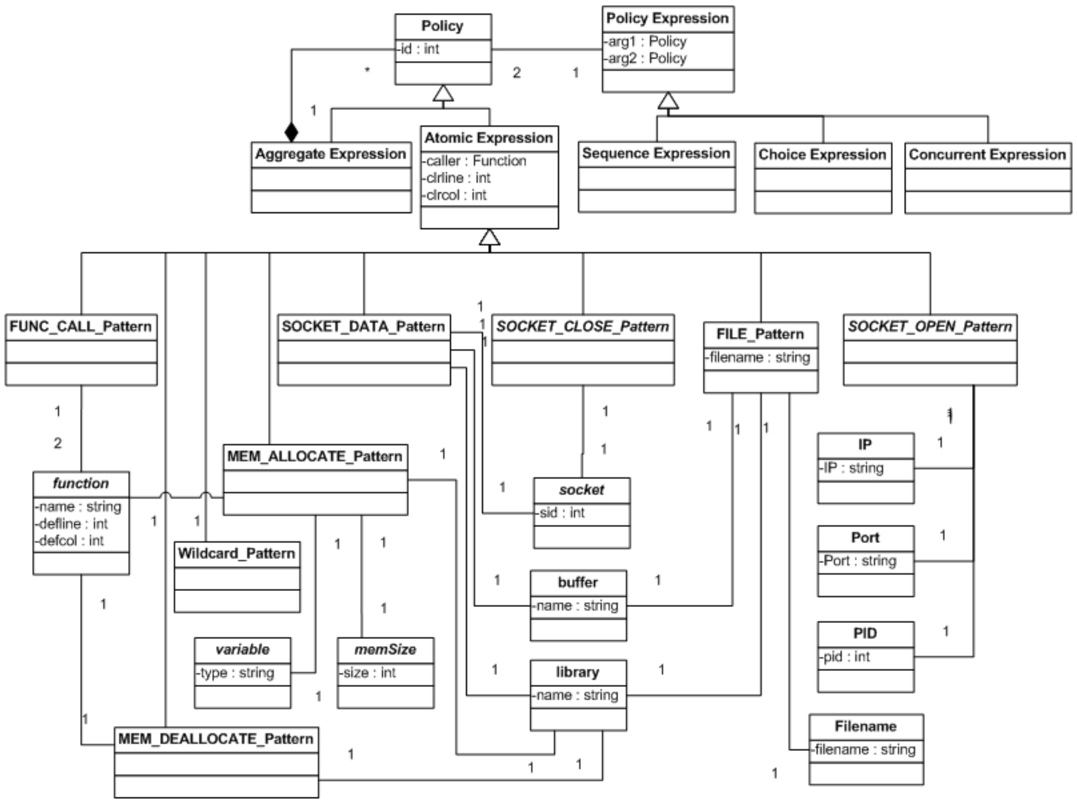


Figure 4.1: Monitoring Policy Domain Model

Table 4.1: Object Table for Policy Domain Model

OBJECT	**Policy**
PARENT	None
CHILDREN	Aggregate Expression, Atomic Expression
ATTRIBUTES	id : integer
OBJECT	**PolicyExpression**
PARENT	None
CHILDREN	Sequence Expression, Concurrent Expression, Choice Expression
ATTRIBUTES	id : integer policy1 : Policy policy2 : Policy
OBJECT	**SequenceExpression**
PARENT	PolicyExpression
CHILDREN	None
ATTRIBUTES	None
OBJECT	**ConcurrentExpression**
PARENT	PolicyExpression
CHILDREN	None
ATTRIBUTES	None

OBJECT	**ChoiceExpression**
PARENT	PolicyExpression
CHILDREN	None
ATTRIBUTES	None

OBJECT	**AggregateExpression**
PARENT	Policy
CHILDREN	None
ATTRIBUTES	subexpression : Policy

OBJECT	**AtomicExpression**
PARENT	Policy
CHILDREN	FUNC_CALL_Pattern, FILE_Pattern, SOCKET_OPEN_Pattern, SOCKET_CLOSE_Pattern, SOCKET_DATA_Pattern, MEM_ALLOCATE_Pattern, MEM_DEALLOCATE_Pattern, Wildcard_Pattern
ATTRIBUTES	caller : Function clrline : int clrcol : int

OBJECT	**FUNC_CALL_Pattern**
PARENT	Atomic Expression
CHILDREN	None
ATTRIBUTES	callee : Function

Object	**FILE_Pattern**
Parent	Atomic Expression
Children	None
Attributes	l : Library f : Filename b : Buffer

Object	**SOCKET_OPEN_Pattern**
Parent	Atomic Expression
Children	None
Attributes	source : PID targetIP : IP targetPort : Port

Object	**SOCKET_CLOSE_Pattern**
Parent	Atomic Expression
Children	None
Attributes	s : Socket

Object	**SOCKET_DATA_Pattern**
Parent	Atomic Expression
Children	None
Attributes	l : Library s : Socket b : Buffer

Object	**MEM_ALLOCATE_Pattern**
Parent	Atomic Expression
Children	None
Attributes	l : Library size : MemSize t : Variable

Object	**MEM_DEALLOCATE_Pattern**
Parent	Atomic Expression
Children	None
Attributes	l : Library

Object	**Wildcard_Pattern**
Parent	Atomic Expression
Children	None
Attributes	None

Object	**Buffer**
Parent	None
Children	None
Attributes	name : string

Object	**Filename**
Parent	None
Children	None
Attributes	filename : string

OBJECT	**Function**
PARENT	None
CHILDREN	None
ATTRIBUTES	name : string defline : int defcol : int

OBJECT	**IP**
PARENT	None
CHILDREN	None
ATTRIBUTES	IP : String

OBJECT	**Library**
PARENT	None
CHILDREN	None
ATTRIBUTES	policy1 : Policy policy2 : Policy

OBJECT	**Memsize**
PARENT	None
CHILDREN	None
ATTRIBUTES	policy1 : Policy policy2 : Policy

OBJECT	**PID**
PARENT	None
CHILDREN	None
ATTRIBUTES	pid : int

OBJECT	**Port**
PARENT	None
CHILDREN	None
ATTRIBUTES	port : int

OBJECT	**Socket**
PARENT	None
CHILDREN	None
ATTRIBUTES	sid : int

OBJECT	**Variable**
PARENT	None
CHILDREN	None
ATTRIBUTES	type : string

4.3 Language Grammar

While the previous section detailed the constructs for a monitoring policy, this section will discuss the grammar and syntax of the monitoring policy. Table 4.2 presents the grammar for the monitoring policy language in EBNF (Extended Backus-Naur Form). It is noteworthy that this grammar is recursive and hence allows for nested structures. Because of this possibility of nested structures, the associativity of each operator will be explicitly defined using parentheses.

Table 4.2: EBNF Grammar for Monitoring Policy Language

Policy	= (PolicyExpression)+
PolicyExpression	= AtomicExpression | AggregateExpression
AggregateExpression	= SequenceExpression | ChoiceExpression | ConcurrentExpression
SequenceExpression	= "(" Policy ";" Policy ")"
ConcurrentExpression	= "(" Policy "|" Policy ")"
ChoiceExpression	= "(" Policy "+" Policy ")"
AtomicExpression	= Pattern "(" Parameter_List ")" | Pattern | *null*
Pattern	= "FUNC_CALL_PATTERN" | "SOCKET_OPEN_PATTERN" | "SOCKET_CLOSE_PATTERN" | "SOCKET_DATA_PATTERN" | "FILE_PATTERN" | "MEM_ALLOCATE_PATTERN" | "MEM_DEALLOCATE_PATTERN" | "*"
Parameter_List	= Parameter ("," Parameter)+
Parameter	= *identifier* | *null*

4.4 Language Semantics

This section discusses in detail the meaning of the patterns that the monitoring policy language defines. The semantics of each pattern will be described in prose as well as in mathematical notation.

4.4.1 Atomic Policies

Atomic policies are the fundamental building blocks of a monitoring policy and is represented by a simple atomic expression. There are eight atomic policies defined for the monitoring policy language that represents different code constructs, namely function call patterns, file operation patterns, memory allocate and deallocate patterns, socket open, close and data transferral patterns as well as a wildcard pattern.

A `FUNC_CALL_PATTERN` defines the code construct where one function calls another function in its function definition. The function that invokes the function call is known as the *caller*, whereas the function that is being called is known as the *callee*. The `FUNC_CALL_PATTERN` takes two parameters, namely the name of the caller function followed by the name of the callee function. Mathematically, this is shown as:

$$\begin{aligned}\delta(\text{FUNC\_CALL\_PATTERN}(\alpha,\, \beta)) \longrightarrow\ & \{\ \varepsilon = \text{``}\alpha \text{ calls } \beta\text{'' s.t. } \varepsilon \text{ is an event,} \\ & \exists\ (\alpha\text{:function, } \beta\text{:function}), \\ & \alpha \text{ is calling } \beta \text{ in } \alpha\text{'s function definition}\}\end{aligned}$$

A `FILE_PATTERN` defines the code construct where a function performs an open, read, write or close operation on a file. This pattern takes four parameters in the following order: the caller function name, the library used to invoke the file operation (eg. `fopen, fclose, fscanf` etc.), the name of the file that is being operated on, and the buffer or file handle used to make reference to the file. Mathematically, this is represented as:

δ(FILE_PATTERN(α, β, γ, ζ)) $\longrightarrow$ { ε = "α calls β on γ using ζ" s.t. ε is an event, $\exists$ (α:function, β:library, γ:filename, ζ:buffer), α is invoking the β library γ on file via buffer ζ }

A `MEM_ALLOCATE_PATTERN` defines the code construct that dynamically allocates memory from the heap to be used by the program. This pattern takes four parameters in the following order: the caller function name, the library used to invoke the dynamic memory allocation (eg. `malloc, calloc, realloc` etc.), the size of the memory allocated, and the variable type the block of memory is casted to for dereferencing. Mathematically, this is represented as:

δ(MEM_ALLOCATE_PATTERN(α, β, γ, ζ)) $\longrightarrow$ { ε = "α calls β to allocate γ bytes of memory casted as ζ" s.t. ε is an event, $\exists$ (α:function, β:library, γ:memsize, ζ:variable), α is invoking the β library to allocate memory of size γ casted as variable type ζ }

A `MEM_DEALLOCATE_PATTERN` defines the code construct that deallocates memory on the heap previously allocated. This pattern takes two parameters, namely the caller function name followed by the library used to deallocate memory allocation (eg. `free` etc.). Mathematically, this is represented as:

$$\delta(\text{MEM\_DEALLOCATE\_PATTERN}(\alpha,\ \beta)) \longrightarrow \begin{array}{l} \{\ \varepsilon = \text{“}\alpha \text{ calls } \beta \text{ to deallocate} \\ \text{memory” s.t. } \varepsilon \text{ is an event,} \\ \exists\ (\alpha\text{:function, } \beta\text{:library),} \\ \alpha \text{ invokes the } \beta \text{ library to} \\ \text{deallocate memory previously} \\ \text{allocated.}\} \end{array}$$

A `SOCKET_OPEN_PATTERN` defines the code construct that opens a socket for communication such as a TCP/IP socket. This pattern takes four parameters in the following order: the caller function name, the Process ID of the source process opening the socket, the receiver's IP, and lastly the receiver's port with which the socket is communicating with. Mathematically, this is represented as:

$$\delta(\text{SOCKET\_OPEN\_PATTERN}(\alpha,\ \beta,\ \gamma,\ \zeta)) \longrightarrow \begin{array}{l} \{\ \varepsilon = \text{“}\alpha \text{ on } \beta \text{ opens a socket} \\ \text{to } \gamma\text{:}\zeta\text{” s.t. } \varepsilon \text{ is an event,} \\ \exists\ (\alpha\text{:function, } \beta\text{:PID, } \gamma\text{:IP,} \\ \zeta\text{:port), } \alpha \text{ running on process} \\ \text{with PID } \beta \text{ opens a socket} \\ \text{to } \gamma \text{ on port } \zeta\ \} \end{array}$$

A `SOCKET_DATA_PATTERN` defines the code construct that uses an opened communication socket to transmit and receive data. This pattern takes four parameters in the following order: the caller function name, the library used for communication, the socket used for communication and the buffer used for reading and writing data. Mathematically, this is represented as:

$$\delta(\text{SOCKET\_DATA\_PATTERN}(\alpha, \beta, \gamma, \zeta)) \longrightarrow \{\ \varepsilon = \text{“}\alpha \text{ calls } \beta \text{ to communicate over } \zeta \text{ using } \gamma\text{” s.t. } \varepsilon \text{ is an event, } \exists\ (\alpha\text{:function, } \beta\text{:library, } \gamma\text{:socket, } \zeta\text{:buffer), } \alpha \text{ invokes library } \beta \text{ and uses buffer } \zeta \text{ to communicate over socket } \gamma\ \}$$

A SOCKET_CLOSE_PATTERN defines the code construct that closes a previously opened communication socket. Aside from the caller function name, this pattern takes the socket to be closed as the second parameter. Mathematically, this is represented as:

$$\delta(\text{SOCKET\_CLOSE\_PATTERN}(\alpha, \beta)) \longrightarrow \{\ \varepsilon = \text{“}\alpha \text{ closes } \beta\text{” s.t. } \varepsilon \text{ is an event, } \exists\ (\alpha\text{:function, } \beta\text{:socket), } \alpha \text{ closes socket } \beta\ \}$$

A WILDCARD_PATTERN is represented by the asterisk (*) symbol. The wildcard pattern does not take any parameters and can represent any of the aforementioned patterns.

4.4.2 Composite Policies

While the atomic policies define important event patterns for the purposes of monitoring, it is often necessary to combine multiple atomic expressions to form a meaningful and useful monitoring policy. Composite policies introduce operators which can combine multiple atomic policies introduced above in a logical or temporal fahsion.

A *Sequence* operator between two atomic policies defines a temporal ordering of the two atomic policy. Mathematically, this is shown as:

$$\begin{aligned}\delta(\alpha\ ;\ \beta \longrightarrow) \quad & \{\ \varepsilon = \text{``}\alpha\ \beta\text{''} \text{ s.t. } \varepsilon \text{ is an event,} \\ & \exists\ (\alpha\text{:atomic policy, } \beta\text{:atomic policy),} \\ & \alpha \text{ temporally occurs before } \beta\ \}\end{aligned}$$

A *Choice* operator between two atomic policies defines a logical OR choice between the two atomic policy. Mathematically, this is shown as:

$$\begin{aligned}\delta(\alpha + \beta \longrightarrow) \quad & \{\ \varepsilon = \text{``}\alpha\text{''} \text{ or } \varepsilon = \text{``}\beta\text{''} \text{ s.t. } \varepsilon \text{ is an event,} \\ & \exists\ (\alpha\text{:atomic policy, } \beta\text{:atomic policy),} \\ & \text{either } \alpha \text{ or } \beta \text{ occurs }\}\end{aligned}$$

A *Concurrent* operator between two atomic policies defines a temporal parallelism between the two atomic policy. Mathematically, this is shown as:

$$\begin{aligned}\delta(\alpha \mid \beta \longrightarrow) \quad & \{\ \varepsilon = \text{``}\alpha\ \beta\text{''} \text{ or } \varepsilon = \text{``}\beta\ \alpha\text{''} \text{ s.t. } \varepsilon \text{ is an event,} \\ & \exists\ (\alpha\text{:atomic policy, } \beta\text{:atomic policy),} \\ & \alpha \text{ occurs simultaneously with } \beta\ \}\end{aligned}$$

4.5 Examples of Monitoring Policies

This section will present some examples of the monitoring policies described in this chapter. The simplest type of policy is the single term policy, which consists of only one simple pattern, such as those listed below.

- `func_call_pattern(foo, bar)` - Function foo calls Function bar
- `file_pattern(foo, fopen, input.txt, fp1)` - Function foo opens input.txt with buffer fp1
- `mem_allocate_pattern(foo, malloc, 200, int)` - Function foo uses malloc to allocate 200 bytes of memory casted as integers

These monitoring policies, however, will only find the first occurence of such patterns in the source code for monitoring. To find all occurences of the patterns, the wildcard pattern needs to be applied in sequence with the pattern. For example, to find all occurences where Function foo calls Function bar, the monitoring policy would look like this:

```
( * ; func_call_pattern(foo, bar) )
```

The above policy also demonstrates the use of the sequencing (`;`) operator. Note that the entire sequencing expression must be surrounded by parentheses to indicate associativity in cases of nested expressions. Examples of other operators are as follow:

- `(socket_data_pattern(foo, read, sa, sb) | func_call_pattern(bar, check))` - A socket read occurs concurrently with the function call

- `(file_pattern(foo, fopen, input.txt, fp1) + func_call_pattern(foo, params_in))` - Function foo either opens input.txt with buffer fp1 or calls function params_in

- `(mem_allocate_pattern(foo, malloc, 8, char) ; mem_deallocate_pattern(foo, free))` - Function foo uses malloc to dynamically allocate memory then frees it afterwards

Note that for the last example, while the policy strictly specifies the memory deallocation to happen immediately after the memory allocation (with no other code patterns in between), an imperfect match will still occur even if other code patterns do exist in between the first and second patterns of the sequencing operator. However, just as with single term policies, without a wildcard pattern only the first matching code pattern is picked out for monitoring. To illustrate this concept, consider the following code fragment for function foo:

```
(1) int *a = (int*) malloc(10 * sizeof (int));
(2) int *b = (int*) malloc(10 * sizeof (int));
(3) free(a);
(4) free(b);
```

which maps to the following sequence of code patterns for function `foo`:

1. mem_allocate_pattern(foo, malloc, 40, int)

2. mem_allocate_pattern(foo, malloc, 40, int)
3. mem_deallocate_pattern(foo, free)
4. mem_deallocate_pattern(foo, free)

With a monitoring policy of

```
(mem_allocate_pattern(foo, malloc, 8, char) ;
mem_deallocate_pattern(foo, free)),
```

only patterns (1) and (3) from function foo will be matched. To find all instances of memory allocation followed by memory deallocation in function foo, the monitoring policy should be defined as:

```
((mem_allocate_pattern(foo, malloc, 8, char) ; * ) ;
mem_deallocate_pattern(foo, free))),
```

which would match with all four combinations of memory allocation followed by memory deallocation, namely (1) then (3), (1) then (4), (2) then (3) and (2) then (4).

The previous monitoring policy example also demonstrated the use of nesting in defining a monitoring policy and showed why the parentheses are important in determining the operator's associativity. The following are examples of some possible complex nested monitoring policies:

- `(func_call_pattern(foo, bar) ; (file_pattern (bar, FOPEN, barin.txt, barbuf) ; socket_close_pattern(foo, s1)))`

- `(func_call_pattern(foo, bar) ;`

  `(socket_open_pattern(foo, sf, 128.0.0.1, 80) ;`

  `(func_call_pattern(check, bar) ;`

  `(socket_close_pattern(check, sf) ;`

  `socket_data_pattern(bar, read, sf, buff1)))))`

- `(mem_allocate_pattern(foo, calloc, 8, int) ;`

  `(file_pattern(foo, fopen, input.txt, fp1) +`

  `func_call_pattern(foo, params_in)))`

Chapter 5

Monitoring Policy Operational Model

After forming a syntatically correct monitoring policy, the next step is to parse the text based policy into a useful representation for further processing. This chapter discusses the monitoring policy parser developed using JavaCC as well as the tree representation and the Petri Net representation of the monitoring policy as an output of the monitoring policy parser. This chapter will conclude with a few examples demonstrating the parsing process.

5.1 JavaCC Generated Parser

JavaCC (Java Compiler Compiler) [61] is an open source parser generator for the Java programming language licensed under the Berkeley Software Distribution

(BSD) License. Given an EBNF grammar, JavaCC generates a top-down parser in Java source code. However, top-down parsers are limited to the LL(k) class of grammars, and in particular, left recursion cannot be used in the grammar. To circumvent this restriction, JavaCC supports token stream look ahead, and thus offers both semantic and syntactic lookahead. This feature is particularly important here as nesting and recursion is a significant element of the monitoring policy grammar.

Although conceptually speaking, JavaCC does take an EBNF grammar as input and will generate a top-down parser in Java source code as output, the actual input provided to JavaCC is in fact slightly more complicated. First of all, lookahead statements must be added to resolve any choice conflicts that arise between two EBNF productions in the grammar. For example, for the monitoring policy's EBNF grammar listed in Table 4.2, it is important to add a lookahead of 3 for the *AggregateExpression* statements. By looking ahead 3 tokens in the token stream and seeing the operator, the parser can determine whether the current token belongs to a *SequenceExpression*, a *ChoiceExpression* or a *ConcurrentExpression.*

Strictly speaking, running the JavaCC generated parser on a monitoring policy only checks the policy for its syntactic correctness. However, the objective of the monitoring policy parser is more than verification - it needs to convert the text-based monitoring policy into a meaningful representation for further processing. By adding semantic actions to the grammar file itself, code statements will be triggered during parsing. Depending on the grammar production exercised, code blocks containing different data structure constructor statements are executed and a data structure representation of the policy can be built in dynamic memory. The following subsection will present how such semantic actions are used to generate a tree representation of the monitoring policy.

5.2 Tree Representation

As with any language parsers, a natural representation of a language input statement is to use a tree structure. As a simple example, consider the simple monitoring policy of $\alpha\ ;\ \beta$, where α and β are event patterns. The corresponding tree structure will be depicted in Figure 5.1 below. As seen, the operator is the

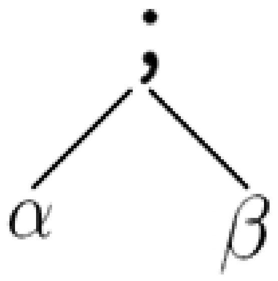


Figure 5.1: Tree Representation of a Simple Monitoring Policy

parent node with its left and right children corresponding to the event patterns the operator joins together. However, for all practical purposes, monitoring policies are usually more complicated with nested structures and repeated notes. Moreover, the event patterns also have parameters that are not accounted for in the previous simple example.

To handle parameters for the event patterns, a hash table is built to store all parameters found in the monitoring policy, and the event patterns in the tree structure will have pointers pointing to the appropriate parameters. Since these parameters are static and will not change, only one copy of duplicate parameters are stored in the hash table where multiple pointers can make reference to it. For example, for the simple monitoring policy above, if α takes parameters θ and λ and β takes parameters λ and ϕ, then the corresponding tree structure would look like Figure 5.2 below.

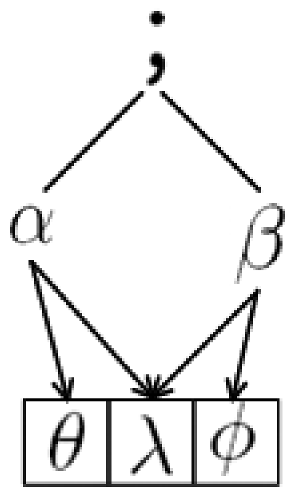


Figure 5.2: Tree Representation of a Simple Monitoring Policy with Parameters

To handle nested structures, each node of the tree representation can be either a event pattern or an operator, and in the latter case, the node must have two children node. Therefore, a nested monitoring policy such as `((`α `+` β`)` | γ`)` would look like Figure 5.3 below (parameters are not displayed for simplicity).

Event patterns, like parameters, are also stored in a hash table where the tree nodes can make reference to via pointers. This addition layer of referencing is necessary to avoid cycles in the tree, which would happen if the event patterns are stored directly in the tree data structure. Consider the example policy of `((`α `;`

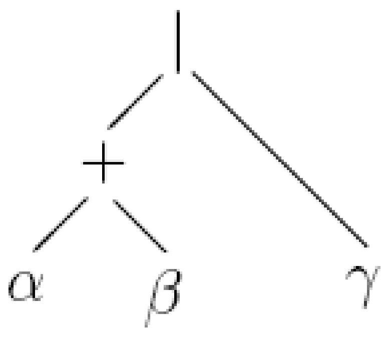


Figure 5.3: Tree Representation of a Nested Monitoring Policy

β) + ((α | β)). Figure 5.4 below clearly exemplifies the need of the additional layer of referencing. Storing the event patterns directly into the tree nodes will result in the tree to the left, whereas having the tree nodes make reference to the event patterns stored in a hash table will generate the desired tree representation as shown in the tree to the right.

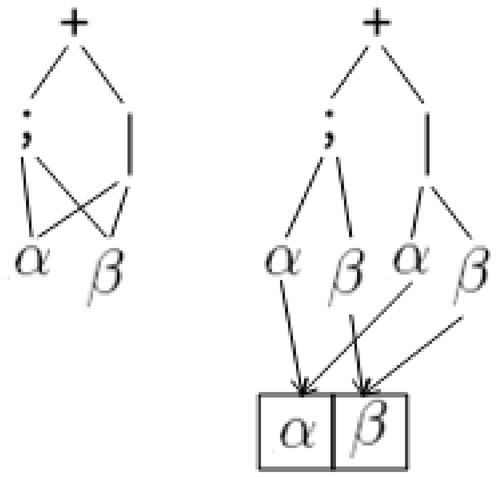


Figure 5.4: Tree Representation of a Monitoring Policy with Repeated Nodes

Ultimately, the entire tree is stored in a binary tree data structure where each node has a left child and a right child (which can be null for leaf nodes). The binary tree data structure also maintains the head node of the tree by checking if a new node contains the most children every time a node is added to the tree.

5.3 Petri Net Representation

The tree representation is only a preliminary representation of the monitoring policy in dynamic memory. For purposes of pattern matching, this tree representation of the monitoring policy needs to be transformed into a Petri Net representation. The Petri Net representation is used very much like a state machine for the purposes of pattern matching, as discussed in Chapter 7; however, since the execution of Petri Nets is non-deterministic, it is well suited for modeling the concurrent behaviour of a software system.

A place in a Petri Net represents a event pattern in the monitoring policy whereas a combination of places, transitions and directed arcs represent an operator. Figure 5.5 presents the mapping between a monitoring policy operator and its corresponding Petri Net representation. These three Petri Net templates are the building blocks of the Petri Net representation of any monitoring policy. The transformation from a tree representation to a Petri Net representation of a monitoring policy relies on building and substituting these three Petri Net templates into each other. The transformation algorithm can be summarized as follows:

Step 1: Traverse the tree in a pre-order fashion.

Step 2: Upon reaching an operator node, build the corresponding Petri Net representation for the operator, leaving the places empty to be filled in later. If

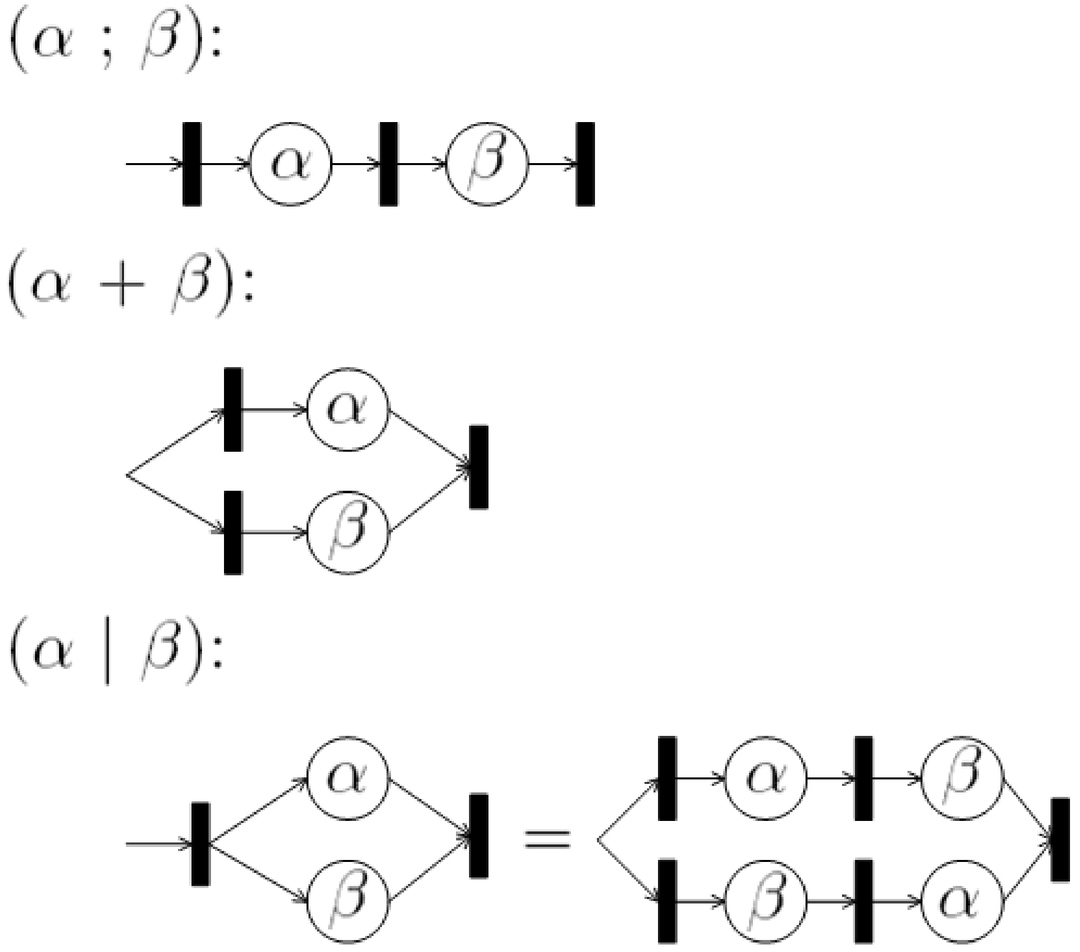


Figure 5.5: Mapping Between Monitoring Policy Operators and Petri Net Representations

an event pattern rather than an operator is reached, then a leaf node of the tree is reached and there is no need to recurse further down the tree.

Step 3: After the corresponding Petri Net template has been constructed for the operator, go to step 1 for its left subtree to populate the first place, and then repeat for the right subtree to populate the second place.

As a simple visual example demosntrating this process, consider the tree representation of the monitoring policy `(`(α `;` β`)` `;` γ`)` shown in Figure 5.6 below, where α, β and γ are all event patterns. Following the algorithm, the tree is tra-

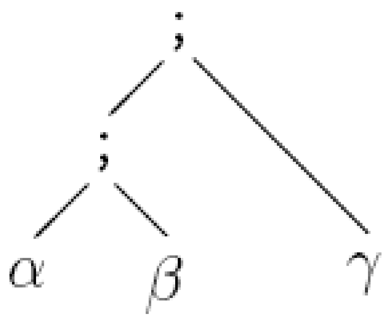


Figure 5.6: Sample Tree Representation of a Monitoring Policy

versed in a pre-order fashion, and the first node encountered is the head node, which is a ; operator. The corresponding Petri Net template is constructed with the two places empty. The left subtree is then traversed, and the node encountered is also a ; operator, and the corresponding Petri Net template is also constructed. The left subtree is traversed again, only this time the event pattern α is reached and the first place in the preceeding Petri Net template is populated. The right subtree is now traversed, and likewise the event pattern β is reached and the second place is populated. Since this subtree is now completely traversed and the

Petri Net template is fully populated, this entire template is substituted into the original Petri net template's first place, which has been left empty. Now the right subtree of the head node is traversed, and since the event pattern γ is reached, the second place of the Petri net is populated, and the entire tree has been traversed and has been transformed into its corresponding Petri Net representation. Figure 5.7 summarizes this process.

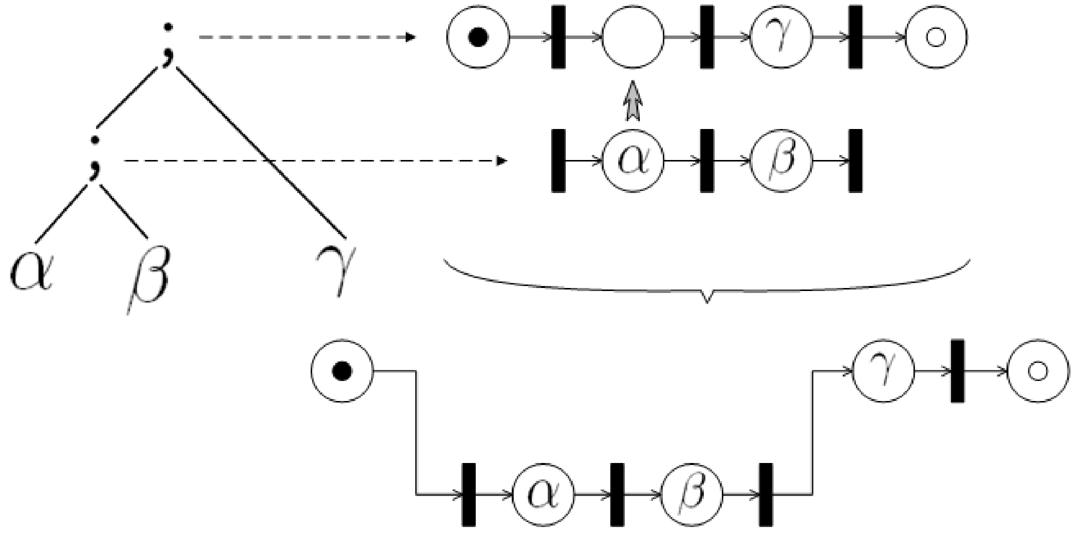


Figure 5.7: Transformation of a Tree Representation to a Petri Net Representation of a Sample Monitoring Policy

5.4 Example of Monitoring Policy Parsing

This section will walk through an example of transforming a monitoring policy from its textual representation to a tree representation through parsing and then to a Petri Net transformation using the algorithm mentioned above. Consider the following monitoring policy:

```
 ( ( (file_pattern(foo, fopen, input.txt, fb1) +
socket_open_pattern(foo, sf, 128.0.0.1, 23) ) ;
func_call_pattern(foo, bar) ) |
( mem_allocate_pattern(bar, malloc, 8, char) +
func_call_pattern(bar, getmem) ) )
```

Putting this monitoring policy through the parser will yield the tree representation shown in Figure 5.8. For simplicity, the event patterns and the parameters it references will be substituted by a Greek letter each as indicated. Once the tree representation has been established, the Petri Net transformation algorithm is applied to the head node of the tree. Figure 5.9 demonstrates this transformation and the final Petri Net representation of the monitoring policy.

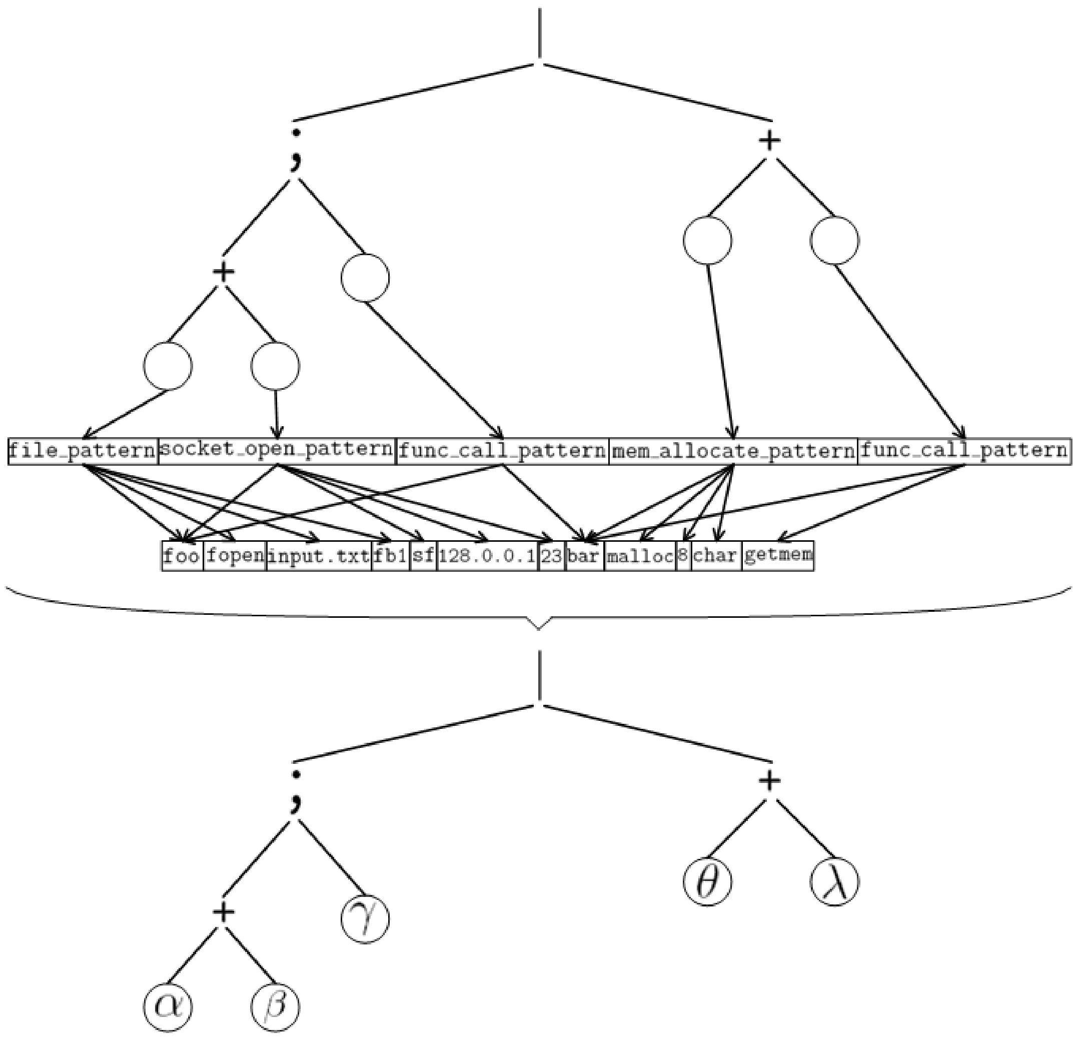


Figure 5.8: Tree Representation of the Example Monitoring Policy

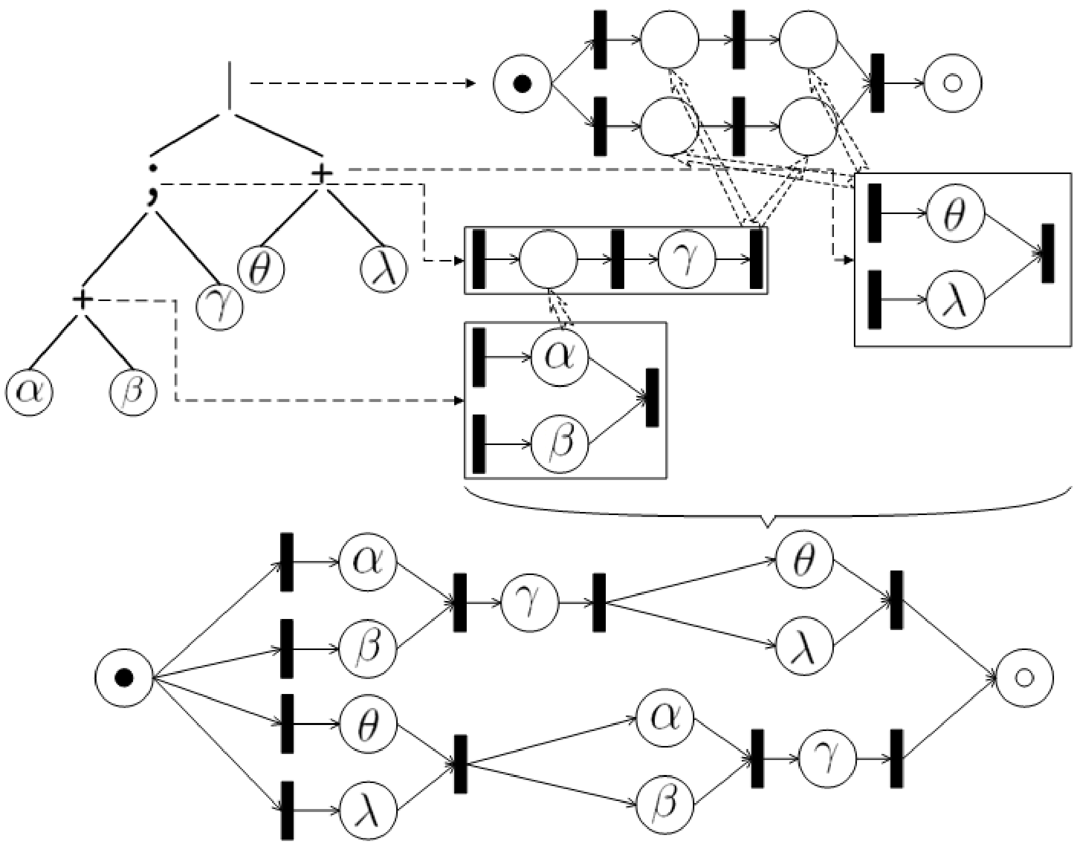


Figure 5.9: Petri Net Representation of the Example Monitoring Policy

Chapter 6

Source Code Annotation

While on one end of the process is to transform the text-based monitoring policy into a Petri Net representation for pattern matching purposes, another aspect of the process is to prepare the source code for pattern matching as well. The preparation of the source code can be divided into two major steps — source code parsing and source code annotation. The end goal of source code preparation is to execute the annotated source code to discover the exeuction path of the piece of code, upon which pattern matching can occur to locate the areas in the source code that corresponds to the patterns defined in the monitoring policy. This chapter will outline the two major steps in detail and demonstrate how this end goal of source code preparation can be reached. While the discussion in this chapter mainly pertains to the C programming language, it will be shown that these concepts can be easily applied to any other programming language.

6.1 C Code Parser

Just as the monitoring policy parser is built with JavaCC, the C code parser is also built with JavaCC. However, there are a number of grammar files publicly available for the C programming language such as from [61], and hence there is no need to analyze and redevelop the grammar file for C. As a proof of concept, the C grammar definition created by Doug South and later modified by Tom Copeland is used. It is also noteworthy that grammar files for different languages such as C++ or Java are also publicly available and hence it is very easy to adapt the concepts disucssed in this chapter to a different programming language.

The C code parser created with JavaCC has the limitation of parsing ANSI C code only. In particular, function calls must be made to functions that are previously defined before the call or has a function prototype declared. Taking advantage of this restriction, a hash table is used to keep track of all functions declared in the source code during parsing to identify all function calls made in the source code.

Similar to the monitoring policy parser, semantic actions are added to the C grammar file to create a data structure that reflects the structure of the source code. Normally semantic actions will be added to all grammar productions to produce a full Abstract Syntax Tree (AST), but for the purposes of system monitoring, only specific types of function calls are of interest and hence only data related to function declarations and function calls are stored. Specifically, semantic action is added for the grammar productions for function declaration, function calls and their corresponding parameter lists. The domain model for the data stored is similar to the one used for the monitoring policy; in fact, the code patterns in the monitoring policy domain model are reused in the source code parser. Specifically, upon reaching a primary expression during parsing, the token is first compared against a list of function names for the code patterns such as file_pattern or mem_allocate_pattern. If a match is found, the corresponding code pattern object is constructed and stored in order in a vector along with the function call's line and column number. This information is critical in relocating the function call during source code annotation. If it is not a match, then the token is compared against the hash table of existing functions in the source code to determine if it is a func_call_pattern, upon which a func_call_pattern will be created and stored in the vector as well. If the token does not match with the hash table either, then the function call must be a system call that is not in the domain model of the monitoring policy and therefore is disregarded.

As a simple example demonstrating the data structures generated by the source code parser, consider the following function:

```
void foo(int* i, int count)
{
   int *iptr = malloc(10 * sizeof (int));
   printf("%d: Hello World from foo", count);
   count--;
   if (count > 0) {
      foo(iptr, count);
   }
}
```

Upon parsing this function, the source code parser will first recognize the function definition in line 1, check if the function name foo is already in the hash table of existing user function names (possibly due to a function prototype earlier on in the code), and add it to the hash table if this is the first encounterance of the function foo. In line 3, the parser encounters a function call to malloc, compares it against the list of predefined functions relevant to monitoring and discovers malloc is one of the functions to monitor. The parser invokes the semantic action to create a mem_allocate_pattern object with the corresponding parameters and appends it to the vector of previously constructed nodes. Proceeding to line 4, the parser identifies a function call to printf. However, printf is not on the list of predefined functions relevant to monitoring, nor is it in the hash table of existing user function names, hence the parser takes no action. Lines 5 and 6 do not contain any explicit function call statements so the parser continues to line 7, where a function call to foo is seen. While foo is not in the list of predefined functions relevant to monitoring, it is in the hash table of existing user function names (either from a function prototype or from the insertion in line 1) and hence the corresponding semantic action is invoked. A func_call_pattern object with foo as both the caller and callee functions at line 7 is constructed, and added to the vector of previously constructed nodes. The parser completes parsing the function with no further additions to the data structures. The result of parsing the above function foo is the addition of two code pattern nodes to the vector, as well as a possible addition (if not already added) of the function foo to the hash table of existing user functions.

6.2 Code Annotator

After the source code has been parsed and a vector of nodes containing all relevant monitoring code patterns in the source code has been created, the source code can then be annotated. As a proof of concept for this thesis, the code annotation are based on simple `printf` statements that will collectively reflect the execution path of the source code along with the annotated output; however, in future work, more complex annotations can be made using this technique, as discussed in Chapter 10. The format of the annotation will be the stamp `*Annotated*` followed by an epoch timestamp and then the pattern type and details regarding the pattern. The annotation will end with the line and column number of where the pattern is located. An example of one such annotation will be as follow:

```
*Annotated*915784682 FunctionCall - Function foo declared at
32:5->Function bar declared at 72:8@36:9
```

The algorithm with annotation is fairly simple and straightforward. Since the nodes in the vector are organized in sequential order of the source code, the annotating process can be completed in one pass by iterating through all the nodes in the vector, annotating the line, and advancing the file pointer to the location specified by the next node in the vector. The only exception is when there are two nodes that reference the same line of source code, upon which the file pointer does not need to be advanced.

Once the correct line in the source code has been reached, the exact position of the function call is located. The annotator then traces backwards to the previous semicolon or start of line, which is the position for the code annotation to occur. This step of annotating before the entire function call statement is taken due to a number of considerations. While it may be easiest to perform the code annotation at the end of line of where the function call is located, there are several problems with this approach. Firstly, since the annotated code only executes after the function call completes, the annotated code may in fact never be executed if the flow of execution does not return from the function call due to a program failure or even a graceful exit in the function called. Secondly, annotation at the end of the line may cause compilation errors in a program. Consider the following code fragment, where `foo` is the name of a function that takes variable i as a valid parameter.

```
(01) if (foo(i) > 0)
(02) {
(03)    //block of code
(04) }
```

Annotating the function call to foo at line 1 at the end of the line will result in a compile error as a opening brace ({) is expected after the if condition. For a similar reason, the the code annotation cannot be placed immediately preceeding the function call, as annotating the function call to foo at line 1 in this manner will lead to the annotation to appear in the if condition and most likely result in a compile error. Therefore, it is most appropriate to place the code annotation at the start of the line where the function call takes place. However, consider the following code fragment that has been validly condensed into one line:

```
if (foo(i) > 0) bar(i);
```

Since there are two function calls on the same line, if the annotation algorithm was to simply place the annotation at the beginning of the line, the annotation for `bar` will come before `foo`, since `foo` is first annotated and `bar`'s annotation will preceed `foo`'s. This results in an improper order of annotation that does not reflect the execution path of the source code. Therefore, the annotation algorithm tracks back to the closest preceeding semicolon from the function call to place the code annotation, and only when no preceeding semicolons are found is the annotation placed at the beginning of the line.

6.3 Example of Source Code Annotation

As an example of how the source code parser and annotator works, consider the following C source code as the input to the source code parser and annotator.

```
void display(int, const char*);
int main()
{
    int a = 42;
    display(a, "Hello World");
    return 0;
}
void display(int i, const char* b)
{
    printf("%s %d\n", b, i);
}
```

The source code parser recognizes the first line as a function prototype and stores the function `display` in the existing function hash table. Similarly, it recognizes the second line as a function definition's header, and stores the function `main` in the existing function hash table. The parser proceeds to recognize lines 3 and 4 as valid C syntax but is insignificant to the purposes of policy-based system monitoring, and hence no semantic actions are invoked. Upon reaching line 5, a function call is recognized by the parser and it compares the function name `display` against its list of system functions to monitor for but does not find a match. However, it finds a match for the function name `display` against the existing function hash table, and hence forms a func_call_pattern object with caller `main` and callee `display` and stores it in the vector of created nodes. The parser then proceeds through lines 6 and 7 but finds nothing special, but upon reaching line 8 it recognizes a function definition's header. However, it matches the function name of `display` to the existing function hash table and hence no further action is taken by the parser. Upon reaching line 10, the parser again recognizes a function call, but this time it fails to match `printf` to the list of system functions to monitor nor the existing function hash table, and hence the parser takes no action. The parser finishes parsing the source code file with the following output indicating the objects in the hash table followed by a listing of nodes currently in the vector.

```
(01) {Functionmain=Function main declared at 2:5, Functiondisplay=
     Function display declared at 1:6}
(02) [FunctionCall-Function main declared at 2:5->Function display
     declared at 1:6@5:9]
```

As a result of the source code parsing, a vector containing the func_call_pattern object is passed to the source code annotator. The source code annotator iterates through the vector and moves the file pointer to line 5 as specified by the func_call_pattern object. It locates the function name `display` on line 5 and traces backwards to find the immediately preceeding semicolon or the start of line. Since there are no semicolons preceeding the function call, the annotation for a func_call_pattern is made at the start of line. After this annotation, there are no more objects in the vector and hence the annotator exits with the following annotated source code.

```
(01) void display(int, const char*);
(02) int main()
(03) {
(04)     int a = 42;
(05)     printf("*Annotated*%d FunctionCall - Function main declared
     at 2:5->Function display declared at 1:6@5:9\n", time((time_t
      *) NULL)); display(a, "Hello World");
(06)     return 0;
(07) }
(08) void display(int i, const char* b)
(09) {
(10)     printf("%s %d\n", b, i);
(11) }
```

Executing the above annotated source code will yield the following program output that can be used for pattern matching.

```
(01) *Annotated*1175736001 FunctionCall - Function main declared at
     3:5->Function display declared at 1:6@6:9
(02) Hello World 42
```

Chapter 7

Pattern Matching

The final step in the entire process is to match the monitoring policy against the source code. The purpose of this step is to identify where intrusive monitors can be added into the source code in an automated fashion. The pattern matcher requires two inputs: the Petri Net representation of the monitoring policy and the annotated source code's output. This chapter will describe the basic pattern matching process and present an example of the process. It will then discuss how this process' underlying algorithm can be improved with concepts taken from Artificial Intelligence theories such as dynamic programming.

7.1 Basic Process

The basic idea behind the pattern matching process is to treat the Petri Net representation of the monitoring policy as an automaton that takes the annotated source code's output (i.e. logged events) as its input. Paths through the Petri Net are explored and assigned a score based on the number of deletions required from the input to formulate a match. The path with the lowest score indicates the closest match to what is specified in the monitoring policy and the path's precise location in the source code can be retrieved from the data provided in the annotated source code's output.

The algorithm which explores the paths through the Petri Net is similar to the class of informed search algorithms found in literature on Artificial Intelligence. The basic process treats the Petri Net as a specification guide to a branch and bound search problem. The fundamental idea behind the pattern matching process is to find the optimal alignment between the Petri Net model that represents the monitoring policy and the sequence of logged events. The process for matching the Petri Net representation of the monitoring policy to the annotated source code's output is described below.

ALGORITHM:	Pattern Matching Algorithm for the Policy Driven Software Monitoring Framework
PURPOSE:	To correlate the sequence of events defined in the monitoring policy to the location in the source code
INPUT:	Annotated source code output, Petri Net representation of monitoring policy
OUTPUT:	Solution paths indicating the location of source code for each pattern in the monitoring policy

Step 1: Take the starting node from the Petri Net and add it to the queue of

feasible nodes for expansion.

Step 2: Make a copy of the queue for comparison (hereafter known as the queue copy). This is necessary as modifications are made during the queue during the comparison and it will be incorrect to make comparisons against the new modifications.

Step 3: Compare the first relevant line of input (from the annotated source code's output) against each node of the queue copy. If the input matches with the node in the queue copy, then the node is consumed, upon which all directly reachable places from the place in the Petri Net which corresponds to the node that was consumed is added to the original queue and the score is carried over to these new nodes. The consumed node is also removed from the original queue. On the other hand, if the input does not match with the node in the queue copy, then a pre-determined penalty score is added to the corresponding node in the original queue.

Step 4: Repeat steps 2 and 3 until the entire set of input from the annotated source code's output has been matched. Once all the input has been exhausted, what is left in the queue are the feasible solutions to the pattern matching problem.

All nodes in the queue that correspond to the end place of the Petri Net represent a complete path through the Petri Net. In other words, a match has been found between the monitoring policy and the source code. It is possible that there are multiple nodes in the queue that correspond to the end place of the Petri Net at the end of the algorithm. In this case, the node with the lowest score represent the path containing the closest match between the monitoring policy and the source code, although other nodes in the queue that correspond to the Petri Net's end place are also feasible solutions. Nodes that are in the queue but do not correspond to the end place of the Petri Net are not complete paths through the Petri Net and hence are not solutions, regardless of the score the node has.

While an ending place node in the queue indicates a solution exists, the actual solution path is the important data that helps the software engineer determine where to place intrusive monitors in the source code. The simplest solution to keep track of the solution path is to keep a pointer to the preceeding node each time a node is added to the queue and the preceeding node consumed. This effectively forms a linked list between the queue nodes. By tracing backwards from the end node, the complete path in the source code matching the monitoring policy can be recovered. For example, the final output of the pattern matcher may be as follow:

```
(01) +++SOLUTION+++
(02) MemAllocate-foo using malloc allocating 42 for int at 505:27
      <--> MemDeallocate-foo using free deallocating at 523:11
(03) 1 Solution(s) found.
```

This output indicates a match has been found between the source code and the monitoring policy at lines 505 and 523 with regards to the mem_allocate_pattern and the mem_deallocate_pattern respectively.

One important exception to the algorithm is the treatment of wildcards in the monitoring policy. The wildcard code pattern should be treated as a special case in the algorithm, and in particular the wildcard node should never be consumed, since it can match with any number of code patterns. In other words, if input is compared against a wildcard node in step 3 of the algorithm above, then it is considered a match and all directly reachable places from the wildcard place in the Petri Net is added to the original queue and the score of the wildcard node is carried over to these new nodes, except the wildcard node is not removed from the original queue. Furthermore, since the wildcard can represent zero as well as multiple code patterns, upon adding reachable nodes from the Petri Net to the queue, if a wildcard node is added, then all directly reachable places from that wildcard place in the Petri Net must also be added to the queue as well. The example in the following section will provide a demonstration of the operation of the pattern matcher with wildcard nodes.

7.2 Example of Pattern Matching

As an example of how the pattern matcher works, consider the following inputs for the pattern matcher. The following is the annotated source code for a simple C program that contains four functions: main, increment, decrement and display.

```
(01) void display(int, const char*);
(02) int increment(int a);
(03)
(04) int main()
(05) {
(06)    int a = 3;
(07)    printf("*Annotated*%d FunctionCall - Function main declared
      at 4:5-> Function display declared at 1:6@7:9\n", time(
     (time_t *)NULL)); display(a, "Hello World");
(08)    return(a);
(09) }
(10)
(11) int decrement(int j)
(12) {
(13)    return j-1;
(14) }
(15)
(16) int increment(int j)
(17) {
(18)    return j+1;
(19) }
(20)
(21) void display(int i, const char* b)
(22) {
(23)    printf("*Annotated*%d FunctionCall - Function display
      declared at 21:6-> Function decrement declared at 11:5@23:
     13\n", time((time_t *)NULL));printf("*Annotated*%d FunctionCall
      - Function display declared at 21:6-> Function increment
     declared at 2:5@23:33\n", time((time_t *)NULL)); if (
     decrement(i) > 0 && increment(5) > 0) {
(24)      printf("%s %d\n", b, i);
(25)      printf("*Annotated*%d FunctionCall - Function display
     declared at 21:6->Function decrement declared at 11:5@25:21\n",
     time((time_t *)NULL));   i = decrement(i);
(26)      printf("*Annotated*%d FunctionCall - Function display
     declared at 21:6->Function display declared at 1:6@26:17\n",
     time((time_t *)NULL));   display(i, b);
(27)    }
(28) }
```

Executing the above source code will yield the following output that will be used as input to the pattern matcher.

```
(01) *Annotated*1175739269 FunctionCall - Function main declared at
     4:5->Function display declared at 1:6@7:9
```

```
(02) *Annotated*1175739269 FunctionCall - Function display declared
     at 21:6->Function decrement declared at 11:5@23:13
(03) *Annotated*1175739269 FunctionCall - Function display declared
     at 21:6->Function increment declared at 2:5@23:33
(04) Hello World 3
(05) *Annotated*1175739269 FunctionCall - Function display declared
     at 21:6->Function decrement declared at 11:5@25:21
(06) *Annotated*1175739269 FunctionCall - Function display declared
     at 21:6->Function display declared at 1:6@26:17
(07) *Annotated*1175739269 FunctionCall - Function display declared
     at 21:6->Function decrement declared at 11:5@23:13
(08) *Annotated*1175739269 FunctionCall - Function display declared
     at 21:6->Function increment declared at 2:5@23:33
(09) Hello World 2
(10) *Annotated*1175739269 FunctionCall - Function display declared
     at 21:6->Function decrement declared at 11:5@25:21
(11) *Annotated*1175739269 FunctionCall - Function display declared
     at 21:6->Function display declared at 1:6@26:17
(12) *Annotated*1175739269 FunctionCall - Function display declared
     at 21:6->Function decrement declared at 11:5@23:13
(13) *Annotated*1175739269 FunctionCall - Function display declared
     at 21:6->Function increment declared at 2:5@23:33
```

It is noteworthy to mention that there may be normal program output (such as lines 4 and 9) mixed in with the annotated source code output; however, these output will not interfere with the pattern matcher as long as it does not begin with the annotation tag of `*Annotated*`.

Suppose the monitoring policy is as follow:

```
(01) (func_call_pattern(display, display);
(02) (* ; (func_call_pattern(display, display)+
(03) func_call_pattern(display, increment))))
```

Parsing this monitoring policy yields the tree representation and Petri Net as shown in Figure 7.1, where α represents func_call_pattern(display, display) and β represents func_call_pattern(display, increment).

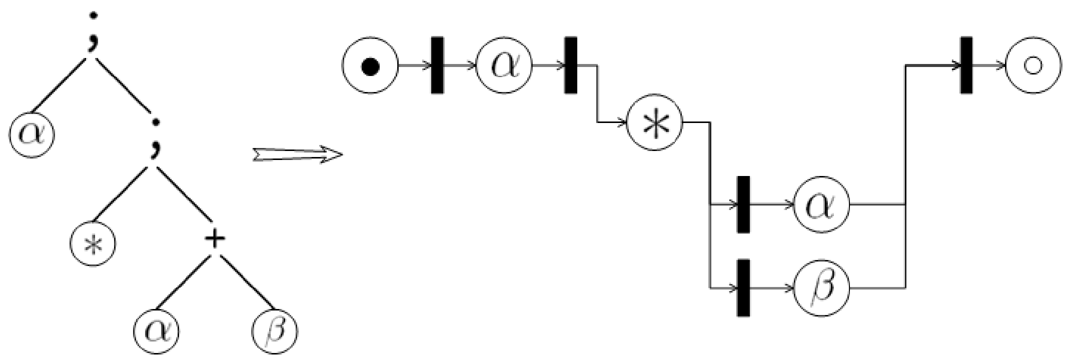


Figure 7.1: Tree and Petri Net Representations of the Sample Monitoring Policy Demonstrating Pattern Matching

Table 7.1 details the operation of the pattern matching algorithm in a step by step manner. The number in square brackets beside each queue item represents the score for that queue node. Function_Call_Pattern has been abbreviated to FC, and a penalty score of p is assigned to each non-matching input.

At the beginning of the algorithm (iteration 0), the queue is initialized to the first node of the Petri Net, which is represented by FC(display, display) and has a score of 0. The next four inputs do not match with this code pattern and hence the score for this queue node increments by p each iteration. The fifth input of FC(display, display) matches with the only node on the queue, upon which the node is expanded by placing all its successors in the Petri Net into the queue and carrying over the score while removing the node. Since the node placed on the queue is a wildcard node, its successors must be placed on the queue as well.

Table 7.1: Operations of the Pattern Matcher on Sample Inputs

Iteration	**Input**	**Queue**
0		FC(display, display)[0]
1	FC(main, display)	FC(display, display)[*p*]
2	FC(display, decrement)	FC(display, display)[*2p*]
3	FC(display, increment)	FC(display, display)[*3p*]
4	FC(display, decrement)	FC(display, display)[*4p*]
5	FC(display, display)	*[*4p*]; FC(display, display)[*4p*]; FC(display, increment)[*4p*]
6	FC(display, decrement)	*[*4p*]; FC(display, display)[*4p*]; FC(display, increment)[*4p*]; FC(display, display)[*5p*]; FC(display, increment)[*5p*]
7	FC(display, increment)	*[*4p*]; FC(display, display)[*4p*]; FC(display, increment)[*4p*]; FC(display, display)[*5p*]; null[*4p*]; FC(display, display)[*6p*]; null[*5p*]
8	FC(display, decrement)	*[*4p*]; FC(display, display)[*4p*]; FC(display, increment)[*4p*]; FC(display, display)[*5p*]; FC(display, increment)[*5p*]; FC(display, display)[*6p*]; null[*5p*]; FC(display, display)[*7p*]; null[*6p*]
9	FC(display, display)	*[*4p*]; FC(display, display)[*4p*]; FC(display, increment)[*4p*]; null[*4p*]; FC(display, increment)[*5p*]; null[*5p*]; FC(display, increment)[*6p*]; null[*6p*]; null[*6p*]; null[*7p*]; null[*7p*]
10	FC(display, decrement)	*[*4p*]; FC(display, display)[*4p*]; FC(display, increment)[*4p*]; FC(display, display)[*5p*]; FC(display, increment)[*5p*]; null[*5p*]; FC(display, increment)[*6p*]; null[*6p*]; FC(display, increment)[*7p*]; null[*7p*]; null[*8p*]; null[*7p*]; null[*8p*]
11	FC(display, increment)	*[*4p*]; FC(display, display)[*4p*]; FC(display, increment)[*4p*]; FC(display, display)[*5p*]; null[*4p*]; FC(display, display)[*6p*]; null[*5p*]; null[*6p*]; null[*6p*]; null[*7p*]; null[*7p*]; null[*8p*]; null[*8p*]; null[*9p*]; null[*9p*]

This is necessary to handle the case when the wildcard matches with nothing. These queue operations result in three nodes on the queue at the end of the fifth iteration, all with the score of $4p$. The sixth input does not match with anything on the queue, and hence normally all queue nodes will have their score incremented and no new nodes will be added to the queue. However, since the wildcard node matches with everything, a match in fact occurs and the wildcard node as well as its two immediate successor nodes are again added to the queue, carrying over their previous score of $4p$. This results in five nodes on the queue at the end of the sixth iteration. The seventh input of FC(display, increment) matches with some of the nodes, and these nodes are expanded accordingly. In this case, it is important to point out that the successors to the matching nodes are the ending node of the Petri Net, and hence a null node is inserted to represent a solution has been found. The way the wildcard nodes and the non-matching nodes are treated is identical to the previous iterations. In the eighth iteration, the input of FC(dispaly, decrement) does not match with any of the queue nodes (except for the wildcard node) and hence the score of all these non-matching nodes are incremented by p. Note that the score for the null nodes are also incremented, despite the fact that these nodes represent a solution. This is because the solution path found does not account for the extraneous input after the solution has been found, and hence is penalized accordingly.

This process is repeated until the eleventh iteration, upon which all the input is exhausted. The pattern matcher then traverses the queue looking for null nodes that represent solution paths. The following is the solution output for the pattern matcher, with a penalty score of 1 assigned.

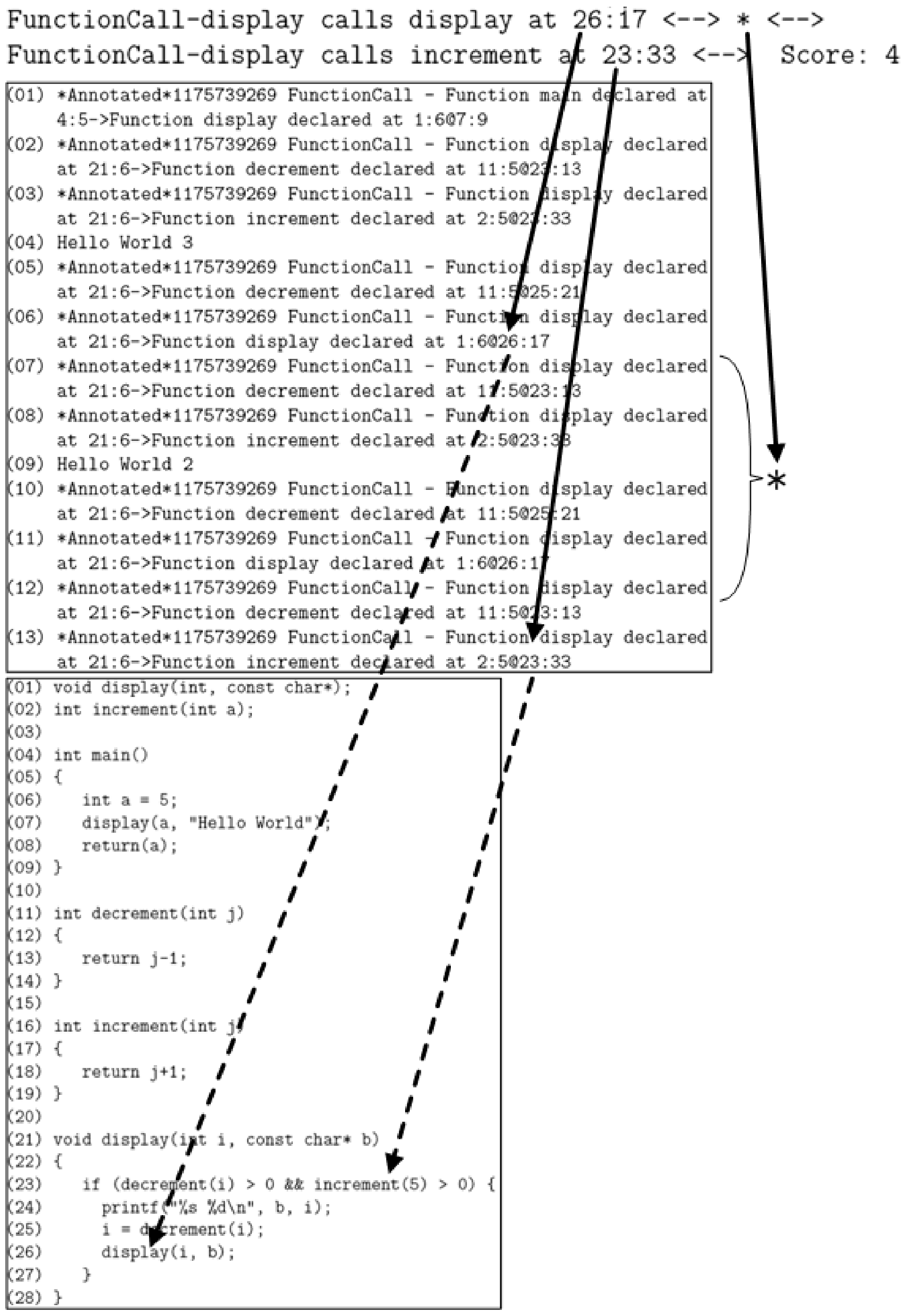
```
(01) +++SOLUTION+++
(02) FunctionCall-display calls display at 26:17 <--> * <-->
     FunctionCall-display calls display at 26:17 <-->  Score: 8
(03) +++SOLUTION+++
(04) FunctionCall-display calls display at 26:17 <--> * <-->
     FunctionCall-display calls increment at 23:33 <-->  Score: 4
(05) +++SOLUTION+++
(06) FunctionCall-display calls display at 26:17 <--> * <-->
     FunctionCall-display calls increment at 23:33 <-->  Score: 6
(07) +++SOLUTION+++
(08) FunctionCall-display calls display at 26:17 <-->
     FunctionCall-display calls increment at 23:33 <-->  Score: 9
(09) +++SOLUTION+++
(10) FunctionCall-display calls display at 26:17 <--> * <-->
     FunctionCall-display calls display at 26:17 <-->  Score: 6
(11) +++SOLUTION+++
(12) FunctionCall-display calls display at 26:17 <--> * <-->
     FunctionCall-display calls increment at 23:33 <-->  Score: 5
(13) +++SOLUTION+++
(14) FunctionCall-display calls display at 26:17 <--> * <-->
     FunctionCall-display calls increment at 23:33 <-->  Score: 7
(15) +++SOLUTION+++
(16) FunctionCall-display calls display at 26:17 <--> * <-->
     FunctionCall-display calls display at 26:17 <-->  Score: 7
(17) +++SOLUTION+++
(18) FunctionCall-display calls display at 26:17 <--> * <-->
     FunctionCall-display calls increment at 23:33 <-->  Score: 8
(19) +++SOLUTION+++
(20) FunctionCall-display calls display at 26:17 <-->
     FunctionCall-display calls display at 26:17 <-->  Score: 9
(21) 10 Solution(s) found.
```

As seen, the ten solution paths outputted by the pattern matcher corresponds to each of the ten null nodes in the queue after the last iteration in Table 7.1 The path with the lowest score, which in this case is 4, is the closest match to the specifications made by the monitoring policy. This path is highlighted in the annotated source code output as well as the source code in Figure 7.2.

7.3 Algorithm Optimization

The algorithm for pattern matching can be optimized by modifying the way the penalty score is assigned. The aforementioned algorithm assigns a constant score for each deletion made to the input in order to achieve a perfect match between the annotated source code output and the monitoring policy. This algorithm effectively does an exhaustive search throughout the Petri Net and hence all the possible solution paths are found at the cost of speed and performance. By adding an admissiable heuristic to the penalty score assignment function and implementing pruning in the algorithm, the pattern matcher can achieve optimality by finding the best matching path at a much better performance. This in essence, is to implement an A* search algorithm from the starting node to the ending node of the Petri Net. It is helpful to borrow concepts from dynamic programming to visualize this optimization. Consider mapping the example monitoring policy and input from the previous section into a 2-dimensional grid as shown in Figure 7.3. The bottom left coordinate (origin) will be the starting point and the coordinate where the last input and the last monitoring policy item intersects will be the goal.

A solution for the pattern matcher is a path that travels from start to goal heading in the positive x and y directions. A match between an input and an element in the monitoring policy is represented by a diagonal segment on the path. The score for a node arises from travelling in the positive x-direction, with the exception of if the y-value of the node is a wildcard node. Figure 7.4 maps the



```
FunctionCall-display calls display at 26:17 <--> * <-->
FunctionCall-display calls increment at 23:33 <-->  Score: 4
```

```
(01) *Annotated*1175739269 FunctionCall - Function main declared at
     4:5->Function display declared at 1:6@7:9
(02) *Annotated*1175739269 FunctionCall - Function display declared
     at 21:6->Function decrement declared at 11:5@23:13
(03) *Annotated*1175739269 FunctionCall - Function display declared
     at 21:6->Function increment declared at 2:5@23:33
(04) Hello World 3
(05) *Annotated*1175739269 FunctionCall - Function display declared
     at 21:6->Function decrement declared at 11:5@25:21
(06) *Annotated*1175739269 FunctionCall - Function display declared
     at 21:6->Function display declared at 1:6@26:17
(07) *Annotated*1175739269 FunctionCall - Function display declared
     at 21:6->Function decrement declared at 11:5@23:13
(08) *Annotated*1175739269 FunctionCall - Function display declared
     at 21:6->Function increment declared at 2:5@23:33
(09) Hello World 2
(10) *Annotated*1175739269 FunctionCall - Function display declared
     at 21:6->Function decrement declared at 11:5@25:21
(11) *Annotated*1175739269 FunctionCall - Function display declared
     at 21:6->Function display declared at 1:6@26:17
(12) *Annotated*1175739269 FunctionCall - Function display declared
     at 21:6->Function decrement declared at 11:5@23:13
(13) *Annotated*1175739269 FunctionCall - Function display declared
     at 21:6->Function increment declared at 2:5@23:33
```

```
(01) void display(int, const char*);
(02) int increment(int a);
(03)
(04) int main()
(05) {
(06)    int a = 5;
(07)    display(a, "Hello World");
(08)    return(a);
(09) }
(10)
(11) int decrement(int j)
(12) {
(13)    return j-1;
(14) }
(15)
(16) int increment(int j)
(17) {
(18)    return j+1;
(19) }
(20)
(21) void display(int i, const char* b)
(22) {
(23)    if (decrement(i) > 0 && increment(5) > 0) {
(24)      printf("%s %d\n", b, i);
(25)      i = decrement(i);
(26)      display(i, b);
(27)    }
(28) }
```

Figure 7.2: Best Matching Path of the Sample Inputs Demonstrating Pattern Matching

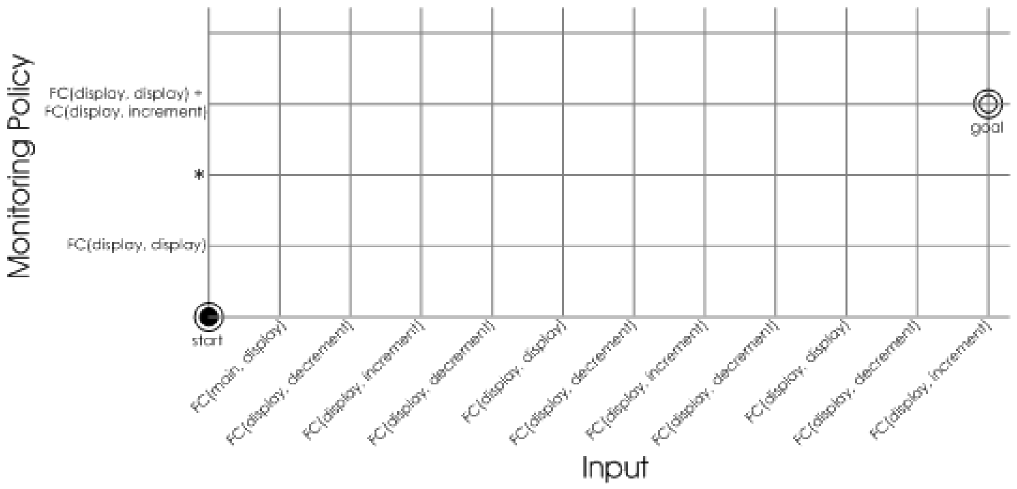


Figure 7.3: 2-Dimensional Grid Visualization of Pattern Matching Sample Problem

optimal solution path from the previous section's example in this 2-dimensional grid.

The sum of the horizontal distances (excluding the horizontal distances travelled at the y-value of the wildcard node) is *4p*, which corresponds to the score of the optimal path. Furthermore, the algorithm can be modified to prune suboptimal paths in the search tree. Consider the two paths in Figure 7.5. Both paths reach the same coordinate on the 2-dimensional grid; however, the top path reaches it with a cumulative score of *4p* while the bottom path reaches it with a cumulative score of *6p*. Since they are at the same point in the graph, therefore it is impossible for the bottom path to obtain a lower score than the top path, and hence the bottom path can be pruned.

Implementation speaking, pruning can be achieved by assigning each place in the Petri Net an unique identifier, and every time an addition needs to be made to

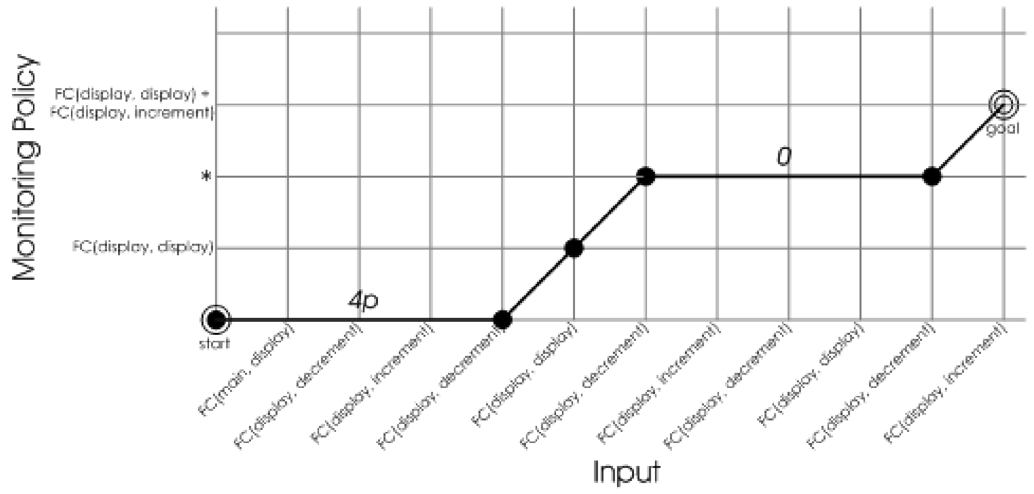


Figure 7.4: Visualization of Optimal Solution Path to Sample Pattern Matching Problem

the queue, the new node is compared against the rest of the queue to see if a node with an identical identifier exists. If that is the case, then their scores are compared and the node with the lowest score should be on the queue. Hence depending on the comparison, the addition to the queue may not occur if the existing node on the queue contains a lower score. Pruning will reduce the size of the search tree which translates to the number of nodes on the queue and thus will significantly speed up computation as there are less nodes to traverse during each iteration.

In essence, the pattern matching problem can been visualized and treated as a graphing problem where the A* search algorithm can be applied which guarantees optimality at the best performance.

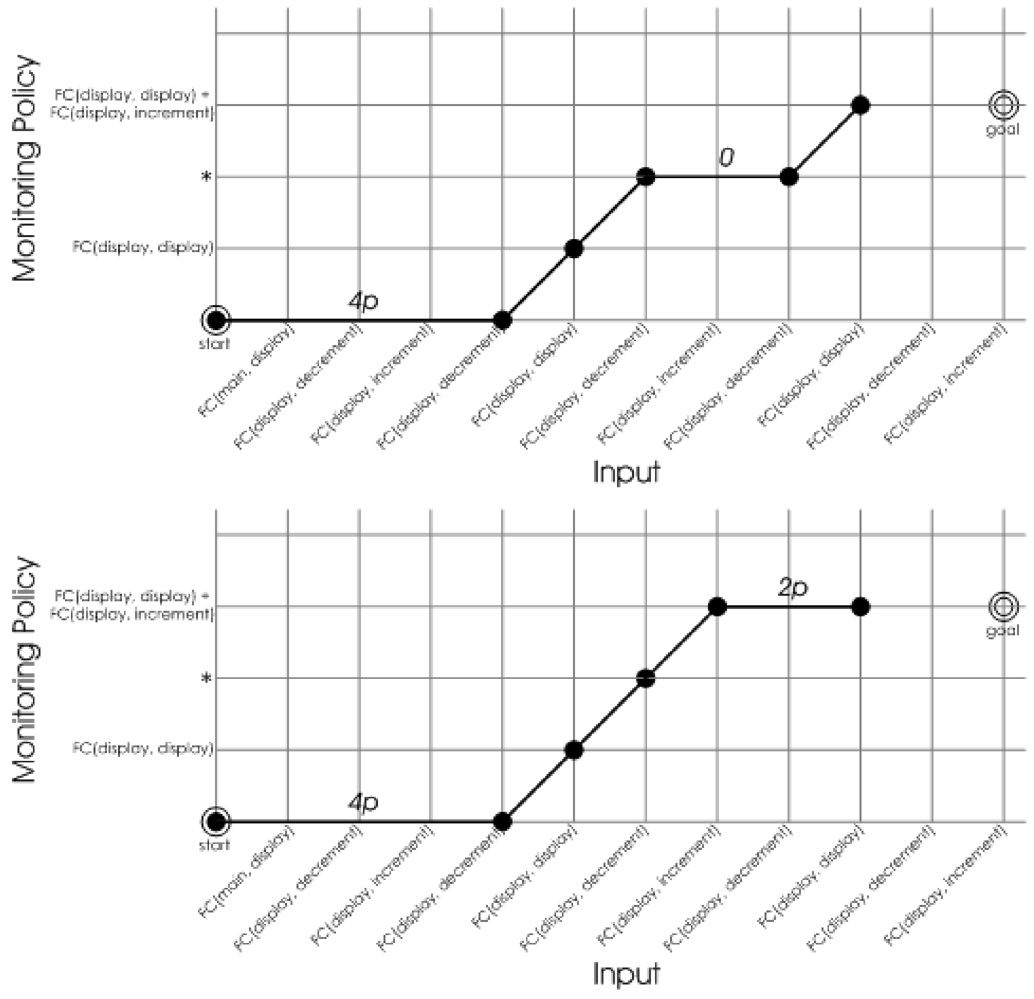


Figure 7.5: Pruning Suboptimal Paths in the Sample Pattern Matching Problem

Chapter 8

Adaptive Logging

While the framework proposed and demonstrated in the previous chapters enhances system monitoring by effectively managing logs through correlating logging output and source code via pattern matching, another effective approach to managing logs is to analyze and improve the way logs are generated. The challenge in log generation is the identification of the proper level of logging and analysis. A software system that logs extensively on every operation and event that occurs will yield rich information about the system but at a high cost of processing complexity, whereas a software system that logs generally on events may be limited in the amount of information they provide for the purposes of system analysis, auditing or optimization. Adaptive logging aims to achieve the right amount of logging detail so that the information provided is useful enough to serve different needs. This chapter will propose the adaptive logging framework and architecture for this purpose.

8.1 Objectives

As its name implies, the salient idea behind adaptive logging is for the logger to respond adaptively by logging data based on different signals or other environment parameters. For the logging framework to behave adaptively, a feedback control loop will be required to adjust the monitoring intensity as needed. The behaviour of the adaptive logging framework is analgous to a policeman keeping watch in a neighbourhood at night. Initially the policeman will be travelling around the neighbourhood at a low level of awareness but keeping an eye out for anything abnormal happening. Upon hearing the sound of shattering glass, for example, the policeman may become slightly more alerted and may head towards the sound to investigate further. Upon discovering the source of the sound is from a television, the policeman's awareness level decreases back to normal and resumes his regular routine. On the other hand, if during his investigation he hears a muffled cry for help followed by a gunshot, the policeman will become alerted and may even need to call for backup. Similarly, an adaptive logging framework will normally log data about a software system at a low level of awareness, until an abnormal event occurs which may trigger the framework to do more detailed logging in a specific area. Further triggers may cause the adaptive logging framework to log in even more extensive detail or return back to a low level of awareness where minimal logging is done.

The adaptive logging framework must therefore know how to process and react to a variety of triggers and events through different event handlers. Because of the wide range of events that may occur throughout the operation of a software system, event handlers should be able to register dynamically with the adaptive logging framework such that the way events are handled are dynamically modifiable to match changing logging requirements. In other words, event handlers should be able to “plug and play” into the adaptive logging framework, such that given a requirement (eg. security) an appropriate event handling scheme can be applied to the framework dynamically.

The adaptive logging framework should be aware of a few plausible concerns. Firstly, the adaptive logging framework should have minimal overhead in terms of system resources. The design and implementation of the framework should be done in a manner that minimizes its impact on the operational profile of the system as a whole. As more monitors or probes are added to the adaptive logging framework due to a higher awareness level, the signature of the adaptive logging framework should be constant or increase sub-linearly. Secondly, the feedback nature of the framework makes it prone to the snowball effect of progressive logging. For example, suppose certain issues with a network firewall application has resulted in slow network behaviour. The firewall may recognize this and trigger the adaptive logging framework to investigate the delay, but the addition of more monitors and probes into the network traffic may slow the network down even more. Therefore, event handlers must be designed in a way to prevent the snowball effect from happening with progressive logging. Lastly, the adaptive logging framework should have minimal, if any, modification to the source code of the application under monitoring.

8.2 Architectural Overview

Based on the objectives and requirements mentioned above, two architectural styles and patterns are seen to be appropriate for the adaptive logging framework. Firstly, the implicit invocation architecture style where different event logging monitors can dynamically register their interest to specific types and categories of events may be useful in establishing the plug and play architecture as well as minimizing the amount of modification needed on the application. This can be achieved by having the adaptive logger running on another process on the system and allowing the application to communicate with the logger process via system signals. Secondly, a blackboard architecture style is considered. The knowledge sources for the blackboard will be a library of *awareness policies* that are designed to raise or lower the awareness level as well as perform the appropriate logging actions based on the triggering event. The blackboard itself consists of the awareness level as well as the state of the system itself, as both of these contribute to the amount of events triggered. The application may need to be slightly annotated to emit signals as triggers to the adaptive logger process, and thus behaves as the control shell of the blackboard system. Figure 8.1 demonstrates the associations between these three components of the adaptive logging framework designed as a blackboard system.

Also, in this context, a hierarchical structure for the awareness policies where new monitors can be activated when specific system states have been reached or specific composite events have been observed will be necessary. One possibility is to define the hierarchy of awareness policy using a State design pattern, where the appropriate awareness policy is referenced and the corresponding event handler

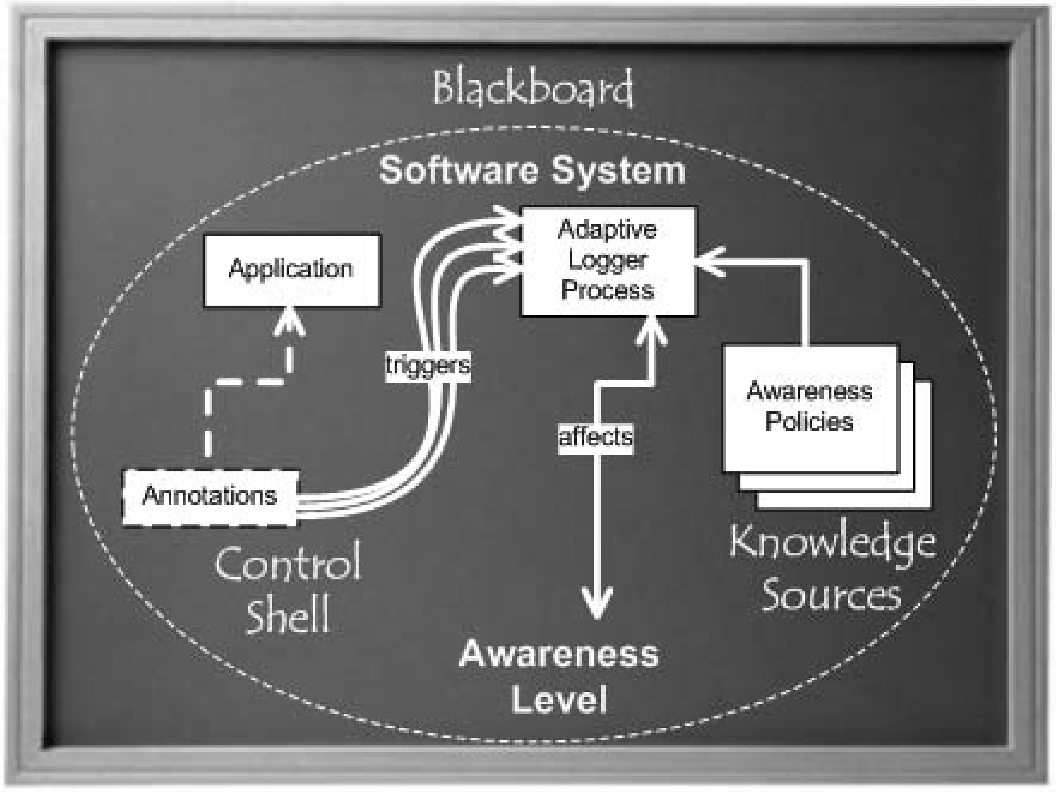


Figure 8.1: Blackboard Architectural Style for the Adaptive Logging Framework

called based on the awareness level of the system. For example, in Figure 8.2, the Adaptive Logger maintains the awareness level of the logging framework, and depending on the incoming trigger identifier and the current awareness level of the system, the appropriate awareness policy class is instantiated and its event handler is called.

An extension to this hierarchical concept is to have a corresponding hierarchy of events and triggers that maps to a hierarchy of awareness policies. Consider the hierarchy presented in Figure 8.3. At awareness level 1, which would be the lowest awareness level, only events with the triggerID of 1 will be handled by an

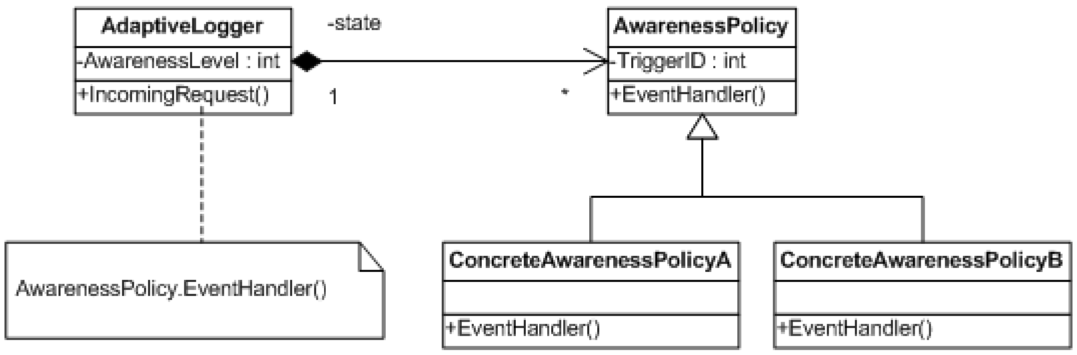


Figure 8.2: State Design Pattern for the Hierarchy of Awareness Policies

event handler, precisely that of AwarenessPolicy1. Events that correspond to a higher awareness level, which would have triggerIDs such as 1.1 or 1.1.1, will be disregarded. However, suppose the event handler of AwarenessPolicy1 raises the awareness level to 2, then the awareness policies that correspond to this awareness level such as AwarenessPolicy1.1 and AwarenessPolicy1.2 will be activated and their respective event handlers will execute based on the incoming triggers. Moreover, their parent's event handler will also be executed. For example, at awareness level 2, if the incoming event has a triggerID of 1.2, then the event handlers for AwarenessPolicy1 and AwarenessPolicy1.2 will be executed. Likewise, at awareness level 3, if the incoming event has a triggerID of 1.2.1, then the event handlers for AwarenessPolicy1, AwarenessPolicy1.2 and AwarenessPolicy1.2.1 will be executed. This hierarchical structure allows for nested subclasses to have more refined or additional actions taken to investigate the issue at hand in their event handlers.

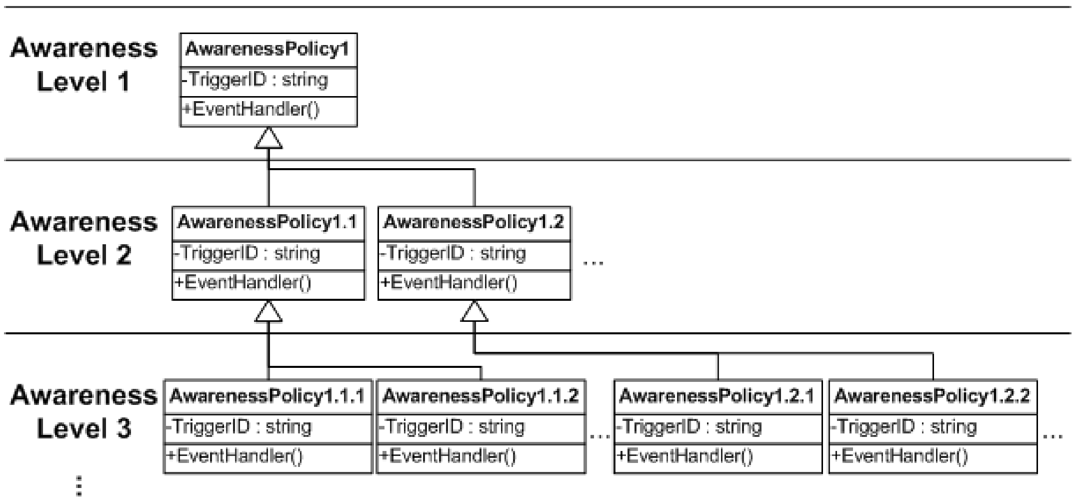


Figure 8.3: Hierarchy of Awareness Policies that Corresponds to a Hierarchy of Triggers

8.3 Design and Implementation

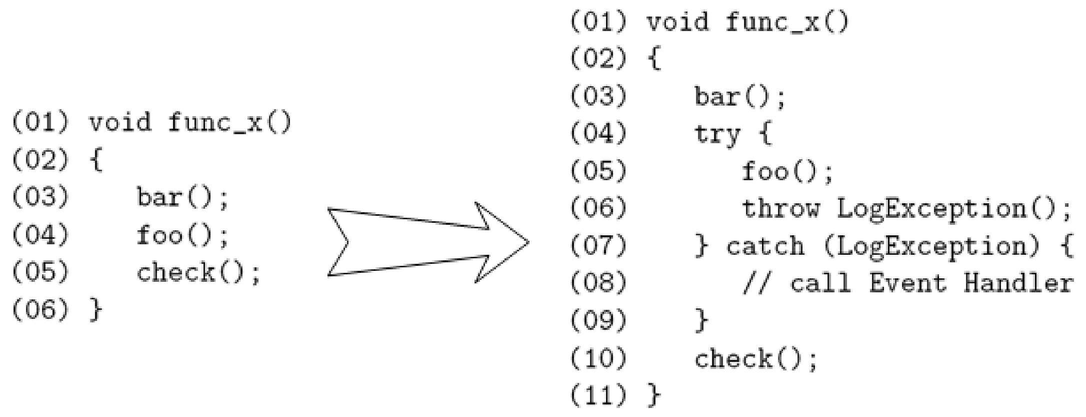
The architecture mentioned in the previous section can be roughly divided into three components, namely the annotated source code that provides the events, the adaptive logger process that manages the library of awareness policies and event handlers, and the actual awareness policies themselves. This section will discuss possible implementation approaches to each of these components.

The source code annotator component's purpose is identical to the source code annotator used for pattern matching as discussed in Chapter 6. As such, the techniques and design of the source code annotator can be applied in this situation as well. However, the design decision remains as to how much to annotate and what to annotate.

In terms of how much to annotate, there are two possible paths to pursue: the annotations can trigger all events regardless of the awareness level and leave it up to the adaptive logger to filter and discard useless signals, or alternatively, the annotations can check against the awareness level and only trigger events at the current awareness level. Both approaches have their advantages and disadvantages; the former generates a large amount of traffic for the adaptive logger to handle, whereas the latter shifts the computation to the application. Because of the objective to minimize modification to the application as well as the impact on its operational profile, the former approach is deemed more appropriate.

In terms of what to annotate, this design decision deals with the means of communiation between the application and the adaptive logger. One way for events to be generated by source code annotations is to use exceptions. Consider the code annotation presented in Figure 8.4. Suppose function `foo` is of monitoring and logging interest and has been selected for annotation. By surrounding the function call to `foo` with a try-catch block and forcing a self-defined exception to be thrown, the logic of the application will remain intact. However, a significant downside of this approach is its high operational profile and impact on the application. The execution path of the application is affected and the fact that the management of the awareness policy needs to happen as part of the application makes this approach an undesirable one.

An alternative approach is to use signals as the method of communication between the application and the adaptive logger. The adaptive logger will then be a thread or process running parallel to the application on the same machine waiting for these system signals from the application. Figure 8.5 presents the code annotation for the same piece of source code previously presented using signals.

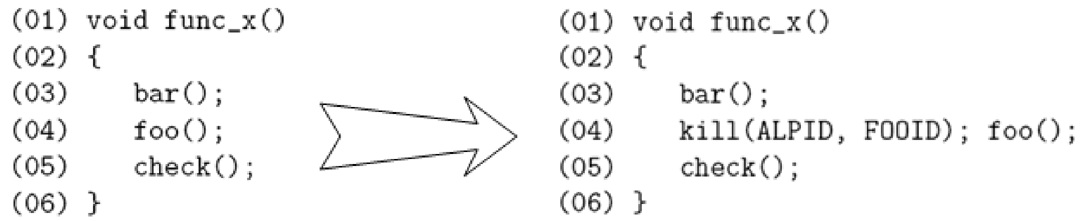


```
void func_x()
{
    bar();
    foo();
    check();
}
```

```
void func_x()
{
    bar();
    try {
        foo();
        throw LogException();
    } catch (LogException) {
        // call Event Handler
    }
    check();
}
```

Figure 8.4: Source Code Annotation with Exceptions

The `kill` command in C sends a system signal to the process with PID `ALPID` with signal number `FOOID`. The adaptive logger process will receive these signals by registering signal handlers with the system function `signal`. However, there are only 32 valid signals allowed in a UNIX system, which may not be robust enough for the adaptive logging framework. This limitation can be circumvented by using other technologies along this line of thought such as using mailboxes (`mbox`), named pipes (`mknod` or `mkfifo`) and other inter-process communication mechanisms to communicate the events to the adaptive logger.



```
void func_x()
{
    bar();
    foo();
    check();
}
```

```
void func_x()
{
    bar();
    kill(ALPID, FOOID); foo();
    check();
}
```

Figure 8.5: Source Code Annotation with Signals

As discussed in the previous section, the adaptive logger itself will follow a state design pattern. The adaptive logger process will keep track of the awareness level and based on this awareness level and the incoming signal from the application, the appropriate concrete awareness policy will be instantiated and its event handler called. To facilitate the capability to plug and play different awareness policies, the event handlers should not be hard coded as part of the adaptive logger; rather, the event handlers should be a function in the awareness policy class which is invoked by the adaptive logger via reflection. The awareness policies can therefore be modified and recompiled anytime without disrupting the adaptive logger or the application.

Chapter 9

Case Studies

This chapter will present an implementation of the policy driven software monitoring framework with the pattern matcher. The structure and layout of the framework will be described using standard UML package and class diagrams. The implementation will then be used to demonstrate the functionality of the policy driven software monitoring framework with three software applications of varying scale. A timing analysis will also be presented to evaluate the performance and scalability of the framework. This chapter will end with a brief feasibility study on the ideas presented for the Adaptive Logging Framework.

9.1 Overall Framework Structure

As mentioned in Chapter 3, the policy driven software monitoring system can be broken down into four major components, namely C Code Parser, Code

Annotator, Monitoring Policy Parser and the Monitoring Policy Pattern Matcher. Furthermore, these four functional components interact with two components that contain the domain model of the artifacts generated by the parsers and annotators, namely the monitoring policy model and the Petri Net model. Figure 9.1 presents the relationship between these different components in a UML package diagram.

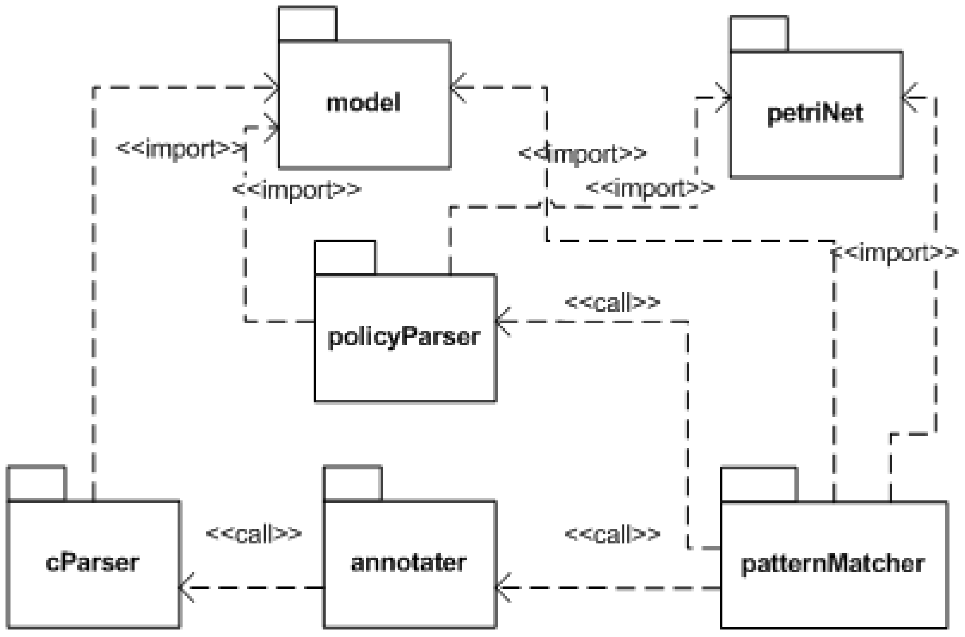


Figure 9.1: Package Diagram of the Policy Driven Software Monitoring System

The following sections will detail the classes within each of these components.

9.1.1 Monitoring Policy

The monitoring policy component contains the classes that define the domain model of a monitoring policy and is used by the source code parser, the monitoring policy concrete model and the pattern matcher to reference a monitoring policy in dynamic memory. The classes for the monitoring policy model closely reflects the basic blocks of a monitoring policy as depicted in Figure 4.1 and therefore the class diagram for the monitoring policy component will not be presented here again.

9.1.2 Petri Net

The Petri Net component contains the classes that define the domain model of the Petri Net generated by the monitoring policy parser and used by the pattern matcher. The classes for the monitoring policy model closely reflects the primary elements of a Petri Net. Figure 9.2 presents the class diagram for the Petri Net component.

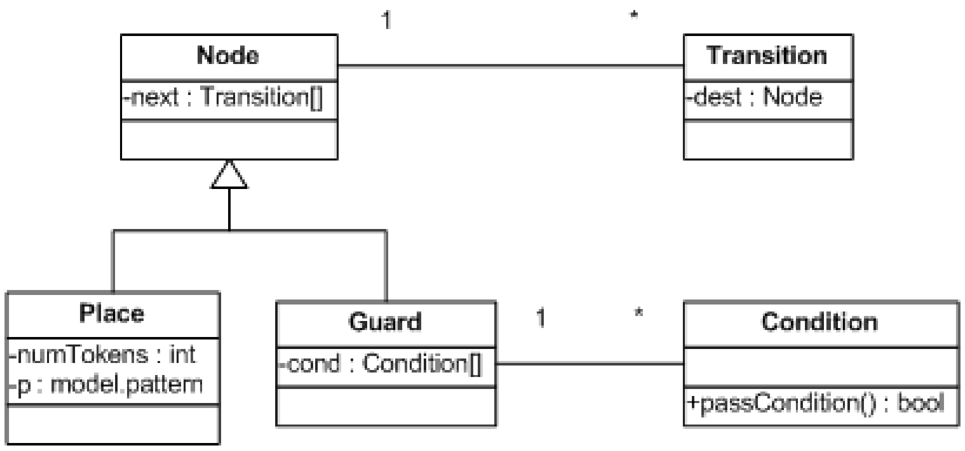


Figure 9.2: Class Diagram of the Petri Net Component

9.1.3 Monitoring Policy Parser

The Monitoring Policy Parser component contains the classes that are used to parse a textual representation of a monitoring policy into a tree representation and then transform it into a Petri Net representation. It therefore imports the monitoring policy model component and the Petri Net component. Figure 9.3 presents the class diagram for the Monitoring Policy Parser component.

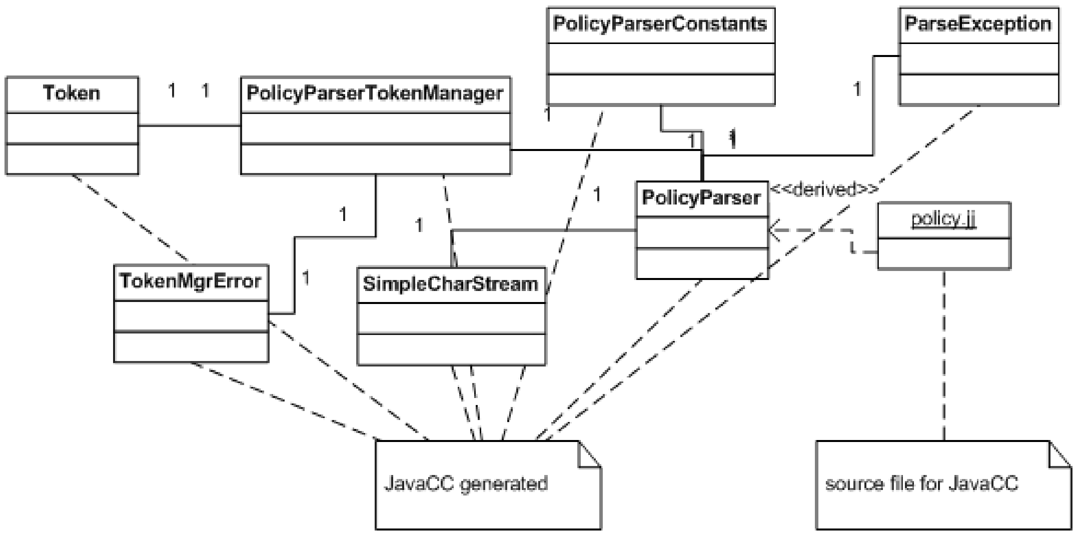


Figure 9.3: Class Diagram of the Monitoring Policy Parser Component

9.1.4 C Code Parser

The C Code Parser component contains the classes that are used to parse C source code and generate a partial Abstract Syntax Tree containing elements pertinent to a monitoring policy. It therefore imports the monitoring policy model component. Figure 9.4 presents the class diagram for the C Code Parser component.

9.1.5 Source Code Annotator

The Source Code Annotator component contains the classes that are used to annotate source code that has been parsed by the C Source Code Parser. The Source

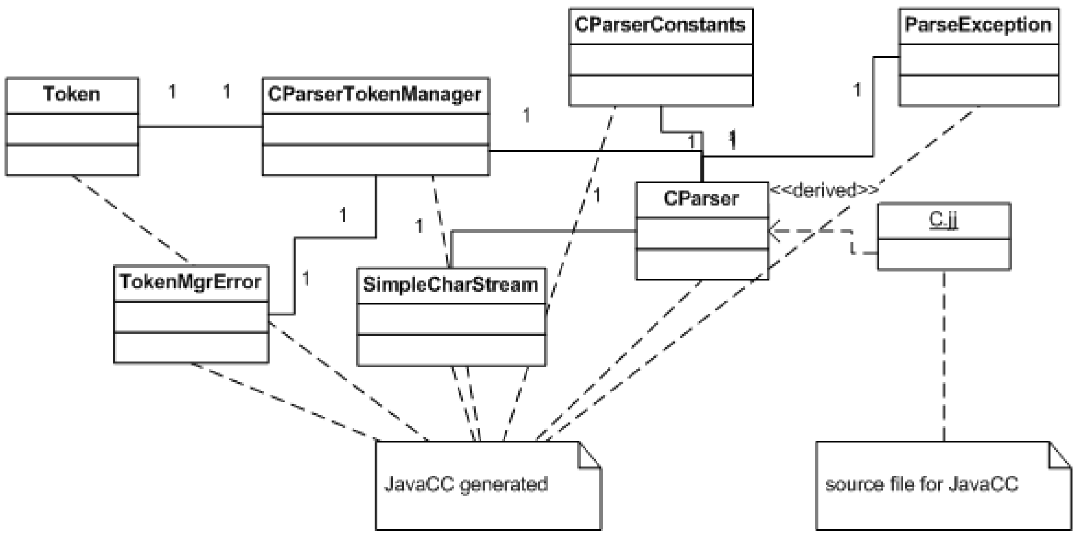


Figure 9.4: Class Diagram of the Source Code Parser Component

Code Annotator contains one class only. Figure 9.5 presents the class diagram for the Source Code Annotator component.

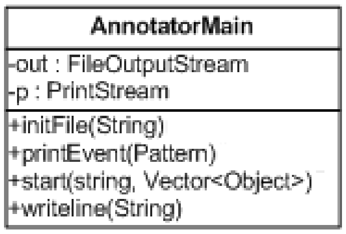


Figure 9.5: Class Diagram of the Source Code Annotator Component

9.1.6 Pattern Matcher

Lastly, the Pattern Matcher component contains the classes that take a Petri Net representation of a monitoring policy and the output of the annotated source code to locate the points for monitoring in the source code. It therefore imports both the monitoring policy model and the Petri Net model. Figure 9.6 presents the class diagram for the Pattern Matcher component.

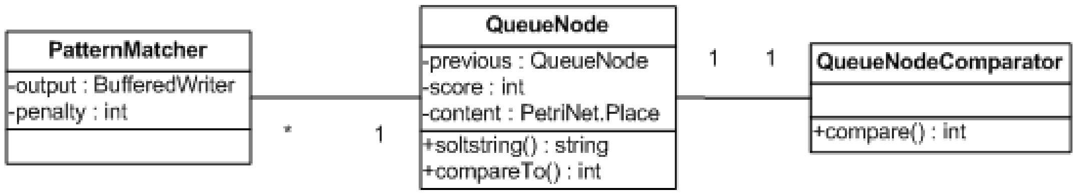


Figure 9.6: Class Diagram of the Pattern Matcher Component

9.2 Examples of the Policy Driven Software Monitoring Framework

This section will apply the presented framework to three C applications of different scale and demonstrate that the framework is scalable and is capable of handling source code of different complexity. The framework will first be applied to a simple file merger application that is publicly available at [62]. The simple file merger application combines two sorted files of strings and consists of approximately 100 lines of code. The annotated source code's output for a small sample input resulted in 124 lines of output which consists of a number of function calls and file patterns. A small snippet of the output is presented below; the complete list of output is reproduced in Appendix A.

(1) *Annotated*1175739682 FunctionCall - Function main declared at 81:5– Function stringMerge declared at 26:5@86:41
(2) *Annotated*1175739682 FilePattern -with fopen Function stringMerge declared at 26:5–model.Library@1690726,null,null@36:12

(3) *Annotated*1175739682 FilePattern -with fopen Function stringMerge declared at 26:5–model.Library@5483cd,null,null@40:12
(4) *Annotated*1175739682 FilePattern -with fopen Function stringMerge declared at 26:5–model.Library@9931f5,null,null@44:12
(5) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@49:9
(6) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@50:9
(7) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7
(8) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13
(9) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7
(10) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13
(11) ...

The first monitoring policy that is used to test the framework is a simple atomic policy as follow:

```
file_ pattern (stringMerge, fprintf, stringout, null)
```

This simple monitoring policy finds the first occurence where the function `string Merge` calls the system library `fprintf` to write to file `stringout`. The results of running the pattern matcher with the given inputs yields the following output.

```
(01) +++SOLUTION+++
(02) File-stringMerge fprintf-ing stringout at 57:7 <-->  Score: 122
(03) 1 Solution(s) found.
```

A slightly more complex monitoring policy can be used to test the accuracy and effectiveness of the framework in finding all instances of an event pattern as specified by the monitoring policy in the source code. The monitoring policy listed below finds all instances of the function call to fopen from stringMerge.

```
(* ; file_ pattern(stringMerge, fopen, null, null))
```

The output is listed below:

```
(01) +++SOLUTION+++
(02) File-stringMerge fopen-ing stringout at 44:12 <-->  Score: 122
(03) +++SOLUTION+++
(04) * <--> File-stringMerge fopen-ing stringout at 44:12 <-->
      Score: 121
(05) +++SOLUTION+++
(06) * <--> File-stringMerge fopen-ing stringout at 44:12 <-->
     Score: 120
(07) +++SOLUTION+++
(08) * <--> File-stringMerge fopen-ing stringout at 44:12 <-->
     Score: 119
(09) 4 Solution(s) found.
```

While there are only 3 fopen file_ patterns listed in the annotated source code output, a total of 4 paths are found. This is due to the use of the wildcard pattern at the start of the pattern. The first solution found, as the output path indicates, does not use the wildcard pattern at all in its match against the monitoring policy. In other words, the wildcard pattern represents nothing, and hence the first path takes a penalty from the mismatch of the first line, matches with the fopen file_ pattern found on the second line, and takes penalty for the rest of the output. This is clearly not an optimal path and is reflected by the highest score of 122 amongst all 4 solution paths found. The other 3 solution paths represent the wildcard pattern consuming the first line, the first and second lines, and the first, second and third lines of the annotated source code output respectively. These 3 solution paths matches with the first, second and third fopen file_ patterns found in the annotated source code output respectively, and hence all of the fopen file_ patterns are recovered, and all solution paths retrieved are correct.

After a positive test has been done, a negative test is also done to ensure the framework does not identify false positives if in fact there are no matches. The monitoring policy listed below is used to test the framework for this purpose.

```
 (((file_ pattern(stringMerge, fopen, null, null) +
file_ pattern(stringMerge, fprintf, null, null)) +
func_ call_ pattern(stringMerge, getline)) ;
func_ call_ pattern(main, stringMerge))
```

Since the function call from main to stringMerge is the first event pattern, there should be no paths that contains any event pattern preceeding this function call. Therefore, no solution paths should be found if this monitoring policy is used with the previous annotated source code output by the pattern matcher. The pattern matcher's output is as follows, which matches what is expected.

```
(01) 0 Solution(s) found.
```

Lastly, a complicated monitoring policy involving multiple terms is used to test the policy driven software monitoring framework's abillity to handle complexity. The monitoring policy used is listed below.

```
 ((((func_ call_ pattern(main, stringMerge) +
file_ pattern(stringMerge, fopen, null, null)) ;
file_ pattern(stringMerge, fprintf, null, null)) ;
*) ; file_ pattern(stringMerge, fclose, null, null))
```

The pattern matcher executed successfully and discovered a total of 232 solution paths. This is largely due to the different permutations possible with the wildcard pattern. Removing the wildcard pattern from the monitoring policy yields the following output, which finds the first matching instance of the event patterns specificed in the monitoring policy.

```
(01) +++SOLUTION+++
(02) FunctionCall-main calls stringMerge at 86:41 <--> File-string
     Merge fprintf-ing null at 57:7 <--> File-stringMerge fclose-ing
     null at 75:3 <-->  Score: 120
(03) +++SOLUTION+++
(04) File-stringMerge fopen-ing null at 36:12 <--> File-stringMerge
     fprintf-ing null at 57:7 <--> File-stringMerge fclose-ing null
     at 75:3 <-->  Score: 120
(05) 2 Solution(s) found.
```

Table 9.1 summarizes the different tests performed on the aforementioned inputs to verify the correctness of the policy driven software monitoring framework.

Similar experiments were performed successfully on two larger applications developed in C. The first application is a Blackjack game which has approximately 1000 lines of code, and the second application is the open source BASH shell [63] which contains over 50000 lines of code. Due to the sheer volume of output, the

Table 9.1: Summary of Tests Performed to Verify the Policy Driven Software Monitoring Framework

Test Name	**Test Description**	**Solutions Found**
Basic Test	Verify the framework is operational	1
Positive Test	Verify the framework correctly identifies all locations in the source code that matches with the monitoring policy	4
Negative Test	Verify the framework does not identify any false positive locations in the source code	0
Robustness Test	Verify the framework can handle complex monitoring policies and identify multiple solutions	232

results of these experiments will not be reproduced here, but it is important to point out the robustness of the framework in handling software applications of different scales.

9.3 Timing Analysis

This section evaluates the performance and scalability of the policy driven software monitoring framework by analyzing the time it takes for the pattern matcher to execute with varying complexity in the monitoring policy. As additional event patterns are added to the monitoring policy, its Petri Net representation will increase in size and consequently affecting the performance of the algorithm.

For the purposes of this timing analysis, the string merging program above is used with two large input files, resulting in 3075 lines of annotated source code output to match with the Petri Net. Monitoring policies of varying length and complexity are applied to the pattern matcher with this annotated source code output and the pattern matcher's execution time is recorded. In the first test, the monitoring policies will consist of only file patterns and function call patterns which are combined with a choice operator. The results are shown in Figure 9.7 below. As seen, the performance of the pattern matcher is related linearly to the number of choice expressions found in the monitoring policy.

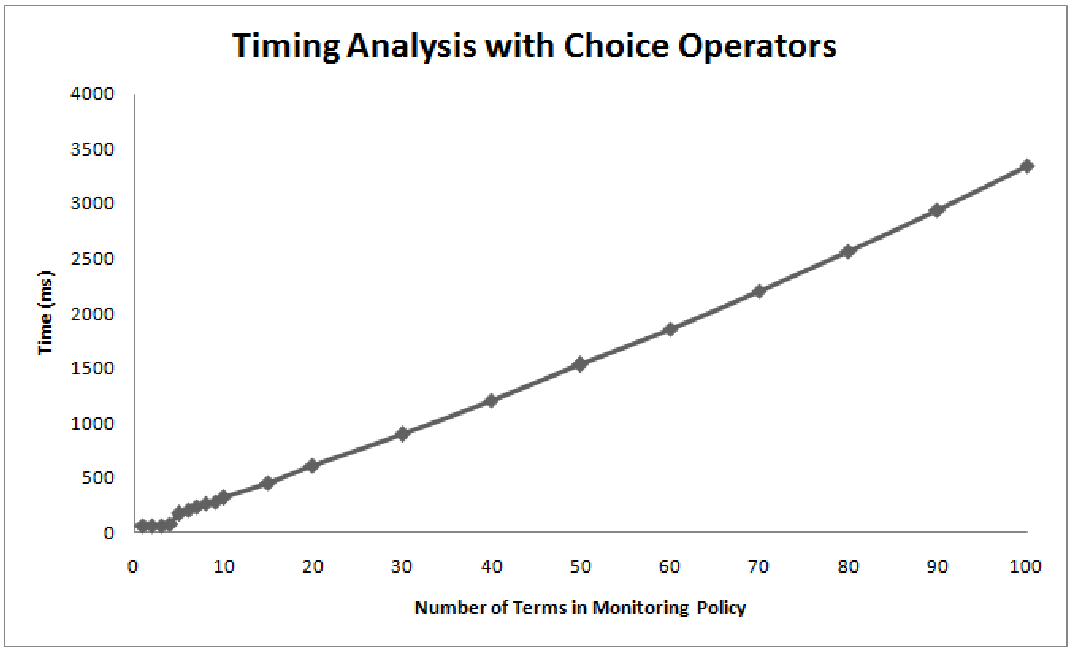


Figure 9.7: Timing Analysis of the Pattern Matcher with Varying Monitoring Policies Using Choice Operators

Similarly, a second test is performed with monitoring policies where file patterns and function call patterns are combined with a sequence operator. The results are shown in Figure 9.8 below. As seen, the performance of the pattern matcher is constant regardless of the number of sequence operators in the monitoring policy. This is because in the pattern matching algorithm, a sequence expression does not add extra nodes to the queue; in other words, the queue size is always 1. As the pattern matcher traverses through the input, the one node in the queue is compared against in each iteration and hence the performance of the pattern matcher is not affected by the number of sequence operators in the monitoring policy. On the other hand, the size of the queue is directly proportional to the number of choice expressions in the monitoring policy, as each possible choice is entered as a node in the queue, which has a direct effect on the performance of the pattern matcher in each iteration.

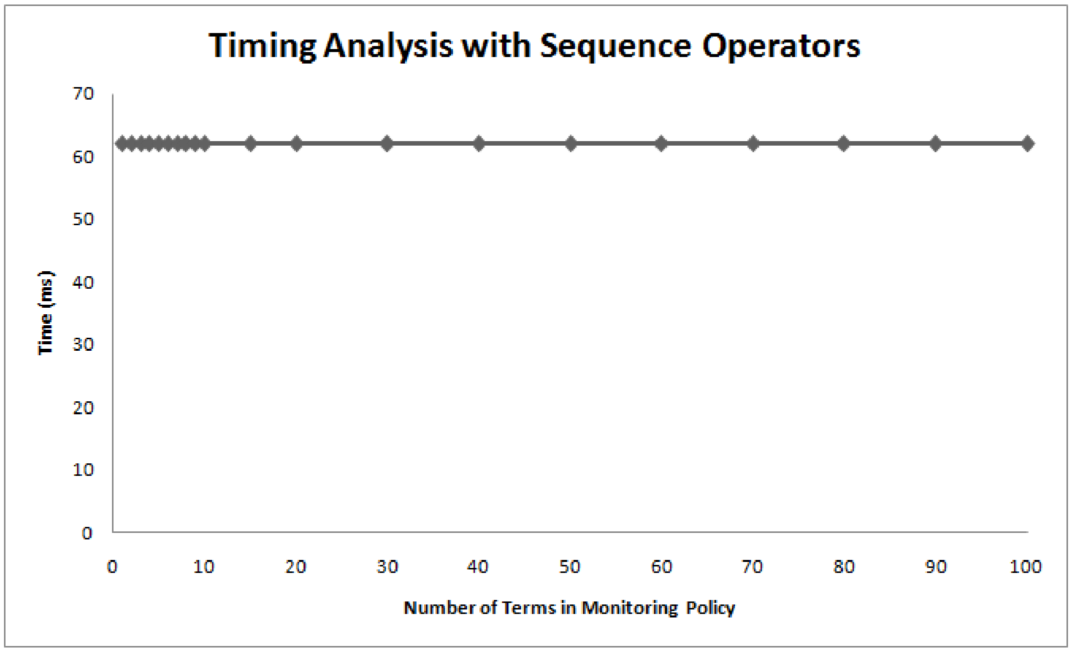


Figure 9.8: Timing Analysis of the Pattern Matcher with Varying Monitoring Policies Using Sequence Operators

The third test is performed with monitoring policies where file patterns and function call patterns are combined with a concurrent operator. Since the concurrent operator translates to a combination of sequence and choice operators, each concurrent operator adds an additional layer of complexity to the monitoring policy as each choice will map exponentially to multiple choices once the concurrent operator is expanded. Therefore, it is expected that the execution time will increase exponentially with the addition of more concurrent expressions. The results shown in Figure 9.9 is consistent with this explanation. Note the different scales used on the axis for this figure.

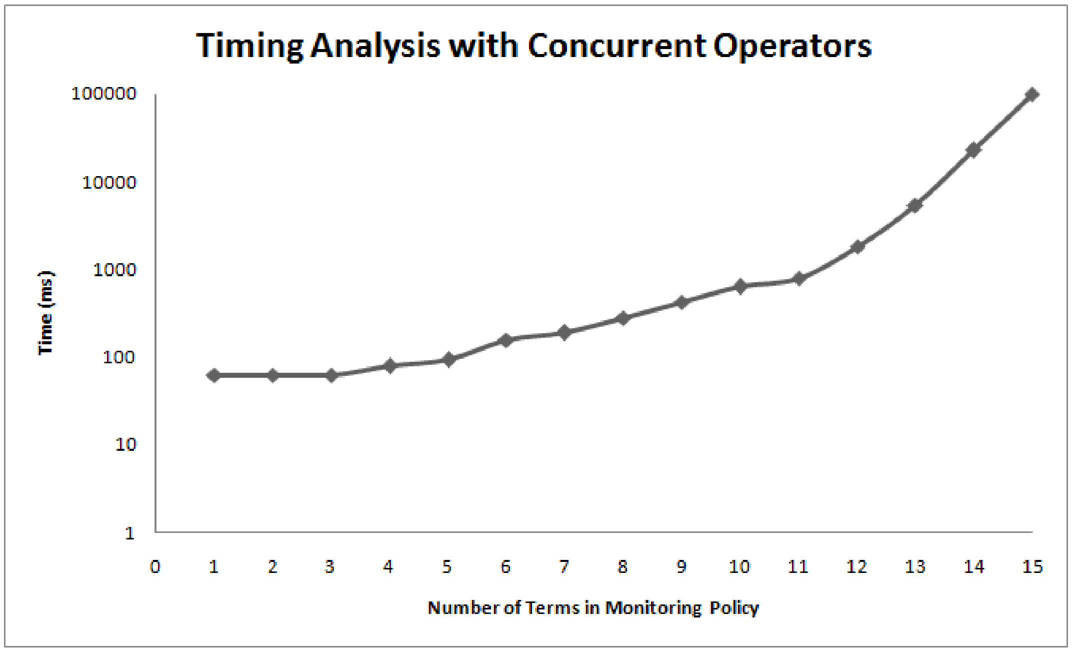


Figure 9.9: Timing Analysis of the Pattern Matcher with Varying Monitoring Policies Using Concurrent Operators

The last test is performed with the addition of a wildcard pattern at the start of the monitoring policy. Table 7.1 previously demonstrated the presence of a wildcard node in the monitoring policy adds significant complexity to the algorithm as the wildcard node is never removed from the queue and consequently its successor nodes are added to the queue each iteration. Therefore, it is expected that the addition of the wildcard pattern to the monitoring policy will significantly affect the performance of the pattern matcher. For this analysis, the wildcard pattern is combined with other file patterns with sequence operators, as is the typical usage of the wildcard pattern. The results are shown in Figure 9.10 below. While the performance is significantly slower when compared against the tests without a wildcard pattern, the performance of the pattern matcher still increases linearly with the number of patterns to match in the monitoring policy.

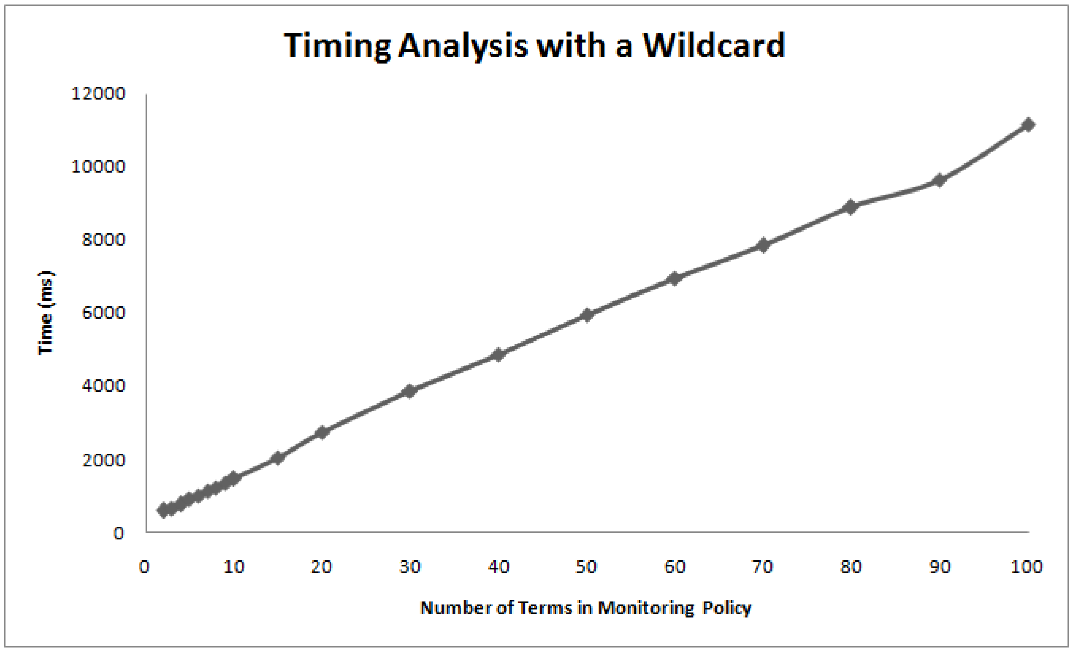


Figure 9.10: Timing Analysis of the Pattern Matcher with Varying Monitoring Policies with a Wildcard Pattern

9.4 Feasiblity Study of Adaptive Logging Framework

To investigate the feasibility of the proposed adaptive logging framework in Chapter 8, two simple experiments were performed to determine the functional feasibility of using signals [64] and named pipes [65] as the interprocess communication protocol between the application and the adaptive logger. Both experiments involve two processes, namely a sender and a receiver, establishing a uni-directional (half duplex) channel of communication. Both experiments are conducted in a UNIX environment with source code written in C.

The first experiment deals with using signals to communicate between two processes. The sender and receiver processes execute code from `sigsend.c` and `sigrcv.c` respectively. The following is the source code listing for `sigsend.c`. It takes the PID of the receiver as an input argument and sends three signals via the system command `kill` to the receiver process.

```
// sigsend.c
#include <stdio.h>
#include <signal.h>

int main(int argc, char *argv[])
{
     int pid;

     if (argc != 2) {
        printf("Usage : %s <pid of receiver>\n", argv[0]);
        exit (1);
      }

      pid = atoi(argv[1]);

      printf("SENDER: sending 15 to %d\n", pid);
      kill(pid, 15);
      sleep(3); /* pause for 3 secs */

      printf("SENDER: sending 16 to %d\n", pid);
      kill(pid, 16);
      sleep(3); /* pause for 3 secs */

      printf("SENDER: sending 17 to %d\n", pid);
      kill(pid, 17);
      sleep(3);
}
```

The following is the source code listing for `sigrcv.c`. It registers three signal handlers for signal numbers 15, 16 and 17. The signal handler for 17 exits the process.

```
// sigrcv.c
#include <stdio.h>
#include <signal.h>

// Routines called upon sigtrap
void f15();
void f16();
void f17();

int main()
{
     // register signal handlers
       signal(15,f15);
       signal(16,f16);
       signal(17,f17);
       for(;;); // loop forever
}

void f15() {
   printf("RCVproc: I have received a signal 15\n");
}

void f16() {
   printf("RCVproc: I have received a signal 16\n");
}

void f17() {
  printf("RCVproc: I have received a signal 17. Exit now.\n");
  exit(0);
}
```

The following program output demonstrates that the receiver has successfully received the three signals sent from the sender and executes the corresponding event handler in the correct order.

```
bash-2.05b$ ./sigrcv &
[1] 20721
bash-2.05b$ ./sigsend 20721
SENDER: sending 15 to 20721
RCVproc: I have received a signal 15
SENDER: sending 16 to 20721
RCVproc: I have received a signal 16
SENDER: sending 17 to 20721
RCVproc: I have received a signal 17. Exit now.
[1]+  Done                    ./sigrcv
```

A significant limitation of sending system signals is that signal numbers can only range from 0 to 31. For a robust adaptive logging framework this may not be acceptable, and hence another interprocess communication method is explored.

The second experiment uses named pipes as a method of interprocess communication. Aside from the sender and receiver which are named `np_ sender.c` and `np_ receiver.c` respectively, there is also a header file named `halfduplex.h` that contains the location of the half duplex named pipe and the maximum buffer size. The following is the listing for the header file.

```
#define HALF_DUPLEX "/tmp/halfduplex"
#define MAX_BUF_SIZE 255
```

The following is the source code listing for `np_ receiver.c`. It creates the named pipe and then opens it for reading, and outputs to the screen what was read.

```
// np_receiver.c
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include "halfduplex.h" // this header file specifies the location of the named pipe

int main(int argc, char *argv[])
{
    int fd, ret_val, count, numread;
    char buf[MAX_BUF_SIZE];

    // Create the named pipe
    ret_val = mkfifo(HALF_DUPLEX, 0666);

    if ((ret_val == -1) && (errno != EEXIST)) {
        perror("Error creating the named pipe!");
        exit (1);
    }

    // Open the pipe for reading
    fd = open(HALF_DUPLEX, O_RDONLY);

    // Read from the pipe
    numread = read(fd, buf, MAX_BUF_SIZE);

    buf[numread] = '\0';

    // Output what was read
    printf("Named Pipe Reader: Read From the pipe : %s\n",
     buf);
    printf("Named Pipe Reader: Total Characters Read : %d\n",
     numread);
}
```

Lastly, the following is the source code listing for `np_ sender.c`. It takes one parameter, which is the string to be sent to the receiver through the named pipe, and writes it to the named pipe.

```
// np_sender.c
#include <stdio.h>
#include <fcntl.h>
#include "halfduplex.h" // this header file specifies the
location of the named pipe

int main(int argc, char *argv[])
{
    int fd;

    if (argc != 2) {
        printf("Usage : %s <string to be sent to the receiver>
         \n", argv[0]);
        exit (1);
    }

    // Open the pipe for writing
    fd = open(HALF_DUPLEX, O_WRONLY);

    // Write to the pipe
    write(fd, argv[1], strlen(argv[1]));
}
```

The following program output demonstrates that the receiver has successfully received the string written to the named pipe by the sender. It is therefore possible to expand on this method as a channel of communication between the application and the adaptive logger.

```
bash-2.05b$ ./np_receiver &
[1] 21142
bash-2.05b$ ./np_sender feasibility_test
Named Pipe Reader: Read From the pipe : feasibility_test
Named Pipe Reader: Total Characters Read : 16
[1]+  Exit 1                  ./np_receiver
bash-2.05b$
```

Chapter 10

Conclusions and Future Work

Software monitoring and logging is one of the most important tools a software engineer has when faced with auditing or analysing a software system. However, one of the challenges software engineers face in software re-engineering is the difficulty in effectively monitoring a system, managing its logs and cross referencing them with source code. This thesis aimed to address this issue by providing a framework that enables pattern matching between a software log and source code based on a monitoring policy. It further expands upon this work by proposing an adaptive logging framework that will greatly improve the quality of log management.

10.1 Thesis Overview and Findings

This thesis has proposed a policy driven monitoring architecture, which consists of a domain model of event expressions that are used to specify the monitoring policy, a monitoring policy parser, a source code annotator and a pattern matcher. Firstly, the objects to be monitored are described by a monitoring policy expression that is used to represent important event patterns. The monitoring policy expressions are able to define objects of monitoring interest both logically and temporally through a combination of event expressions, which are represented after parsing as a binary tree, and then consequently transformed into a Petri Net representation.

Secondly, the source code annotator parses the C source code and generates a partial Abstract Syntax Tree. The annotator then makes annotations to the C source code based on the results from the source code parser. Executing the source code with the annotations will yield output that reveals the program's events with respect to the objects that need to be logged.

The emitted events and the Petri Net representation of the monitoring policy together serves as the input to the Pattern Matcher. The pattern matcher associates a monitoring policy with source code by matching the annotated source code output against the Petri Net like an automaton. The underlying algorithm involves a branch-and-bound search from the start to the end place of the Petri Net, but the algorithm can be improved by adapting concepts from Artificial Intelligence literature such as using pruning, A* search and dynamic programming to guarantee optimality and performance. The pattern matcher is particularly useful when a software engineer is trying to pinpoint the location in the source code that may cause performance or security problems when the only knowledge regarding the issue is a sequence of events that happen dynamically in the operation of the application.

The thesis then discusses extensions to the policy driven software monitoring framework by outlining a proposal for an adaptive logging framework. The adaptive logging framework dynamically and autonomously adjusts the logging level of the system so that the appropriate amount of information is logged. The adaptive logging framework comprises of a feedback loop that determines the logging level of the system by signals and events generated by the application and transferred via a form of interprocess communication mechanism as well as a hierarchy of awareness policies that dictate how the adaptive logger should react based on the input and how the logging level of the system should be adjusted. Further experiments were performed to demonstrate that named pipes is a feasible and scalable method of interprocess communication for the adaptive logging framework.

10.2 Future Work

The framework and ideas presented in this thesis has opened the doors to a few avenues of future work. Aside from the optimizations in the pattern matching algorithm mentioned in Chapter 7, improvements can be made to the monitoring policy in terms of expanding the amount of event patterns available as well as allowing a more detailed specification of the function parameters in the monitoring policy. Furthermore, an investigation on the possibility of having if-conditions in the monitoring policy will prove to be interesting; the implementation will not deviate significantly from the proposed framework as the Guard conditions of the Petri Net can be used to handle the if-conditions.

The adaptive logging framework is also inspirational on a number of possible paths to pursue for future work in this subject area. Right now the proposed framework contains awareness policies that are stagnant; a more powerful and robust architecture will involve the design and development of complex composite awareness policies that can embed conditions dictating when policies will be triggered. These complex composite awareness policies will take full advantage of object oriented concepts such as inheritance in the taxonomy of awareness policies.

Lastly, the monitoring techniques presented in this paper are all intrusive techniques which require modification and recompilation of the source code, which may be a time consuming process for large scale legacy systems. The investigation of non-intrusive monitoring techniques by taking note of the application's circumstantial surroundings such as memory usage and other operative system primitives will be a significant contribution to this area of study. Such work will be especially useful when the software engineer is not provided with the source code to the application and must perform black box monitoring and auditing.

Appendix A

Program Output for Example in Case Study

```
(1) *Annotated*1175739682 FunctionCall - Function main declared at 81:5– Func-
    tion stringMerge declared at 26:5@86:41
(2) *Annotated*1175739682 FilePattern -with fopen Function stringMerge de-
    clared at 26:5–model.Library@1690726,null,null@36:12
(3) *Annotated*1175739682 FilePattern -with fopen Function stringMerge de-
    clared at 26:5–model.Library@5483cd,null,null@40:12
(4) *Annotated*1175739682 FilePattern -with fopen Function stringMerge de-
    clared at 26:5–model.Library@9931f5,null,null@44:12
(5) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–
    Function getline declared at 10:5@49:9
(6) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–
    Function getline declared at 10:5@50:9
(7) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge de-
    clared at 26:5–model.Library@1f1fba0,null,null@57:7
(8) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–
    Function getline declared at 10:5@58:13
(9) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge de-
    clared at 26:5–model.Library@1f1fba0,null,null@57:7
```

(10) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13
(11) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7
(12) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13
(13) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@19ee1ac,null,null@54:7
(14) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@55:13
(15) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@19ee1ac,null,null@54:7
(16) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@55:13
(17) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@19ee1ac,null,null@54:7
(18) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@55:13
(19) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@19ee1ac,null,null@54:7
(20) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@55:13
(21) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@19ee1ac,null,null@54:7
(22) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@55:13
(23) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7
(24) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13
(25) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7
(26) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13
(27) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7
(28) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13
(29) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge de-

clared at 26:5–model.Library@1f1fba0,null,null@57:7

(30) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13

(31) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7

(32) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13

(33) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7

(34) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13

(35) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7

(36) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13

(37) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7

(38) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13

(39) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7

(40) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13

(41) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7

(42) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13

(43) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7

(44) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13

(45) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7

(46) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13

(47) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7

(48) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13

(49) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7
(50) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13
(51) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7
(52) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13
(53) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@19ee1ac,null,null@54:7
(54) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@55:13
(55) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7
(56) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13
(57) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7
(58) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13
(59) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7
(60) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13
(61) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@19ee1ac,null,null@54:7
(62) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@55:13
(63) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@19ee1ac,null,null@54:7
(64) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@55:13
(65) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7
(66) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13
(67) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7
(68) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–

Function getline declared at 10:5@58:13

(69) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7

(70) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13

(71) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7

(72) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13

(73) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7

(74) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13

(75) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@19ee1ac,null,null@54:7

(76) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@55:13

(77) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@19ee1ac,null,null@54:7

(78) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@55:13

(79) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@19ee1ac,null,null@54:7

(80) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@55:13

(81) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7

(82) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13

(83) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@19ee1ac,null,null@54:7

(84) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@55:13

(85) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@19ee1ac,null,null@54:7

(86) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@55:13

(87) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@19ee1ac,null,null@54:7

(88) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@55:13
(89) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7
(90) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13
(91) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7
(92) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13
(93) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7
(94) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13
(95) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@19ee1ac,null,null@54:7
(96) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@55:13
(97) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@19ee1ac,null,null@54:7
(98) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@55:13
(99) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7
(100) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13
(101) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7
(102) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13
(103) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7
(104) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13
(105) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge declared at 26:5–model.Library@1f1fba0,null,null@57:7
(106) *Annotated*1175739682 FunctionCall - Function stringMerge declared at 26:5–Function getline declared at 10:5@58:13
(107) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge de-

```
      clared at 26:5–model.Library@19ee1ac,null,null@54:7
(108) *Annotated*1175739682 FunctionCall - Function stringMerge declared at
      26:5–Function getline declared at 10:5@55:13
(109) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge de-
      clared at 26:5–model.Library@19ee1ac,null,null@54:7
(110) *Annotated*1175739682 FunctionCall - Function stringMerge declared at
      26:5–Function getline declared at 10:5@55:13
(111) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge de-
      clared at 26:5–model.Library@1f1fba0,null,null@57:7
(112) *Annotated*1175739682 FunctionCall - Function stringMerge declared at
      26:5–Function getline declared at 10:5@58:13
(113) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge de-
      clared at 26:5–model.Library@1f1fba0,null,null@57:7
(114) *Annotated*1175739682 FunctionCall - Function stringMerge declared at
      26:5–Function getline declared at 10:5@58:13
(115) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge de-
      clared at 26:5–model.Library@19ee1ac,null,null@54:7
(116) *Annotated*1175739682 FunctionCall - Function stringMerge declared at
      26:5–Function getline declared at 10:5@55:13
(117) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge de-
      clared at 26:5–model.Library@19ee1ac,null,null@54:7
(118) *Annotated*1175739682 FunctionCall - Function stringMerge declared at
      26:5–Function getline declared at 10:5@55:13
(119) *Annotated*1175739682 FilePattern -with fprintf Function stringMerge de-
      clared at 26:5–model.Library@14b7453,null,null@70:7
(120) *Annotated*1175739682 FunctionCall - Function stringMerge declared at
      26:5–Function getline declared at 10:5@71:11
(121) *Annotated*1175739682 FilePattern -with fclose Function stringMerge de-
      clared at 26:5–model.Library@c21495,null,null@75:3
(122) *Annotated*1175739682 FilePattern -with fclose Function stringMerge de-
      clared at 26:5–model.Library@1d5550d,null,null@76:3
(123) *Annotated*1175739682 FilePattern -with fclose Function stringMerge de-
      clared at 26:5–model.Library@c2ea3f,null,null@77:3
(124) We have 57 merged records
```

Bibliography

[1] M. Mansouri-Samani and M. Sloman, "A configurable event service for distributed systems," in *Proceedings of the Third International Conference on Configurable Distributed Systems*, pp. 210–217, ICCDS, 6–8 May 1996.

[2] J. Sosnowski and M. Poleszak, "On-line monitoring of computer systems," in *Proceedings of the Third IEEE International Workshop on Electronic Design, Test and Applications*, DELTA, 17–19 January 2006.

[3] R. Gwadera, M. Atallah, and W. Szpankowski, "Reliable detection of episodes in event sequences," in *Proceedings of the Third IEEE International Conference on Data Mining*, pp. 67–74, ICDM, 19–22 November 2003.

[4] E. Bertino, E. Ferrari, and G. Guerrini, "An approach to model and query event-based temporal data," in *Proceedings of the Fifth International Workshop on Temporal Representation and Reasoning.*

[5] N. M. Goldman, "Smiley – an interactive tool for monitoring inter-module function calls," in *Proceedings of the 8th International Workshop on Program Comprehension*, pp. 109–118, IWPC, 10–11 June 2000.

[6] M. Sefika, A. Sane, and R. H. Campbell, "Monitoring compliance of a software system with its high-level design models," in *Proceedings of the 18th International Conference on Software Engineering*, pp. 387–396, ICSE, 25–30 March 1996.

[7] A. Ulrich, H. Hallal, A. Petrenko, and S. Boroday, "Verifying trustworthiness requirements in distributed systems with formal log-file analysis," in *Proceedings of the 36th Annual Hawaii International Conference on System Sciences*, 6–9 Jan 2003.

[8] A. Kishon, P. Hudak, and C. Consel, “Monitoring semantics: a formal framework for specifying, implementing, and reasoning about execution monitors,” *ACM SIGPLAN Notices*, vol. 26, pp. 338–352, June 1991.

[9] A. W. Moore, A. J. McGregor, and J. W. Breen, “A comparison of system monitoring methods, passive network monitoring and kernel instrumentation,” *ACM SIGOPS Operating Systems Review*, vol. 30, pp. 16–38, January 1996.

[10] W. N. Robinson, “Monitoring software requirements using instrumented code,” in *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*, p. 276, HICSS, 2002.

[11] T.-T. Y. Lin and D. P. Siewiorek, “Error log analysis: Statistical modeling and heurisitc trend analysis,” *IEEE Transactions on Reliability*, vol. 39, pp. 419–432, October 1990.

[12] J. Bowring, A. Orso, and M. J. Harrold, “Monitoring deployed software using software tomography,” *ACM SIGSOFT Software Engineering Notes*, vol. 28, pp. 2–9, January 2003.

[13] C. Peng, B. Wu, and X. Sun, “Test by distributed monitoring,” in *Proceedings of the Eighth Asian Test Symposium*, pp. 218–223, ATS, 16–18 November 1999.

[14] W. N. Robinson, “Implementing rule-based monitors within a framework for continuous requirements monitoring,” in *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, HICSS, 3–6 January 2005.

[15] K. Kent and M. Souppaya, “Guide to computer security log management,” Tech. Rep. 800-92, Nataional Institute of Standards and Technology, September 2006.

[16] Sun Microsystems, “Dtrace tool.” `http://www.sun.com/bigadmin/content/dtrace/`.

[17] Computer Associates International, Inc., *Unicenter Network and Systems Management–Inside Event Management and Alert Management, r11.*

[18] Microsoft Corporation, “Microsoft operations manager.” `http://www.microsoft.com/mom/default.mspx`.

[19] Eclipse Foundation Inc., “Eclipse birt.” `http://www.eclipse.org/birt/phoenix/`.

[20] Eclipse Hyades Project, "Eclipse test and performance tools platform." `http://www.eclipse.org/tptp/index.html`.

[21] IBM Rational Software Architect, "Determining problems in distributed applications." `http://publib.boulder.ibm.com/infocenter/rtnlhelp/v6r0m0/index.jsp?topic=/org.eclipse.hyades.log.ui.doc.user/concepts/ceautcom.htm`.

[22] P. Panchamukhi, "Kernel debugging with kprobes." `http://www-128.ibm.com/developerworks/library/l-kprobes.html?ca=dgr-lnxw42Kprobe`.

[23] F. C. Eigler, "Systemtap tutorial." `http://sourceware.org/systemtap/tutorial/systemtap.html`.

[24] IBM Corporation, "Common base event." `http://www-128.ibm.com/developerworks/library/specification/ws-cbe/`.

[25] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, pp. 41–50, January 2003.

[26] R. Barga, S. Chen, and D. Lomet, "Improving logging and recovery performance in phoneix/app," in *Proceedings of the 20th International Conference on Data Engineering*, pp. 486–497, ICDE, 30 March – 2 April 2004.

[27] A. Kongmunvattana and N.-F. Tzeng, "Lazy logging and prefetch-based crash recovery in software distributed shared memory systems," in *Proceedings of the 13th International and 10th Symposium on Parallel and Distributed Processing*, pp. 399–406, IPPS, 12–16 April 1999.

[28] A. Kongmunvattana and N.-F. Tzeng, "Logging and recovery in adaptive software distributed shared memory systems," in *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, pp. 202–211, RELDIS, 19–22 October 1999.

[29] J. H. Andrews and Y. Zhang, "Broad-spectrum studies of log file analysis," in *Proceedings of the 2000 International Conference on Software Engineering*, pp. 105–114, ICSE, 4–11 June 2000.

[30] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *Proceedings of the Third IEEE Workshop on IP Operations and Management*, pp. 119–126, IPOM, 1–3 October 2003.

[31] J. Stearley, "Towards informatic analysis of syslogs," in *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pp. 309–318, 20–23 September 2004.

[32] Y.-C. Lin and P. Hadingham, "Tracking frequent traversal areas in a web site via log analysis," in *Seventh International Conference on Parallel and Distributed Systems: Workshops*, pp. 321–325, PADSW, 4–7 July 2000.

[33] T. Koch, A. Ardö, and K. Golub, "Browsing and searching behaviour in the renardus web service: A study based on log analysis," in *Proceedings of the 2004 Joint ACM/IEEE Conference on Digital Libraries*, p. 378, 7–11 June 2004.

[34] A. Maeda, K. Sugiyama, and K. Mase, "Log analyses of human behaviour on interactive amusement media," in *Proceedings of the Fourth International Conference on Knowledge-Based Intelligent Engineering Systems and Allied Technologies*, vol. 1, pp. 229–232, KES, 30 Aug – 1 Sept 2000.

[35] L. Rostad and O. Edsberg, "A study of access control requirements for healthcare systems based on audit trails from access logs," in *Proceedings of the 22nd Annual Computer Security Applications Conference*, pp. 175–186, ACSAC, December 2006.

[36] D. E. Knuth, *The Art of Computer Programming*, vol. 1. Reading, Massachusetts: Addison-Wesley Publishing Company, 2nd edition ed., 1973.

[37] J. ichi Aoe, ed., *Computer Algorithms: String Pattern Matching Strategies.* Los Alamitos, CA, USA: IEEE Computer Society Press, 1994.

[38] G. Davies and S. Bowsher, "Algorithms for pattern matching," in *Software–Practice and Experience*, vol. 16, pp. 575–601, John Wiley and Sons, Ltd., June 1986.

[39] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal of Computing*, vol. 6, pp. 323–350, June 1977.

[40] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, pp. 62–72, October 1977.

[41] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987.

[42] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Communications of the ACM*, vol. 18, pp. 333–340, June 1975.

[43] K. Sitaraman, A. Ejnioui, and N. Ranganathan, "A parallel algorithm and architecture for object recongition in images," in *Proceedings of the 2003 IEEE International Workshop on Computer Architectures for Machine Perception*, CAMP, 12–16 May 2003.

[44] P. A. V. Hall and G. R. Dowling, "Approximate string matching," *ACM Computing Surveys*, vol. 12, pp. 381–402, December 1980.

[45] C. M. Hoffmann and M. J. O'Donnell, "Pattern matching in trees," *Journal of the ACM*, vol. 29, pp. 68–95, January 1982.

[46] H.-T. L. Wuu, "A simple tree pattern-matching algorithm." `http://citeseer.ist.psu.edu/616969.html`.

[47] S. R. Kosaraju, "Efficient tree pattern matching," in *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pp. 178–183, SFCS, 30 Oct – 1 Nov 1989.

[48] M. Dubiner, Z. Galil, and E. Magen, "Faster tree pattern matching," in *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, vol. 1, pp. 145–150, FSCS, 22-24 Oct 1990.

[49] G. Valiente, "On the algorithm of Berztiss for tree pattern matching," in *Proceedings of the Fifth Mexican International Conference in Computer Science*, pp. 43–49, 2004.

[50] A. Ejnioui and N. Ranganathan, "Systolic algorithms for tree pattern matching," in *Proceedings of the 1995 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 650–655, ICCD, 2–4 Oct 1995.

[51] J. T. L. Wang, K. Zhang, K. Jeong, and D. Shasha, "A tool for tree pattern matching," in *Proceedings of the Third International Conference on Tools for Artificial Intelligence*, pp. 436–444, TAI, 10–13 Nov 1991.

[52] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang, "Code generation using tree matching and dynamic programming," *ACM Transactions on Programming Languages and Systems*, vol. 11, pp. 491–516, October 1989.

[53] A. V. Aho and S. C. Johnson, "Optimal code generation for expression trees," *Journal of the ACM*, vol. 23, no. 3, pp. 488–501, 1976.

[54] T.-S. Chen, F. Lai, and R.-J. Shang, "A simple tree pattern matching algorithm for code generator," in *Proceedings of the Nineteenth Annual International Computer Software and Applications Conference*, pp. 162–167, COMPSAC, 9–11 Aug 1995.

[55] R. Ramesh and I. V. Ramakrishnan, "Nonlinear pattern matching in trees," *Journal of the ACM*, vol. 39, no. 2, pp. 295–316, 1992.

[56] D. E. Denning, "An intrusion-detection model," *IEEE Transactions on Software Engineering*, vol. SE-13, pp. 222–232, February 1987.

[57] A. P. Moore, R. J. Ellison, and R. C. Linger, "Attack modeling for information security and survivability," Technical Note CMU/SEI-2001-TN-001, Carnegie Mellon University and Software Engineering Institute, March 2001.

[58] S.-P. Shieh and V. D. Gligor, "On a pattern-oriented model for intrusion detection," *IEEE Transactions on Knowledge and Data Engineering*, vol. 9, pp. 661–667, July/August 1997.

[59] S. Dharmapurkar and J. W. Lockwood, "Fast and scalable pattern matching for network intrusion detection systems," *IEEE Journal on Selected Areas in Communications*, vol. 24, pp. 1781–1792, October 2006.

[60] D. Xu and K. E. Nygard, "Theat-driven modeling and verification of secure software using aspect-oriented petri nets," *IEEE Transactions on Software Engineering*, vol. 32, pp. 265–278, April 2006.

[61] Java.Net, "JavaCC Home." `https://javacc.dev.java.net/`.

[62] G. Ingargiola, "Some simple C programs." `http://www.cis.temple.edu/~ingargio/cis71/code/`.

[63] Free Software Foundation, "BASH - GNU Project." `http://www.gnu.org/software/bash/`.

[64] L. Frye, "Sigtalk." `http://faculty.kutztown.edu/frye/CIS352/Examples/sigTalk.c`.

[65] F. Faruqui, "Introduction in interprocess communication using named pipes." `http://developers.sun.com/solaris/articles/named_pipes.html`.