TROUBLESHOOTING INTERACTIVE COMPLEXITY BUGS

BY

MOHAMMAD MAIFI HASAN KHAN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Doctoral Committee:

Professor Tarek F. Abdelzaher, Chair and Director of Research
Professor Jiawei Han
Professor Lui Sha
Dr. Jie Liu, Microsoft Research
Abstract

The term “interactive complexity” was introduced by Charles Perrow in his famous book *Normal Accidents: Living with High-Risk Technologies* [1]. He used the term to describe the interacting tendency of systems with large number of components. He argued that, in systems with large number of components, multiple failures often interact in some unexpected way, leading to catastrophic failures in systems such as planes or nuclear power plants. He also suggested that with increasing interactive complexity and tight coupling, unexpected interactions of failures are bound to happen. Indeed, with the proliferation of Internet enabled cheap embedded devices with built in sensors and actuators (e.g., smart phones, smart appliances), the physical world is increasingly becoming an integral part of the logical world of computation. As computing systems are becoming much more interactive and responsive to the surrounding physical environments, it is becoming increasingly difficult to test such systems to full extent before deployment in real world. Hence, due to increased interactive complexity and tight coupling between physical and logical world, such systems often fail or preform poorly once deployed in real life. Unintended interactions among various system components, or across computing systems and physical environments are often to blame for the problem. With this growing trend, the bugs that arise due to interaction among different distributed components across multiple nodes are likely to get worse, and are going to affect the reliability of the system significantly. This calls for new tools and techniques to troubleshoot future software systems.
In this dissertation, we address this significant challenge of troubleshooting interactive complexity bugs in emerging cyber-physical systems using data mining techniques. More specifically, we applied discriminative sequence mining algorithm to isolate chains of events (not necessarily contiguous) that is causally correlated to failure by analyzing system logs. In the first part of our thesis, using our tool, we successfully identified multiple bugs in various real systems such as multi-channel MAC (medium access control) layer protocol for wireless sensor network [2], kernel level race condition bug in the LiteOS operating system, and corner case design flaw in the directed diffusion protocol [3]. Next, we extended our approach to identify “symbolic” patterns, where absolute values are replaced with abstract symbols whenever appropriate to identify more subtle patterns across multiple system logs. Next, we have examined the applicability of our approach to troubleshoot harmful interactive complexity that may arise due to poor integration of adaptive components in server clusters. More specifically, we extended our approach to identify “cyclic” patterns in data center applications, which potentially highlights self-reinforcing loops. Finally, to complement our work on troubleshooting interactive complexity, we address the challenge of diagnosing occasional “lack of interaction” in deployed system. Such “lack of interaction” is often caused by unresponsive nodes. We develop the tele-diagnostic powertracer, an in-situ troubleshooting tool that uses external power measurements to determine the internal health condition of an unresponsive host and the most likely cause of its failure. Using our tool, we successfully distinguish between several categories of failures that cause unresponsive behavior including energy depletion, antenna damage, radio disconnection, system crashes, and anomalous reboots. To the best of our knowledge, we are the first to present a diagnostic tool that uses power measurements to diagnose sensor system failures remotely.
To my dear wife, Liza Zahid,

and

my beloved son, Yafee Ahyan Khan,

who sacrificed the most for this thesis.
Acknowledgements

By the grace of almighty Allah, it is a great privilege that I got the chance to express my gratitude to numerous people who helped and guided me through this tough journey of earning a PhD.

First and foremost, I would like to express my heartiest gratitude towards my PhD advisor, Professor Tarek F. Abdelzaher, who guided me with great patience and wisdom all these years. It is his continuous support, guidance, constructive suggestions, and criticisms that transformed me from an immature enthusiast to an able researcher. No amount of word can describe his contributions behind my success. He will be my role model, both in my personal and professional life, for the rest of my life. Most importantly, he made me a better person.

Next, I would like to thank Professor Jiawei Han, who helped me from the very beginning of my PhD research. It was my privilege to get the chance to work with one of the greatest scholars in data mining. His suggestions helped me immensely during the course of my dissertation research. Despite his phenomenal success as a researcher, he is one of the most humble and kind person I ever met, which truly moved me. He will be always a constant source of inspiration for me. I would also like to thank my other committee members, Professor Lui Sha and Dr. Jie Liu for their invaluable inputs. Their insightful comments helped improve this thesis significantly. I was truly honored and blessed to have them on my committee. I would like to thank all of the co-authors of my papers \[4, 5, 6, 7, 8, 9\]. I was extremely fortunate to get the chance to work with so many great people, who provided invaluable feedback on my work.

In the first year of my PhD, I worked as a teaching assistant (TA) under the supervision of Professor Marsha Woodbury for two semesters. She is one of the kindest persons I have ever seen. She
showed me what it takes to be a phenomenal teacher and educator. She was always very helpful whenever I had any problems. Her positive energy and outlook towards life had changed me forever. I will cherish every moment I spent with her for the rest of my life.

I would also like to thank Molly Flesner for her kindness and cooperation all these years. She always took care of all the complicated administrative tasks for me with a big smile on her face. I must also thank Barb Cicone, Mary Beth Kelley and Shirley Finke for their support and kindness.

I would also like to thank Professor Geneva Belford, who was always kind and compassionate whenever I needed help and suggestions.

I would also like to take this opportunity to thank all of my friends who supported me during these years. I had the good fortune of having a lot of good friends while I stayed at Urbana, IL. They always cheered me up when I was down, and gave me hope. The people of the Bangladeshi community made our experience a unique one, through their hospitality and support in tough times.

It is almost impossible to mention all the names who helped me and my family during this arduous journey. I would like to thank all who helped us, and made it possible for us.

In this turning point of my life, I especially remember my parents for their kindness and support. My father, Mohammad Zahirul Haque Khan, and my mother, Tahmina Zahir, showed great patience during all these years. Without their prayers, hard work, sacrifices and unconditional support, I would not be here today. I would also like to thank my brothers for their support and kindness. I am truly blessed to have such a wonderful family. I thank my father-in-law, Imam Abu Zahid, and my mother-in-law, Dil Afroza Akhter, for their support, patience and prayers as well.

Most importantly, I would not be writing this thesis today without the sacrifices of my wife, Liza Zahid, and my beloved son, Yafee Ahyan Khan. I was extremely fortunate to have such a wonderful family. They were with me all these years, and shared the great pain of financial difficulty and loneliness with me. In this long journey, I never felt alone. Without their continuous support and encouragement, I would not be able to follow my dream. It is at the expense of their sacrifices, I earned this degree today. I would like to express my heartiest gratitude towards them for their
invaluable sacrifices, and seemingly endless patience. Hence, I dedicate this thesis to my wife, Liza Zahid, and my beloved son, Yafee Ahyan Khan. I thank Allah for his endless mercy towards us.

Finally, I would like to thank the funding agencies for supporting my PhD work. Work on this thesis was funded in part by NSF grants CNS 09-05014, CNS 09-16028, and CNS 10-35736, and ARL under Cooperative Agreement Number W911NF-09-2-0053. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.
# Table of Contents

**List of Tables** ............................................................ xi

**List of Figures** .......................................................... xiii

## Chapter 1 Introduction .................................................. 1

1.1 Tool 1: Discriminative Sequence Mining and Interactive Complexity .................................... 4

1.2 Tool 2: Symbolic Sequence Mining and Interactive Complexity .......................................... 5

1.3 Tool 3: Identifying “Vicious” Cycles in Server Clusters ...................................................... 6

1.4 Troubleshooting the “Lack of Interaction” ............................................................................ 6

1.5 Research Contributions ........................................................................................................... 7

1.6 Organization ............................................................................................................................ 7

## Chapter 2 Discriminative Sequence Mining for Troubleshooting ............................................. 9

2.1 Dustminer Overview ................................................................................................................. 9

2.1.1 Frequent Pattern Generation: The Basic Algorithm .......................................................... 14

2.2 Adaptation of Sequence Mining for Debugging ........................................................................ 17

2.2.1 Challenge-I: Preventing False Frequent Patterns ............................................................... 18

2.2.2 Challenge-II: Suppressing Redundant Subsequences ......................................................... 19

2.2.3 Challenge-III: Capturing the Timing Effect ......................................................................... 21

2.2.4 Challenge-IV: Identifying the Accumulative Effect ............................................................ 22

2.3 Scalability Enhancements ........................................................................................................ 24

2.3.1 Two Stage Mining .............................................................................................................. 24

2.3.2 Progressive Discriminative Sequence Mining ................................................................. 26

2.4 Dustminer Architecture ........................................................................................................... 30

2.4.1 Data Collection Front-End ................................................................................................. 30

2.4.2 Data Preprocessing Middleware .......................................................................................... 31

2.4.3 Data Analysis Back-End ..................................................................................................... 32

2.5 Dustminer Implementation ....................................................................................................... 33

2.5.1 The Front-End: Acquiring System State ............................................................................ 33

2.5.2 The Data Analysis Back-End ............................................................................................. 37

2.6 Evaluation: Identifying the Timing Violations ........................................................................ 38

2.7 Evaluation: Identifying the Accumulative Effect ..................................................................... 39

2.8 Evaluation: Progressive Discriminative Sequence Mining .................................................. 40

viii
List of Tables

2.1 Timing Features .................................................. 22
2.2 Pre-processed Bad Log .......................................... 22
2.3 Progressive Discriminative Sequence Mining (Part 1) ........ 26
2.4 Progressive Discriminative Sequence Mining (Part 2) ....... 28
2.5 Log Statistics for the Timing Bug ......................... 39
2.6 Discriminative Timing Events .................................. 39
2.7 Log Statistics for the Accumulative Bug ................. 40
2.8 Discriminative Events for Accumulative Bug .............. 40
2.9 Logged events for diagnosing LiteOS application bug .... 41
2.10 Log Statistics for the LiteOS Bug ....................... 43
2.11 Discriminative frequent patterns found only in “good” log for LiteOS bug 44
2.12 Discriminative frequent patterns found only in “bad” log for LiteOS bug 44
2.13 Comparison of different schemes for the LiteOS bug .... 46
2.14 Logged events for diagnosing multichannel MAC protocol 48
2.15 Log Statistics for the MAC channel Bug ............... 49
2.16 Discriminative frequent patterns for multichannel MAC protocol .... 50
2.17 Comparison of different schemes for multichannel MAC Protocol bug 52
2.18 Log Statistics for the Directed Diffusion Bug .......... 53
2.19 Comparison of different schemes for Directed diffusion protocol bug 54

3.1 Sample Log File .................................................. 61
3.2 Symbolic Pattern Extraction Algorithm .................... 65
3.3 Top Patterns for Failure scenario - I .................... 69
3.4 Top Patterns for Directed Diffusion Protocol Bug ......... 70

4.1 Logged Events for Case Study I ............................. 79
4.2 Discriminative patterns due to conflicts between the DVS policy and the admission controller 79
4.3 Logged Events for Case Study II ............................. 82
4.4 Discriminative patterns due to conflict between the DVS policy and the machine On/Off policy 82
4.5 Discriminative patterns due to conflict between the load balancer and the thermal management policy 86
4.6 Transition Matrix for Patterns in Table 4.5 .................................................. 87

5.1 Confusion matrix for classification using HMM (window size=30 minutes, Number of states=50, Normalization used: Outlier filtering) ............................................ 105
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Calculation of Across Support and in-file Support</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>Index Structure</td>
<td>16</td>
</tr>
<tr>
<td>2.3</td>
<td>Flow of Basic Discriminative Analysis</td>
<td>16</td>
</tr>
<tr>
<td>2.4</td>
<td>Flow of Progressive Discriminative Analysis</td>
<td>26</td>
</tr>
<tr>
<td>2.5</td>
<td>Debugging framework</td>
<td>31</td>
</tr>
<tr>
<td>2.6</td>
<td>Impact of buffer size and event rate on logging performance</td>
<td>34</td>
</tr>
<tr>
<td>2.7</td>
<td>Flash space layout</td>
<td>35</td>
</tr>
<tr>
<td>4.1</td>
<td>Conflict Between the DVS Policy and the Admission Controller</td>
<td>80</td>
</tr>
<tr>
<td>4.2</td>
<td>Applying the DVS and the Machine On/Off Policy Together Consumes More Energy at High Load</td>
<td>83</td>
</tr>
<tr>
<td>4.3</td>
<td>Throughput of the System in Different Conditions</td>
<td>85</td>
</tr>
<tr>
<td>4.4</td>
<td>Machine Temperature During Normal Operation</td>
<td>87</td>
</tr>
<tr>
<td>4.5</td>
<td>Machine Temperature During the Anomaly</td>
<td>88</td>
</tr>
<tr>
<td>4.6</td>
<td>CPU Utilization</td>
<td>89</td>
</tr>
<tr>
<td>4.7</td>
<td>Number of Machines in Use</td>
<td>89</td>
</tr>
<tr>
<td>4.8</td>
<td>Average Number of Tasks Per Machine</td>
<td>90</td>
</tr>
<tr>
<td>5.1</td>
<td>A power-based tele-diagnostic system as an “add-on” to a sensing system</td>
<td>94</td>
</tr>
<tr>
<td>5.2</td>
<td>(a) Calibrate the power consumption of the computing subsystems indoor for the initial training of our tele-diagnostic system, (b) Outside look of a node in the solar-powered sensor network testbed</td>
<td>95</td>
</tr>
<tr>
<td>5.3</td>
<td>One node drowned in the flood by heavy rains</td>
<td>97</td>
</tr>
<tr>
<td>5.4</td>
<td>Possible failure states in our system. App-I is the application responsible for sensing sound using microphone. App-II is the application responsible for recording images.</td>
<td>98</td>
</tr>
<tr>
<td>5.5</td>
<td>Effect of the window size on classification accuracy for static feature based classification (Data preprocessing used: Outlier filtering)</td>
<td>100</td>
</tr>
<tr>
<td>5.6</td>
<td>Effect of number of HMM states on classification accuracy( Window size=30 minutes, Data preprocessing used: Outlier filtering)</td>
<td>103</td>
</tr>
<tr>
<td>5.7</td>
<td>Effect of sampling rate on the classification accuracy of HMM (Window size=30 minutes, Number of states=50, Data preprocessing used: Outlier filtering)</td>
<td>104</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>5.8</td>
<td>Effect of window size on the classification accuracy of HMM (Number of states=50, Data preprocessing used: Outlier filtering)</td>
<td>105</td>
</tr>
<tr>
<td>5.9</td>
<td>Effect of window size on the classification accuracy of HMM (Number of states=50, Data preprocessing used: Outlier filtering, z-score normalization)</td>
<td>106</td>
</tr>
<tr>
<td>5.10</td>
<td>Effect of window size on the classification accuracy of HMM (Number of states=50, Data preprocessing used: Outlier filtering, difference between consecutive signal)</td>
<td>107</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

With the proliferation of cheap embedded devices with built in sensors, actuators, and wireless connectivity, physical world is increasingly becoming an integral part of the computational world. Wild life habitat and environment monitoring [10, 11, 12], physical infrastructure monitoring [13], assisted living facility for caring and monitoring elderly patients [14, 15], community sensing applications [16, 17], interactive gaming (e.g., Wii), and 3-D tele-immersive computing [18, 19] are only a few of many such applications. More importantly, in near future, billions of devices with built in sensors and actuators (e.g., Internet enabled televisions to microwaves, interactive multi-party gaming consoles to smart meters) are expected to be connected to the Internet, and access various services leveraging the cloud platform. Similar vision is also shared by Internet pioneers, such as David Clark [20]. Given that most of the software and hardware are increasingly becoming adaptive to changing execution conditions, both physical and computational, we are heading towards building the largest cyber-physical system ever.

However, as our reliance grows on software, poor reliability of software may become more costly. In an early report on software certification and dependability [21], the authors expressed grave concern regarding the overall safety of the society as the dependence on software increases. They identified that, due to increase in interconnection and interdependence in software systems, local faults may cause wider failures. In a more recent book titled “Software for Dependable Systems: Sufficient Evidence?” [22], the authors stated as follows: “Both interactive complexity, where components may interact in unanticipated ways, and tight coupling, wherein a single fault cannot be isolated but brings about other faults that cascade through the system, are correlated with the
likelihood of system failure. Software-intensive systems tend to have both attributes”. In this book, the authors investigated diverse set of software systems, such as medical devices, avionics, infrastructure, defense, and distribution of energy and goods. They reported numerous software bugs that even cost human lives.

Indeed, as society’s dependence on software is growing dramatically, the poor reliability of the software is becoming more obvious and costly, as predicted in the book titled “Software for Dependable Systems: Sufficient Evidence?” [22]. Myriads of recalls of various models of cars by various automakers such as GM, Nissan, and Toyota due to software bugs, hours of downtime of Amazon cloud service, prevalence of life threatening software bugs in medical devices [23] are only few of many recent examples. With ubiquitous adoption and scale, the economic and social losses incurred due to software bugs are on the rise. Most importantly, with the ubiquitous adaptation and reliance on software (Internet enabled oven to software controlled fuel injection systems in our cars); software bugs may be very costly. Stalling of cars on the highway, and malfunction of hospital equipments may cost human lives. Given the implications of failures in such emerging cyber-physical computing systems, the importance of reliability of software systems can not be overemphasized.

This thesis is a step towards building more reliable software systems by automating the process of troubleshooting complex distributed cyber-physical systems. More specifically, our work focuses on the increased interactive complexity of software systems, which often spans across computational, physical, and social spaces, that causes reliability challenges, amplified by unexpected component interactions. Blending the computational world with the physical environment creates particularly hard-to-debug category of distributed component interaction errors that are environmentally induced, which are typically dependent on conditions in the physical environment. In such systems, bugs are often not localized to one component that is faulty, but rather result from complex and unexpected interactions among multiple, possibly individually non-faulty components. As a part of this thesis, I developed an extensible, generic framework that identifies the
“culprit” interaction patterns, which trigger the problems, by performing discriminative sequence analysis on logs of user-defined runtime events (e.g., message transmission events, disk access events, timer events etc.). Contrary to simple classification based approaches such as Bayesian network analysis and decision trees that identify combinations of unordered single features as discriminative and responsible for failure, our approach can identify ordered chains of events (not necessarily contiguous) and execution conditions that are causally correlated to failure.

Our tool attempts to answer high level questions such as “Why is throughput low?”, “Why is the communication delay so high?”, or “Why does the localization service failed?”, or “Why does the network reported two targets in response to one physical target?”. In principle, the philosophy behind the generic system troubleshooting framework we developed in this thesis [5, 6] is applicable to any system suffering from interactive complexity. The framework provides an extensible skeleton that may fit a variety of specific application needs. In this thesis, we took a more experimental approach towards troubleshooting interactive complexity leveraging discriminative sequence analysis techniques, and explored the subtleties in manifestations of interactive complexity in different application domains and extended our framework accordingly.

At a high level, our tool consists of three main components -(i) data collection front-end, (ii) data pre-processing middleware, and (iii) the data analysis back-end. The data collection front-end is used to collect user-defined run time events such as communication messages, timer firing events, disk access events and so on. The data pre-processing middleware provides the necessary functionalities for seamless integration between the data collection front-end and the data analysis back-end, which makes our tool widely applicable for a wide variety of applications. The data analysis back-end is independent of the data collection front-end, and performs discriminative frequent pattern analysis to identify the “culprit” sequences of events that are highly correlated to failure.

We realize that for different types of bugs, we may need different approaches and tools to diagnose the problem. We also do not want to use computationally expensive and complicated tools
to diagnose bugs that can be addressed by simpler tools and techniques. Keeping that in mind, we investigated multiple approaches, and built several tools that vary in diagnostic capabilities, resource requirements and runtime overheads which we discuss below.

1.1 Tool 1: Discriminative Sequence Mining and Interactive Complexity

In this part of our thesis, we focus on identifying ordered chains of events (not necessarily contiguous) and execution conditions that are causally correlated to failure. The key observation that leads to this part of our thesis is as follows. In a distributed computing environment, nodes interact with each other to perform tasks correctly. These interaction patterns can be represented as the concatenation of distributed sequences of events happening across multiple nodes. Most of these distributed protocols can be represented as a state transition graph where the system moves from one state to another in response to various events such as message received event, leader hand-off event, disk access event and so on. In a correctly functioning system, the state transition happens in a predefined pattern that the system is designed to handle. Occasionally due to design flaw and/or implementation flaw, system deviates from the expected sequence and potentially causes the system to fail/malfunction. The problem happens when something unexpected happens and the system deviates and enters a faulty state that was not handled properly (e.g., missing a step in a hand shaking protocol, or reordering of events leading to a corner case race condition).

In principle, if we log different types of events happening in the network, we can expect to be able to capture the “unexpected” sequence of events that leads to failure (along with thousands of other sequences of events). The key challenge for the diagnostic tool is to automatically identify this “culprit” sequence of events. We developed a customized discriminative sequence mining algorithm [6] to identify the “culprit” sequence of events by analyzing system logs. We identified that existing sequence mining algorithms [24, 25, 26] are ill suited when applied for debugging,
and addressed those limitations [6]. Using our tool, we identified corner case design flaw in a multi-channel MAC protocol for wireless sensor network, and a kernel level race condition bug in the LiteOS operating system.

1.2 Tool 2: Symbolic Sequence Mining and Interactive Complexity

In distributed environment, often multiple seemingly different event patterns lead to the same type of failure manifestation. In such cases, often a hidden relationship exists, in those patterns, among event attributes that is somehow responsible for the failure. Unfortunately, discriminative sequence mining is not very effective to pinpoint the cause of the problem due to its limited power to look at abstract relationships among event attributes. For example, in some systems, a message might always get corrupted if the sender is more than two hops away from the receiver (which is a distance relationship) irrespective of the senderId and receiverId. This observation leads to the next part of our thesis. To uncover such failure-causing relationships, we developed a new symbolic pattern extraction technique that identifies and symbolically expresses relationships correlated with anomalous behavior. Symbolic pattern mining generalizes discriminative patterns by substituting absolute values with abstract symbols across multiple runs and identifies the subtle interactions [7]. Symbolic pattern extraction is a new concept in debugging that is unique in its ability to generalize over patterns that involve different combinations of nodes or message exchanges by extracting their common relationship. We have identified corner case design bug in the directed diffusion protocol [3] using our approach.
1.3 Tool 3: Identifying “Vicious” Cycles in Server Clusters

In this part of our thesis, we investigated a different kind of interactive complexity which often arises due to incompatible composition of adaptive subsystems. In large scale systems such as server clusters, unintended interactions between components can cause performance problems even in the absence of bottlenecks or failures. In such systems, interactions often manifest as self-reinforcing feedback cycles that lead to degraded performance without any apparent reason. Leveraging discriminative sequence analysis technique developed in the earlier part of our thesis, we developed an algorithm to identify cyclic patterns representing such self-reinforcing loops. Instead of reporting all cyclic interactions inside the system (many of which are actually normal), the tool highlights only those patterns that are semantically conflicting. Such patterns may arise if actions taken by different performance mechanisms are conflicting with each other and causing instability. We successfully identified causes of energy consumption anomalies [9] in data centers using our tool.

1.4 Troubleshooting the “Lack of Interaction”

Finally, to complement our effort towards troubleshooting interactive complexity bugs, we looked into troubleshooting the cause behind occasional “lack of interactions” as well; for example, remotely-deployed sensor nodes that become unresponsive. Troubleshooting the cause of anomaly often requires field inspections, which can be costly and inconvenient. To address this challenge, we developed an in-situ tool [8] for troubleshooting unresponsive nodes. Based on different power signatures in different execution states (e.g., sensing, communication, disk access), the tool determines the internal health conditions of unresponsive hosts and the most likely cause behind the apparent lack of interactions (e.g., energy depletion, broken antenna, router failure, sensing failure) using Markov model.
1.5 Research Contributions

The contribution of this thesis towards automated troubleshooting interactive complexity bugs can be summarized as follows:

- This thesis presents a generic troubleshooting framework that enables easy integration of heterogeneous data collection front-end and data analysis back-end to facilitate troubleshooting diverse applications.

- We developed the first gapped subsequence mining algorithm for troubleshooting interactive complexity by analyzing system logs.

- We developed the first symbolic sequence mining algorithm that can effectively identify subtle patterns across multiple execution logs by generating symbolic patterns that lead to the problem.

- We developed a diagnostic powertracing tool that can successfully identify various hardware and software failures by analyzing power consumption traces.

- This thesis highlights the generality of our approach by applying an extended version of our tool to identify adverse interactions in data center applications.

1.6 Organization

The rest of the dissertation is organized as follows. Chapter 2 presents the first part of our thesis that describes the tool architecture, and our work on troubleshooting interactive complexity bugs using discriminative sequence mining. Chapter 3 highlights the strength of symbolic sequence mining for debugging interactive complexity bugs in distributed systems. We illustrate the generality of our tool in Chapter 4 by troubleshooting the adverse interactions among adaptive components in server clusters. Chapter 5 illustrates the approach behind our diagnostic powertracing tool that
can successfully troubleshoot the cause behind occasional “lack of interaction”. Related work in the area of troubleshooting distributed systems is discussed in Chapter 6. Finally, we present our conclusions along with future directions in Chapter 7.
Chapter 2

Discriminative Sequence Mining for Troubleshooting

In this chapter of our thesis, we introduce the key intuition behind discriminative sequence mining for troubleshooting interactive complexity bugs. In general, unintended interactions may be due to some protocol design flaw (e.g., missed corner cases that the protocol does not handle correctly) or unexpected artifacts of component integration. Interaction errors are often non-reproducible since repeating the experiment might not lead to the same corner-case again. The non-localized, hard-to-reproduce nature of such errors makes them especially hard to find. Hence, in contrast to previous debugging tools, in our thesis, we focus on finding (generally non-deterministically occurring) bugs that arise from interactions among seemingly individually sound components. We developed specialized algorithm to identify discriminative chain of events causally correlated to failure by mining system logs. In this chapter, we focus on troubleshooting wireless sensor network applications which often fail not because of a single node coding error but as a result of improper interaction between components. This leads to the development of Dustminer troubleshooting architecture, which we describe below.

2.1 Dustminer Overview

Dustminer is based on the idea that, in a distributed wireless sensor network, nodes interact with each other in a manner defined by their distributed protocols to perform cooperative tasks. Unexpected sequences of events, subtle interactions between modules, or unintended design flaws in

\[^{1}\]Part of the work presented in this chapter was done in collaboration and was published earlier [6, 5].
protocols may occasionally lead to an undesirable or invalid state, causing the system to fail or exhibit poor performance. Hence, in principle, if we log different types of events in the network, we may be able to capture the unexpected sequence that leads to failure (along with thousands of other sequences of events). The challenge for the diagnostic tool is to automatically identify this culprit sequence.

Our approach exploits both (i) non-determinism and (ii) interactive complexity to improve ability to diagnose distributed interaction bugs. This point is elaborated below:

- **Exploiting non-reproducible behavior:** We adapt data mining approaches that use examples of both “good” and “bad” system behavior to be able to classify the conditions correlated with good and bad. In particular, note that conditions that cause a problem to occur are correlated (by causality) with the resulting bad behavior. Root causes of non-reproducible bugs are thus inherently suited for discovery using such data mining approaches; the lack of reproducibility itself and the inherent system non-determinism improve the odds of occurrence of sufficiently diverse behavior examples to train the troubleshooting system to understand the relevant correlations and identify causes of problems.

- **Exploiting interactive complexity:** Interactive complexity describes a system where scale and complexity cause components to interact in unexpected ways. A failure that occurs due to such unexpected interactions is typically hard to “blame” on any single component. This fundamentally changes the objective of a troubleshooting tool from aiding in stepping through code (which is more suitable for finding a localized error in some line, such as an incorrect pointer reference), to aiding with diagnosing a sequence of events (component interactions) that leads to a failure state.

Before we describe the details of the algorithm, we introduce the concepts of event, sequence, frequent pattern, discriminative frequent pattern, and pattern ranking in the context of our dissertation below.
Event:

For the purpose of the discussion below, let us define an event to be the basic element in the log that is analyzed for failure diagnosis. The structure of an event in our log is as follows:

\(<\text{NodeId}, \text{EventType}, \text{attribute}_1, \text{attribute}_2, \ldots \text{attribute}_n, \text{Timestamp}\>\)

\text{NodeId} is used to identify the node that records the event. \text{EventType} is used to identify the event type (e.g., message dropped, flash write finished, etc). Based on the event type, it is possible to interpret the rest of the record (the list of attributes).

Since event parameter lists may be different for different event types, calling each variation a different event will cause a combinatorial explosion of the alphabet. For example, an event with 10 parameters, each of 10 possible values will generate a space of \(10^{10}\) possible combinations. To address the problem, continuous or fine-grained parameters need to be discretized into a smaller number of ranges. Multi-parameter events need to be converted into sequences of single-parameter events each listing one parameter at a time. Hence, the exponential explosion is reduced to linear growth in the alphabet, proportional to the number of discrete categories a single parameter can take and the average number of parameters per event. This point is further elaborated in Section 2.2.

Sequence:

The set of distinct \text{EventTypes} is often called the alphabet \text{alphabet} in an analogy with strings. In other words, if events were letters in an alphabet, we are looking for strings that cause errors to occur. These strings represent event sequences (ordered lists of events). The generated log can be thought of as a single sequence of logged events. For example, \(S_1 = (\langle a \rangle, \langle b \rangle, \langle c \rangle, \langle d \rangle, \langle b \rangle, \langle b \rangle, \langle a \rangle, \langle c \rangle)\) is an event sequence. Elements \(\langle a \rangle, \langle b \rangle, \ldots\) are events.

Support:

In our dissertation, we use two kinds of supports. First, for each unique event, we derive “across support”. Any event has two across supports, one is in respect of the good logs, and the other is in respect of the bad logs. If an event \(X\) occurs (irrespective of the number of times it occurred)
in 3 out of the 10 good logs, its across support is 0.3. This implies that there is a 30% probability that event X can be found in a good log. Second, we derive “in-file support” for each of the event. In real life, the amount of logged events and the corresponding frequency of patterns can be different from run to run depending on factors such as length of execution and system load. A higher sampling rate at sensors, for example, may generate more messages and cause more events to be logged. Many logged event patterns in this case will appear to be more frequent. This is problematic when it is desired to compare the frequency of patterns found in “good” and “bad” data piles for purposes of identifying those correlated with bad behavior. To address this issue, we need to normalize the frequency count of events in the log. In the case of single events (i.e., patterns of length 1), we use the ratio of occurrence of the event instead of absolute counts. In other words, the support of any particular event, \(< e >\) in the event log is divided by the total number of events logged, yielding in essence the probability of finding that event in the log, \(P(e)\). Alternately, the probability can be calculated based on time windows as well (i.e., probability of happening event \(< e >\) in any 5 sec window). Next, we normalize that probability across multiple files. The main idea behind across support and in-file support is illustrated in Figure 2.1.

Please note that across support is not used by the basic algorithm described in Section 2.1.1.
**Frequent Pattern:**
In our earlier algorithm [6], a pattern is considered frequent if it has support larger than some predefined minimumSupport threshold. Support for each pattern was calculated for each file separately, and if a particular pattern had support lower than the minimumSupport threshold, it was considered infrequent. The progressive discriminative sequence mining algorithm, described in Section 2.3.2, considers a pattern frequent only if it has high across support. This implies, a pattern will be considered a frequent good (or bad) pattern only if it exists in majority of the good (or bad) log files.

**Discriminative Frequent Pattern:**
A discriminative pattern between two data sets is a subsequence of (not necessarily contiguous) events that occurs with a “significantly” different support in the two sets. In case of progressive discriminative sequence mining algorithm described in Section 2.3.2, there can be two cases. First, a pattern is considered discriminative if it has high across support in the bad logs (or good logs) but not in the good logs (or bad logs). Second, a pattern can be discriminative if it has high across support in both good and bad logs, but significantly different ratio of in-file support. The larger the difference, the better the discrimination.

**Pattern Ranking:**
If the algorithm identifies multiple discriminative patterns, first, the algorithm ranks the patterns based on “across support”. If multiple patterns has same across support, the algorithm ranks them based on “in-file” support ratio between good and bad. The larger the ratio, the higher the rank.

With the above terminology in mind, we present the main idea behind our tool below.

**Main Idea:**
At a high level, our tool first uses a data collection front-end to collect runtime events for post-mortem analysis. Once the log of runtime events is available, the tool separates the collected sequence of events into two piles - a “good” pile, which contains the parts of the log when the system performs as expected, and a “bad” pile, which contains the parts of the log when the system
fails or exhibits poor performance. This data separation phase can be done manually, or based on a predicate that defines “good” versus “bad” behavior, provided by the application developer. For example, the predicate applied offline to logged data, might state that a sequence of more than 10 consecutive lost messages between a sender and receiver is bad behavior (hence return “bad” in this case). To increase diagnostic accuracy, experiments can be run multiple times before data analysis. In all of our case studies, the data separation was done manually except the one described in Section 2.8.3. In the directed diffusion protocol bug, the logs were labeled “bad” if more than 10 consecutive messages were lost during the experiment.

A discriminative frequent pattern mining algorithm then looks for patterns (sequences of events) that exist with very different frequencies in the two piles. These patterns are called *discriminative*. We present the main idea behind frequent pattern generation below.

### 2.1.1 Frequent Pattern Generation: The Basic Algorithm

A well-known algorithm for finding frequent itemset in data mining is the Apriori algorithm [24]. Some of the more efficient algorithms for frequent itemset mining includes FPgrowth [25] and PrefixSpan [27]. But these algorithms do not handle gapped subsequence mining. For efficiency, we developed and implemented our own algorithm [5], which we refer as “the basic algorithm” in this thesis. The basic algorithm works as follows.

The basic algorithm, used in our earlier work [5], is an iterative algorithm that proceeds as follows. At the first iteration, it counts the number of occurrences (called *support*) of each distinct event in the data set (i.e., in the “good” or “bad” pile). Next, the algorithm discards all events that are infrequent (their support is less than some parameter *minSup*). The remaining events are frequent patterns of length 1. Assume the set of frequent patterns of length 1 is \( S_1 \). At the next iteration, the algorithm generates all the candidate patterns of length 2 which is \( S_1 \times S_1 \). Here ‘\( \times \)’ represents the Cartesian product. For example, if \( S_1 \) includes events \( A, B, \) and \( C \), \( S_1 \times S_1 \) includes \( AB, AC, BC, BA, CA, CB \). It then computes the frequency of occurrence of each pattern
in $S_1 \times S_1$, and discards those with support less than $\text{minSup}$ again. The remaining patterns are the frequent patterns of length 2. Let us call them set $S_2$. Similarly, the algorithm will generate all the candidate patterns of length 3 which is $S_2 \times S_1$, and discards infrequent patterns (with support less than $\text{minSup}$) to generate $S_3$ and so on. It continues this process until it cannot generate any more frequent patterns, or it reaches the maximum length defined by the user. Next, it performs discriminative analysis to identify patterns that occur with significantly different support in good and bad piles. As this algorithm does not include the extensions described in Section 2.2, we refer to this implementation as the “Basic Algorithm” in the rest of the dissertation. The flow of the basic algorithm is illustrated in Figure 2.3

For efficient computation of support, we build an index data structure for each input file. For each unique event, an array is constructed that lists the index of occurrences of that event in a particular file. It requires a single scan of the data file. During the sequence mining, we leverage this index structure to check whether a particular pattern exists or not, and also to count support. This avoids repeated scanning of input files. The main idea behind this data structure is shown in Figure 2.2. The process of counting frequency is explained below using an example. Let us assume that we want to count the frequency of the pattern $<ABC>$. At the first scan, we build the index structures for $A$, $B$, and $C$ as shown in Figure 2.2. Next, we count the frequency of the pattern $<AB>$. This returns the index structure for pattern $<AB>$ as $<(1,2),(4,5),(8,9)>$. Now, when searching for pattern $<ABC>$, we can leverage the intermediate index list for pattern $<AB>$ from the previous step of the algorithm. This helps to avoid repeated scanning of the input files, and speeds up the mining process significantly.

We show in this dissertation that the basic algorithm has serious limitations and extend this algorithm to suite the purpose of sensor network debugging as described in section 2.2 and Section 2.3. Please note that the limitations described in Section 2.2 is true for any other sequence mining algorithm.
Figure 2.2: Index Structure

Figure 2.3: Flow of Basic Discriminative Analysis
2.2 Adaptation of Sequence Mining for Debugging

Performing discriminative frequent pattern mining based on frequent patterns generated by the basic algorithm poses several challenges that need to be addressed before we can apply discriminative frequent pattern mining for debugging. As defined in Section 2.1, an event is the basic element in the log that is analyzed for failure diagnosis. The structure of an event in our log is as follows:

\(<\text{NodeId}, \text{EventType}, \text{attribute}_1, \text{attribute}_2, \ldots \text{attribute}_n, \text{Timestamp}>\)

As different event types can have a different number of attributes in the tuple, mining frequent patterns becomes much more challenging as the mining algorithm has no prior knowledge of which attributes in a specific event are correlated with failure and which are not. For example, consider the following sequence of events:

\(<\text{msg\_sent}, \text{nodeid} = 1, \text{msgtype} = 2, \text{nodetype} = l>\)
\(<\text{msg\_sent}, \text{nodeid} = 2, \text{msgtype} = 2, \text{nodetype} = m>\)

In the above pattern, we do not know which of the attributes are correlated with failure (if any are related at all). It could be nodeid, or msgtype, or a combination of msgtype and nodetype and so on. One trivial solution is to try all possible permutations. However, this is exponential in the number of attributes and becomes unmanageable very quickly. Rather, we split such multi attribute events into a sequence of single attribute events, each with only one attribute of the original multi-attribute event. The converted sequence for the above example is as follows:

\(<\text{msg\_sent}, \text{nodeid} = 1>\)
\(<\text{msg\_sent}, \text{msgtype} = 2>\)
\(<\text{msg\_sent}, \text{nodetype} = l>\)
\(<\text{msg\_sent}, \text{nodeid} = 2>\)
\(<\text{msg\_sent}, \text{msgtype} = 2>\)
\(<\text{msg\_sent}, \text{nodetype} = m>\)

We can now apply simple (uni-dimensional) sequence mining techniques to the above sequence.
As before, the user will be given the resulting sequences (that are most correlated with failures). In such sequences, only the relevant attributes of the original multidimensional events will likely survive. Attributes irrelevant to the occurrence of failure will likely have a larger spread of values (since these values are orthogonal to the failure) and hence a lower support. Sequences containing them will consequently have a lower support as well. The top ranking sequences are therefore more likely to focus only on attributes of interest, which is what we want to achieve.

Moreover, when logged parameters are continuous, it is very hard to identify frequent patterns as there are potentially an infinite number of possible values for them. To map continuous data to a finite set, we simply discretize into a number of categories (bins).

In our case, a discriminative pattern between two data sets is a subsequence of (not necessarily contiguous) events that occurs with a different count in the two sets. The larger the difference, the better the discrimination. With the above terminology in mind, we present how the algorithm is tailored to apply to debugging.

### 2.2.1 Challenge-I: Preventing False Frequent Patterns

The basic algorithm generates all possible combinations of frequent subsequences of the original sequence. As a result, it generates subsequences combining events that are “too far” apart to be causally correlated with high probability and thus reduces the chance of finding the “culprit sequence” that actually caused the failure. This strategy could negatively impact the ability to identify discriminative patterns in two ways; (i) it could lead to the generation of discriminative patterns that are not causally related, and (ii) it could eliminate discriminative patterns by generating false patterns. Consider the following example.

Suppose we have the following two sequences:

\[
S_1 = (<a>, <b>, <c>, <d>, <a>, <b>, <c>, <d>)
\]

\[
S_2 = (<a>, <b>, <c>, <d>, <a>, <c>, <b>, <d>)
\]

Suppose the system fails when \(<a>\) is followed by \(<c>\) before \(<b>\). As this condition is violated
in sequence $s_2$, ideally, we would like our algorithm to be able to detect $(<a>,<c>,<b>)$ as a discriminative pattern that distinguishes these two sequences.

Now, if we apply the basic algorithm, it will generate $(<a>,<c>,<b>)$ as an equally likely pattern for both $s_1$, and $s_2$. As in both $s_1$ and $s_2$, it will combine the first occurrence of $<a>$ and the first occurrence of $<c>$ with the second occurrence of $<b>$. So it will get canceled out at the differential analysis phase.

To address this issue, the key observation here is that the first occurrence of $<a>$ should not be allowed to combine with the second occurrence of $<b>$ as there is another event $<a>$ after the first occurrence of $<a>$ but before the second occurrence of $<b>$ and the second occurrence of $<b>$ is correlated with second occurrence of $<a>$ with higher probability.

To prevent such erroneous combinations, we use a dynamic search window scheme where the first item of any candidate sequence is used to determine the search window. In this case, for any pattern starting with $<a>$, the search window is $[1, 4]$ and $[5, 8]$ in $s_1$ and $s_2$. With this search window, the algorithm will search for pattern $(<a>,<c>,<b>)$ in window $[1, 4]$ and $[5, 8]$ and will fail to find it in $s_1$ but will find it in sequence $s_2$ only. As a result, the algorithm will be able to report pattern $(<a>,<c>,<b>)$ as a discriminative pattern.

This dynamic search window scheme also speeds up the search significantly. In this scheme, the original sequence (of size 8 events) was reduced to windows of size 4 making the search for patterns in those windows more efficient.

### 2.2.2 Challenge-II: Suppressing Redundant Subsequences

At the frequent pattern generation stage, if two patterns, $s_i$ and $s_j$, have support $\geq \minSup$, keeping both sequences as frequent patterns even if one is a subsequence of the other and both have equal support can be problematic. For example, when mining the “good” data set, the above strategy assumes that any subset of a “good” pattern is also a good pattern. In real-life, this is not true. Forgetting a step in a multi-step procedure may well cause failure. Hence, subsequences of good
sequences are not necessarily good. Keeping these subsequences as examples of “good” behavior leads to a major problem at the differential analysis stage when discriminative patterns are generated since they may incorrectly cancel out similar subsequences found frequent in the other (i.e., “bad” behavior) data pile. For example, consider two sequences below:

\[ S_1 = (a, b, c, d, a, b, c, d) \]
\[ S_2 = (a, b, c, d, a, b, d, c) \]

Suppose, for correct operation of the protocol, event \( a \) has to be followed by event \( c \) before event \( d \) can happen. In sequence \( S_2 \) this condition is violated. Ideally, we would like our algorithm to report the following sequence \( S_3 \) as the “culprit” sequence:

\[ S_3 = (a, b, d) \]

However, if we apply the basic algorithm, it will fail to catch this sequence. This is because it will generate \( S_3 \) as a frequent pattern both for \( S_1 \) and \( S_2 \) with support 2 and will get canceled out at the differential analysis phase. As expected, \( S_3 \) will never show up as a “discriminative pattern”. Note that with the dynamic search window scheme alone, we cannot prevent this.

To illustrate, suppose a successful message transmission involves the following sequence of events:

\[ (<enableRadio>, <messageSent>, <ackReceived>, <disableRadio>) \]

Now although sequence:

\[ (<enableRadio>, <messageSent>, <disableRadio>) \]

is a subsequence of the original “good” sequence, it does not represent a successful scenario as it disables radio before receiving the “ACK” message.

To solve this problem, we need an extra step (which we call sequenceCompression) before we perform differential analysis to identify discriminative patterns. This step is similar to closed item set mining [28, 29] commonly used in data mining to eliminate frequent subsets of a super set. At this step, we remove the sequence \( s_i \) if it is a subsequence of \( s_j \) with the same support. This will remove all the redundant subsequences from the frequent pattern list. Subsequences with

\[ ^1 \text{This mechanism can be extended to remove subsequences of a similar but not identical support.} \]
a (sufficiently) different support, will be retained and will show up after discriminative pattern mining.

In the above example, pattern \(<a>,<b>,<c>,<d>\) has support 2 in \(s_1\) and support 1 in \(s_2\). Pattern \(<a>,<b>,<d>\) has support 2 in both \(s_1\) and \(s_2\). Fortunately, at the \textit{sequenceCompression} step, pattern \(<a>,<b>,<d>\) will be removed from the frequent pattern list generated for \(s_1\) because it is a subsequence of a larger frequent pattern of the same support. It will therefore remain only on the frequent pattern list generated for \(s_2\) and will show up as a discriminative pattern.

### 2.2.3 Challenge-III: Capturing the Timing Effect

Although all the events in the log are temporally ordered, the algorithm used in Dustminer \cite{6} does not consider the time differences between events as a feature while mining for discriminative sequences of events. This may cause the algorithm to fail identify certain patterns that involve the timing relations. For example, due to hardware limitations, sampling the sensors “too” frequently may cause the sensor readings to become faulty due to capacitance effect. Moreover, various actions may be triggered due to timing of the events (e.g., periodic neighbor discovery protocol). Unfortunately, taking into account all the timing differences can be very challenging due to an exponential number of possibilities. To address this challenge, we introduce a data pre-processing step. In this step, first, we calculate the average time differences and standard deviations among all possible event types. For example, if A, B, and C are three different event types, we calculate average time differences and standard deviations for AB, AC, BA, BC, CA, and CB. Please note that in this step we only consider the set of good logs. In the second step, we consider both good and bad logs. In the second step, we check each individual occurrence of pair of events and calculate the time difference. If the time difference is larger than “\(x\)” times \((x = 1.5\) in our implementation\) of the standard deviation, we insert a new event at that place. As we insert fake events only if the difference is larger or smaller than certain threshold, this pre-processing step increases the size of the log only if there is a potential for timing violations. Table \ref{table:2.1} highlights the process.
of calculating the timing statistics. Table 2.2 illustrates the process of inserting fake events that captures the potential timing violations. Once the pre-processing is done, we run our algorithm as before. If the anomalous behavior is due to timing violation, our algorithm will identify the events that were inserted during the data preprocessing step.

### 2.2.4 Challenge-IV: Identifying the Accumulative Effect

Pair wise causality relationship (e.g., one message sent is followed by one acknowledgement) is fairly common in the program execution. But there are cases when this is not true. For example,
in many file systems, it is often common practice to buffer data before actually writing to disk to minimize disk access and to save energy [30]. In a correctly functioning system, it can be expected that multiple buffer operations will be followed by a single disk write operation. Assume that the data buffering and buffer flush operations are represented by symbols \( B \) and \( F \) respectively. A normal operation would generate sequence of events that may looks like as follows:

\[
S_{\text{good}} = (B, B, F, B, B, B, B, F, B)
\]

Now, a failed operation might generate sequence of events as follows:

\[
S_{\text{bad}} = (B, B, F, B, B, B, B, F, B)
\]

Note that in \( S_{\text{bad}} \), the last buffer operation(\( B \)) is not followed by a flush operation(\( F \)).

Unfortunately, sequence mining algorithm would identify the pattern \((B, F)\) as a common pattern in both good and bad logs. It would also identify \( B \) as a common event with support 6 in both cases. As \( B \) has a different support than \((B, F)\) in both good and bad logs, our subsequence elimination rule would fail to eliminate \( B \) in either of the logs and eventually would cancel out in the discriminative analysis.

Please note that, in this example, for simplicity, we assumed there are only two types of events, \( B \) and \( F \). In reality there will be other events in between. However, as we search for gapped subsequence, having other events in between will have no effect on our algorithm in terms of accuracy. For example, when the algorithm searches for pattern \((B, F)\), it ignores other events in the search window.

We observe that a very minor modification to our previously proposed dynamic search window scheme in section 2.2.1 can solve this shortcoming. According to our earlier scheme, any subsequence that starts with \( F \) can combine with any other events that happened before the next occurrence of \( F \). We changed the definition to include the next occurrence of \( F \) in the search window. With the new search window, the algorithm can easily identify that \((B, F, B, F)\) \( F \) is a common pattern in both good and bad logs whereas \((F, B)\) occurred only in bad log. Note that \((F, B)\) will be generated in good log as well but will be
eliminated as it is a subsequence of \((< F >, < B >, < F >)\) with same support (i.e., support 2). But in the bad log, \((< F >, < B >, < F >)\) has support 2, and \((< F >, < B >)\) has support 3. Hence \((< F >, < B >)\) will be retained and will be reported as a discriminative pattern, which clearly indicates that in one case \(< B >\) is not followed by \(< F >\). Without this extension, this pattern can not be identified.

2.3 Scalability Enhancements

2.3.1 Two Stage Mining

In debugging, sometimes less frequent patterns could be more indicative of the cause of failure than the most frequent patterns. A single mistake can cause a damaging sequence of events. For example, a single node reboot event can cause a large number of message losses. In such cases, if frequent patterns are generated that are commonly found in failure cases, the most frequent patterns may not include the real cause of the problem. For example, in case of node reboot, manifestation of the bug (message loss event) will be reported as the most frequent pattern and the real cause of the problem (the node reboot event) may be overlooked.

Fortunately, in the case of sensor network debugging, a solution may be inspired by the nature of the problem domain. The fundamental issue to observe is that much computation in sensor networks is recurrent. Code repeatedly visits the same states (perhaps not strictly periodically), repeating the same actions over time. Hence, a single problem, such as a node reboot or a race condition that pollutes a data structure, often results in multiple manifestations of the same unusual symptom (like multiple subsequent message losses or multiple subsequent false alarms). Catching these recurrent symptoms by an algorithm such as the basic algorithm described in Section 2.1.1 is much easier due to their larger frequency. With such symptoms identified, the search space can be narrowed and it becomes easier to correlate them with other less frequent preceding event
occurrences. To address this challenge, we developed a two stage pattern mining scheme. At the first stage, the basic algorithm generates the usual frequent discriminative patterns that have support larger than $minSup$. For the first stage, $minSup$ is set larger than 1. It is expected that the patterns involving manifestations of bugs will survive at the end of this stage but infrequent events like a node reboot will be dropped due to their low support.

At the second stage, at first, the algorithm splits the log into fixed width segments (default width is 50 events in our implementation). Next, the algorithm counts the number of discriminative frequent patterns found in each segment and ranks each segment of the log based on the count (the higher the number of discriminative patterns in a segment, the higher the rank). If discriminative patterns occurred consecutively in multiple segments, those segments are merged into a larger segment. Next, the algorithm generates frequent patterns with $minSup$ reduced to 1 on the $K$ highest-ranked segments separately (default $K$ is 5 in our implementation) and extracts the patterns that are common in these regions. Note that the initial value of $K$ is set conservatively. The optimum value of $K$ depends on the application. If with the initial value of $K$, the tool failed to catch the real cause, the value of $K$ is increased iteratively. In this scheme, we have a higher chance of reporting single events such as race conditions that cause multiple problematic symptoms. Observe that the algorithm is applied on data that is the total logs from several experimental runs. The race condition may have occurred once at different points of some of these runs.

This scheme has a significant impact on the performance of the frequent pattern mining algorithm. Scalability is one of the biggest challenges in applying discriminative frequent pattern analysis to debugging. For example, if the total number of logged events is of the order of thousands (more than 40000 in one of our later examples), it is computationally impossible to generate frequent patterns of non-trivial length for this whole sequence. Using two stage mining, we can dramatically reduce the search space and make it feasible to mine for longer frequent patterns which are more indicative of the cause of failure than shorter sequences.
2.3.2 Progressive Discriminative Sequence Mining

![Flow of Progressive Discriminative Analysis](image)

**Algorithm: Progressive Discriminative Sequence Mining**

**Input:** Set of Good Logs (GL), Set of Bad Logs (BL)
**User defined thresholds:** $\theta$, $\delta$, $\sigma$, $\pi$
**Output:** Set of discriminative sequences of events

1. $S_{\text{common}} = \emptyset$, $SG = \emptyset$, $SB = \emptyset$, $k = 1$
2. while (SG == $\emptyset$ or SB == $\emptyset$)
   2.1 $S_{\text{good}} = \text{GenerateFrequentSubSequences}(GL, k, S_{\text{common}})$
   2.2 $S_{\text{bad}} = \text{GenerateFrequentSubSequences}(BL, k, S_{\text{common}})$
   2.3 if (SG == $\emptyset$) $SG = SG \cup \text{FindDiscriminative}(S_{\text{good}}, S_{\text{bad}})$
   2.4 if (SB == $\emptyset$) $SB = SB \cup \text{FindDiscriminative}(S_{\text{bad}}, S_{\text{good}})$
   2.5 $S_{\text{common}} = \text{FindCommon}(S_{\text{good}}, S_{\text{bad}})$
   2.6 $k = k + 1$

**Table 2.3: Progressive Discriminative Sequence Mining (Part 1)**

Although two stage mining addresses the scalability issue to some extent, the algorithm still suffers from the following problems. *First,* due to exponential number of combinations, the number of
candidate patterns grows very quickly. This exponential growth rate of the size of the candidate patterns is dependent on the size of the base patterns that is used to generate the candidate set rather than the size of the log. Second, the number of patterns returned is typically in the order of several thousands. Although the “culprit” patterns are expected to be at the top of the list, this number of final patterns is still daunting.

Our prior algorithm [6] generates all the frequent patterns of length up to “n” that are common across all the bad logs and common across all the good logs before performing the discriminative analysis to identify the “culprit” sequences of events. In this approach, the algorithm generates a lot of patterns that are eventually going to be dropped at the last stage.

Inspired by the work presented in DDPMine [26], we developed the progressive discriminative analysis which tries to prune patterns as early as possible without risking the possibility of dropping the “culprit” sequence. Instead of performing discriminative analysis at the last stage, we perform discriminative analysis at each stage \( k \) after generating frequent patterns of length \( k \). Our approach has several differences with the work presented in DDPMine [26]. First, DDPMine is for frequent unordered item set mining. In contrast, our algorithm is for ordered sequence mining. Second, our pruning strategy for early elimination of candidate patterns is different than DDPMine. To prune patterns, DDPMine exploited the idea that the information gain upper bound is lower for less frequent items/events. However, using this approach is not suitable for debugging. We took a different approach as explained below.

Suppose, we have \( I \) number of good files and \( J \) number of bad files. Now, assume that \( S_{\text{good}} \) is the set of good patterns of length \( k \). Each pattern \( p_i \) in \( S_{\text{good}} \) has two supports, \( sup_{\text{inFile}} \) and \( sup_{\text{acrossFile}} \). For \( p_i \) to be frequent, \( sup_{\text{acrossFile}} \) has to be larger than \( \text{minSupport} \) threshold. If \( \text{minSupport} = 0.8 \), this condition implies that each pattern in set \( S_{\text{good}} \) occurred in at least 80% of the good files. Similarly, assume that \( S_{\text{bad}} \) is the set of bad patterns of length \( k \) and each pattern \( q_j \) in \( S_{\text{bad}} \) has two supports, \( sup_{\text{inFile}} \) and \( sup_{\text{acrossFile}} \). For \( q_j \) to be frequent, \( sup_{\text{acrossFile}} \) has to be larger than \( \text{minSupport} \) threshold.
**Function: FindDiscriminative**
**Input:** Set of Frequent SubSequences(A), Set of Frequent SubSequences(B)
**Return:** Set of discriminative SubSequences that distinguishes A from B
**Assumption:** Each sequence $p_i$ in A or B has two supports, $sup_{inFile}$ and $sup_{acrossFile}$.

- $sup_{inFile}$ represents the average number of occurrence of $p_i$ within a file.
- $sup_{acrossFile}$ represents the probability of occurrence of $p_i$ in a file.

1. $S_{discriminative}=\emptyset$
2. for each sequence $p_i$ in A
   2.1 boolean found = FALSE
   2.2 for each sequence $q_j$ in B
      2.2.1 if ($p_i == q_j$) then
         2.2.1.1 found = TRUE
         2.2.1.2 if ($sup_{acrossFile}/sup_{acrossFile} <= \theta$) then
            $S_{discriminative}=S_{discriminative} \cup p_i$
         2.2.1.3 else if ($sup_{inFile}/sup_{inFile} <= \delta$) then
            $S_{discriminative}=S_{discriminative} \cup p_i$
      2.3 if (found == FALSE) then
         $S_{discriminative}=S_{discriminative} \cup p_i$
3. Return $S_{discriminative}$

**Function: FindCommon**
**Input:** Set of Frequent SubSequences(A), Set of Frequent SubSequences(B)
**Return:** Set of SubSequences that are common in set A and set B
**Assumption:** Each sequence $p_i$ in A or B has two supports, $sup_{inFile}$ and $sup_{acrossFile}$.

- $sup_{inFile}$ represents the average number of occurrence of $p_i$ within a file.
- $sup_{acrossFile}$ represents the probability of occurrence of $p_i$ in a file.

1. $S_{common}=\emptyset$
2. for each sequence $p_i$ in A
   2.1 for each sequence $q_j$ in B
      2.1.1 if ($p_i == q_j$) then
         2.1.1.1 if (($sup_{acrossFile}/sup_{acrossFile} >= \pi$) and ($sup_{inFile}/sup_{inFile} >= \sigma$)) then
            $S_{common}=S_{common} \cup p_i$
   3. Return $S_{common}$

**Function: GenerateFrequentSubSequences**
**Input:** Set of Logs(L), sequenceLength(k), baseSet($S_{common}$)
**Return:** Set of frequent SubSequences of length k

1. Use the basic algorithm described in Section 2.1.1 to generate frequent subsequences of length k using baseSet
2. Return frequent subsequences of length k generated at step 1

Table 2.4: Progressive Discriminative Sequence Mining (Part 2)
Now, before generating patterns of length \((k+1)\) we do the followings. We calculate three sets of patterns. \(\text{FinalGP}_k = \text{FindDiscriminative}(S_{\text{good}}, S_{\text{bad}})\), \(\text{FinalBP}_k = \text{FindDiscriminative}(S_{\text{bad}}, S_{\text{good}})\), \(\text{CommonP}_k = \text{FindCommon}(S_{\text{good}}, S_{\text{bad}})\).

The function \(\text{FindDiscriminative}(S_{\text{good}}, S_{\text{bad}})\) returns all the patterns that are in the set of good logs and discriminative. In contrary, the function \(\text{FindDiscriminative}(S_{\text{bad}}, S_{\text{good}})\) returns all the patterns that are in the set of bad logs and discriminative.

The function \(\text{FindCommon}(S_{\text{good}}, S_{\text{bad}})\) returns all the patterns that are found in the set of both good and bad logs, and are frequent. If we fail to find any discriminative pattern of length \(k\), this set is used to grow patterns of length \((k+1)\). The function FindDiscriminative and function FindCommon are defined in Table \(2.4\).

At the next step, we use the patterns in set \(\text{CommonP}_k\) to generate patterns of length \((k+1)\). We stop using patterns in set \(\text{FinalGP}_k\) and \(\text{FinalBP}_k\) to generate longer patterns. Because any pattern in \(\text{FinalGP}_k\) (or \(\text{FinalBP}_k\)) is already discriminative, and by making them longer is not going to make them any more discriminative. Rather it can decrease their potential for being discriminative. So, our algorithm stops as soon as it finds discriminative patterns, which implies we stop searching at the minimal length.

This scheme has the following three advantages. \textit{First}, as we prune at each stage, it reduces the size of the candidate patterns that needs to be checked substantially and speeds up the overall process by a huge factor. \textit{Second}, it reduces the size of the final patterns returned as the discriminative patterns. In one example, it returned just five patterns instead of thousands. \textit{Third}, in this scheme, the user can now decide to stop the analysis as soon as the set of discriminative good and bad patterns become nonempty. Earlier we have to specify the parameter “\(n\)” which is the maximum length of the pattern that the user wishes to generate. The optimum value of “\(n\)” can be hard to guess apriori. If it is “too short” or “too long”, the algorithm may fail to identify the “culprit” pattern. The algorithm is presented in Table \(2.3\) and Table \(2.4\). The flow of the algorithm is illustrated in Figure \(2.4\). We evaluated progressive discriminative sequence mining in section \(2.8\).
2.4 Dustminer Architecture

We realize that the types of debugging algorithms needed are different for different applications, and are going to evolve over time with the evolution of hardware and software platforms. Hence, we aim to develop a modular tool architecture that facilitates evolution and reuse. Keeping that in mind, we developed a software architecture that provides the necessary functionality and flexibility for future development. The goal of our architecture is to facilitate easy use and experimentation with different debugging techniques and foster future development. As there are numerous different types of hardware, programming abstractions, and operating systems in use for wireless sensor networks, the architecture must be able to accommodate different combinations of hardware and software. Different ways of data collection should not affect the way the data analysis layer works. Similarly we realize that for different types of bugs, we may need different types of techniques to identify the bug and we want to provide a flexible framework to experiment with different data analysis algorithms. Based on the above requirements, we designed a layered, modular architecture as shown in Figure 2.5. We separate the whole system into three subsystems; (i) a data collection front-end, (ii) data preprocessing middleware and (iii) a data analysis back-end.

2.4.1 Data Collection Front-End

The role of data collection front-end is to provide the debug information (i.e., log files) that can be analyzed for diagnosing failures. The source of this debug log is irrelevant to the data analysis subsystem. As shown in Figure 2.5, the developer may choose to analyze the recorded radio communication messages obtained using a passive listening tool, or the execution traces obtained from simulation runs, or the run-time sequences of events obtained by logging on actual application motes and so on. With this separation of concerns, the front-end developer can design and implement the data collection subsystem more efficiently and independently. The data collection
front-end developer merely needs to provide the format of the recorded data. These data are used by the data preprocessing middleware to parse the raw recorded byte streams.

2.4.2 Data Preprocessing Middleware

This middleware that sits between the data collection front-end and the data analysis back-end provides the necessary functionality to change or modify one subsystem without affecting the other. The interface between the data collection front-end and the data analysis back-end is further divided into the following layers:

- **Data cleaning layer**: This layer is front-end specific. Each supported front-end will have one instance of it. The layer is the interface between the particular data collection front-end
and the data preprocessing middleware. It ensures that the recorded events are compliant with format requirements.

- **Data parsing layer:** This layer is provided by our framework and is responsible for extracting meaningful records from the recorded raw byte stream. To parse the recorded byte stream, this layer requires a header file describing the recorded message format. This information is provided by the application developer (i.e., the user of the data collection front-end).

- **Data labeling layer:** To be able to identify the probable causes of failure, the data analysis subsystem needs samples of logged events representing both “good” and “bad” behavior. As “good” or “bad” behavior semantics are an application specific criterion, the application developer needs to implement a predicate (a small module) whose interface is already provided by us in the framework. The predicate, presented with an ordered event log, decides whether behavior is good or bad.

- **Data conversion layer:** This layer provides the interface between the data preprocessing middleware and the data analysis subsystem. One instance of this layer exists for each different analysis back-end. This layer is responsible for converting the labeled data into appropriate format for the data analysis algorithm. The interface of this data conversion layer is provided by the framework. As different data analysis algorithms and techniques can be used for analysis, each may have different input format requirements. This layer provides the necessary functionality to accommodate supported data analysis techniques.

### 2.4.3 Data Analysis Back-End

Data analysis back-end is responsible for identifying the causes of failures. The approach is extensible. As newer analysis algorithms are developed that catch more or different types of bugs, they can be easily incorporated into the tool as alternative back-ends. Such algorithms can be applied in parallel to analyze the same set of logs to find different problems with them.
2.5 Dustminer Implementation

In this section, we describe the implementation of the data collection front-end and the data analysis back-end that are used for failure diagnosis. We used three different data collection front-ends for three different case studies. The front-end used for the first case study was a built-in logging support functionality provided by the LiteOS operating system for MicaZ motes. For the second case study, the front-end is implemented by us and used for real time logging of user defined events on flash memory in MicaZ motes. For the last case study, we used the logging support provided by TOSSIM [31] for logging different runtime events in simulation. At the data analysis back-end, we used discriminative frequent pattern analysis for failure diagnosis. We describe the implementation of each of these next.

2.5.1 The Front-End: Acquiring System State

In our thesis, we used three different data collection front-ends to collect data: (i) event logging system implemented by us for MicaZ platform in TinyOS 2.0, (ii) the kernel event logger for MicaZ platform provided by LiteOS, and (iii) the logging support provided by TOSSIM [31] for logging different runtime events in simulation. The format of the event logged by the three subsystems are completely different. We were able to use our framework to easily integrate the two different front-ends and use the same back-end to analyze the cause of failures, which shows modularity. We briefly describe the first two of these front-ends below. For logging in TOSSIM, interested readers are encouraged to read the manual for TOSSIM [31].

Data Collection Front-End for TinyOS

The event logger for MicaZ hardware is implemented using the TinyOS 2.0 BlockRead and BlockWrite interfaces to perform read and write operations respectively on flash. BlockRead and BlockWrite interfaces allow accessing the flash memory at a larger granularity which minimizes the
recording time to flash. To minimize the number of flash accesses we used a global buffer to accumulate events temporarily before writing to flash. Two identical buffers (buffer A and B) are used alternately to minimize the interference between event buffering and writing to flash. When buffer A gets filled up, buffer B is used for temporary buffering and buffer A is written to flash and vice versa. In Figure 2.6 we show the effect of buffer size on logging performance for single buffer and double buffer respectively. Using two buffers increases the logging performance substantially. As shown in figure, for event rate of 1000 events/second, using one buffer of 512 bytes has a success ratio (measured as the ratio of successfully logged events to the total number of generated events) of only 60% whereas using two buffers of 256 bytes each (512 bytes in total) can give almost 100% success ratio. For a rate of 200 events/second, two buffers of 32 bytes each is enough for 100% success ratio.

The sizes of these buffers are configurable as different applications need different amounts of runtime memory. It is to be noted that if the system crashes while some data are still in the RAM buffer, those events will be lost. The flash space layout is given in Figure 2.7.

A separate MicaZ mote (LogManager) is used to communicate with the logging subsystem to start and stop logging. Until the logging subsystem receives the “StartLogging” command, it will not log anything and after receiving “StopLogging” command it will flush the remaining data that is
in the buffer to flash and stop logging. This gives the user the flexibility to start and stop logging whenever they want. It also lets the user to run their application without enabling logging, when needed, to avoid the runtime overhead of logging functionality without recompiling the code.

We realize that occasional event reordering can occur due to preemption, interrupts, or task scheduling delays. An occasional invalid log entry is not a problem. An occasional incorrect logging sequence is fine too as long as the same occasional wrong sequence does not occur consistently. This is because common sequences do not have to occur every time, but only often enough to be noticed. Hence, they can be occasionally mis-logged without affecting the diagnostic accuracy.

**Time Synchronization:**

We need to timestamp the recorded events so that events recorded on different nodes can be serialized later during offline analysis. To avoid the overhead of running a time synchronization protocol on the application mote, we used an offline time synchronization scheme. A separate node (*TimeManager*) is used to broadcast its local clock periodically. The event logging component will receive the message and log it in flash with a local timestamp. From this information we can calculate the clock skew on different nodes in reference to *TimeManager* node, adjust the timestamp of the logged events and serialize the logs. We realize that the serialized log may not be exact but it is good enough for pattern mining.
System Overhead:

The event logging support requires 14670 bytes of program memory (this includes the code size for BlockRead and BlockWrite interface provided by TinyOS 2.0) and 830 bytes of data memory when 400 bytes are used for buffering (two buffers of 200 bytes each) data before writing to flash. User can choose to use less buffer space if the expected event rate is low. To instrument code, the program size increase is minimal. To log an event with no attributes, it needs a single line of code. To log an event with \( n \) attributes, it takes \( n + 1 \) lines of code, \( n \) lines are to initialize the record and 1 line to call the \texttt{log()} function.

API for Logging in TinyOS:

The only part of the data collection front-end that is exposed to the user is the interface for logging user defined events. Our design goal was to have an easy-to-use interface and efficient implementation to reduce the runtime overhead as much as possible. One critical issue with distributed logging was to timestamp the recorded events so that events on different nodes can be serialized later during offline analysis. To make event logging functionality simpler, we defined the interface to the logging component as follows:

\[
\text{log}(\text{EventId}, (\text{void} *) \text{buffer}, \text{unit8_t size})
\]

\texttt{log(EventId, (void *)buffer, unit8_t size)} is the key interface between application developers and the logging subsystem. To log an event, the user has to call the \texttt{log()} function with appropriate parameters. For example, if the user wants to log the event that a radio message was sent and also wants to log the receiverId along with the event, he/she needs to define the appropriate record structure in a header file (this file will also be used to parse the data) with these fields, initialize the record with appropriate values and call the \texttt{log} function with that record as the parameter. This simple function call will log the event. The rest is taken care of by the logging system underneath. The logging system will pad the timestamp with the recorded event and log as a single event. Note that \texttt{NodeId} information is added when data is uploaded to PC for offline analysis.
Data Collection Front-End for LiteOS

LiteOS [32] provides the required functionality to log kernel events on MicaZ platforms as follows. Specifically, the kernel logs events including system calls, radio activities, context switches and so on. An event log entry is a single 8-bit code without attributes. There is a specific macro for each system call that enables the logging for that specific call. The user has to call the specific Macro for that system call to enable logging that particular system call. The resulting log contains only the unique id of the system call that is sent to the serial port if that system call is invoked during the execution. This is done by calling a function “addTrace(systemCallId)” in each invocation of the system call if the corresponding macro value is set. The user requires to recompile the kernel to enable logging. We used an experimental set up of a debugging testbed with all motes connected to a PC via serial interfaces. In pre-deployment testing on our indoor testbed, logs can thus be transmitted in real-time through a programming board via serial communication with a base-station. When a system call is invoked or a radio packet is received on a node, the corresponding code for that specific event is transmitted through the serial port to the base station (PC). The base station collects event codes from the serial port and records it in a globally ordered file. Please note that the logging support for LiteOS is not a contribution of this thesis, and is described here for the purpose of completeness only.

2.5.2 The Data Analysis Back-End

At the back-end, we implement the data preprocessing and discriminative frequent pattern mining algorithm. To integrate the data collection front-end with the data preprocessing middleware, we provide a simple text file interface that describes the storage format of the raw byte streams collected for each of the front-ends. This file is used to parse the recorded events. Once the data is parsed, the user can either manually label the data files as “good” or “bad”, or the user can supply different predicates as a Java function that can be used to annotate data automatically. The rest
of the system is a collection of data analysis algorithms such as discriminative frequent pattern mining, or any other tool such as Weka [33]. Our algorithm automatically generates discriminative patterns and report it to the user.

2.6 Evaluation: Identifying the Timing Violations

In this section, we illustrate the effectiveness of our extension to identify timing violations described in Section 2.2.3 by injecting a bug in the directed diffusion protocol [3]. Directed diffusion [3] is a widely popular data centric communication protocol in wireless sensor network. For completeness, we briefly describe the design of the protocol below. In directed diffusion, any node that is interested in a particular type of data (e.g., detected vehicle in a particular region in a surveillance network) would first need to broadcast its “interest” in the network. This interest message includes the type of the data, geographic coordinates of interest, and the duration of the interest. Any node receiving the interest message would store that in its cache memory. Later when any node receives a data, it checks its interest cache to verify whether it is on the path, and whether it is supposed to forward that data message to the designated path or not. If the interest cache has no matching entry, it would drop the data silently assuming that it is not in the data forwarding path. In our implementation, the interested node needs to periodically send out a reinforcement message to renew its interest. If a particular node does not receive the reinforcement message within a certain period of time, the interest cache is expired, which subsequently prevents the node from reporting data. In our experiment, the reinforcement message is sent out every 500 ms. If no reinforcement message is received within 2 sec, the interest cache is expired. In our experiment, in the good cases, the reinforcement message is received as expected. But in bad cases, the node missed the reinforcement message occasionally. This leads to missing data occasionally at the base station. However, this does not lead to a permanent failure. As the interest message is sent out periodically, the source node eventually resumes transmission again. To troubleshoot the problem, we logged
Table 2.5: Log Statistics for the Timing Bug

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of good logs</td>
<td>1</td>
</tr>
<tr>
<td>Total number of bad logs</td>
<td>1</td>
</tr>
<tr>
<td>Average number of events per log</td>
<td>10400</td>
</tr>
</tbody>
</table>

Table 2.6: Discriminative Timing Events

<table>
<thead>
<tr>
<th>Discriminative Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <code>&lt;ReinforcementMessageReceived, ReinforcementMessageReceived, StdDev=8.0&gt;</code></td>
</tr>
<tr>
<td>2. <code>&lt;MessageSent, ReinforcementMessageReceived, StdDev=4.0&gt;</code></td>
</tr>
<tr>
<td>3. <code>&lt;ReinforcementMessageReceived, ReinforcementMessageReceived, StdDev=10.0&gt;</code></td>
</tr>
</tbody>
</table>

various events and applied our tool. The tool came up with the patterns listed in Table 2.6 as the top three discriminative patterns. From the patterns, it was obvious that the timing between consecutive reinforcement messages has something to do with the problem. For example, the first pattern indicates that, in failed cases, the timing gaps between consecutive ReinforcementMessages were larger than 8 times of standard deviation. This clue leads to the root cause very easily. It took about 7 seconds for the algorithm to terminate, which includes data pre-processing as well. We did not compare with our other algorithms as they are not designed to identify timing violations. Please note that, if the timing of events is not of concern, the user may choose to turn off the pre-processing stage that inserts events for timing violations, and avoid the additional overhead.

### 2.7 Evaluation: Identifying the Accumulative Effect

In this section, we illustrate the effectiveness of our extension to identify accumulative effect described in Section 2.2.4. We implemented a simple data logging application, where data is periodically flushed to the storage. To keep track of the size of the data on the flash, a counter is also updated whenever data is flushed to disk. Interestingly, data read operation occasionally returns garbage values for the most recent writes. Moreover, the number of garbage readings returned were different at different times.
To troubleshoot the problem, we logged various events related to data flush, data buffering, and counter update. Our algorithm returned three patterns as shown in Table 2.8. The top pattern returned clearly shows that, in bad cases, DataBuffered event is not followed by DataFlushDone event. Indeed, the problem arises if the node reboots or crashes for some reason after the counter is updated but before the data is completely written to disk. In such scenario, when reading back, we read some of the invalid regions of the flash. For correct operation, data buffering or counter update operation must be followed by data flush operation, which ensures that data is written to flash. The algorithm took 88 seconds to finish. Not surprisingly, when we applied our old algorithm, it failed to identify the problem and returned no pattern at all. Although this is a simple example, this highlights the effectiveness of our proposed extension in Section 2.2.4.

### Table 2.7: Log Statistics for the Accumulative Bug

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of good logs</td>
<td>1</td>
</tr>
<tr>
<td>Total number of bad logs</td>
<td>2</td>
</tr>
<tr>
<td>Average number of events per log</td>
<td>4625</td>
</tr>
</tbody>
</table>

Table 2.8: Discriminative Events for Accumulative Bug

<table>
<thead>
<tr>
<th>Discriminative Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.  &lt;DataFlushDone,DataBuffered&gt;</td>
</tr>
<tr>
<td>2.  &lt;DataFlushDone,updateCounterDone&gt;</td>
</tr>
<tr>
<td>3.  &lt;DataFlushDone,updateCounter&gt;</td>
</tr>
</tbody>
</table>

2.8 Evaluation: Progressive Discriminative Sequence Mining

Next, we troubleshoot three real life applications. The first was a kernel level bug in the LiteOS operating system. The second was to debug a multichannel Media Access Control (MAC) protocol [2] implemented in TinyOS 2.0 for MicaZ platform with only one half-duplex radio interface. In the third, we applied our tool to diagnose a protocol design bug [5] in the directed diffusion protocol [3].
We analyze the logs using the following algorithms depending on the applicability: (i) the basic sequence mining algorithm used in our earlier work [5] that does not incorporate the extensions described in section 2.2 (we call this the “Basic Algorithm”), (ii) the sequence mining algorithm that incorporates the extensions described in section 2.2 (we call this the “Extended Algorithm”), (iii) the sequence mining algorithm that incorporates the extensions described in section 2.2 and applies the two stage mining technique described in section 2.3.1 (we call this the “Extended Algorithm with Two Stage Mining”), and (iv) the sequence mining algorithm that incorporates the extensions described in section 2.2 and progressive discriminative analysis described in section 2.3.2 (we call this the “Extended Algorithm with Progressive Discriminative Analysis”).

<table>
<thead>
<tr>
<th>RecordedEvents</th>
<th>AttributeList</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context_Switch_To_User_Thread</td>
<td>Null</td>
</tr>
<tr>
<td>GetCurrent_Thread_Index</td>
<td>Null</td>
</tr>
<tr>
<td>GetCurrent_Radio_Info_Address</td>
<td>Null</td>
</tr>
<tr>
<td>GetCurrent_Radio_Handle_Address</td>
<td>Null</td>
</tr>
<tr>
<td>Post_Thread_Task</td>
<td>Null</td>
</tr>
<tr>
<td>Get_Serial_Mutex</td>
<td>Null</td>
</tr>
<tr>
<td>GetCurrent_Serial_Info_Address</td>
<td>Null</td>
</tr>
<tr>
<td>Get_Serial_Send_Function</td>
<td>Null</td>
</tr>
<tr>
<td>Disable_Radio_State</td>
<td>Null</td>
</tr>
<tr>
<td>Packet_Received</td>
<td>Null</td>
</tr>
<tr>
<td>Packet_Sent</td>
<td>Null</td>
</tr>
<tr>
<td>Yield_To_System_Thread</td>
<td>Null</td>
</tr>
<tr>
<td>GetCurrent_Thread_Address</td>
<td>Null</td>
</tr>
<tr>
<td>Get_Radio_Mutex</td>
<td>Null</td>
</tr>
<tr>
<td>Get_Radio_Send_Function</td>
<td>Null</td>
</tr>
<tr>
<td>Mutex_Unlock_Function</td>
<td>Null</td>
</tr>
<tr>
<td>GetCurrent_Radio_Handle</td>
<td>Null</td>
</tr>
</tbody>
</table>

Table 2.9: Logged events for diagnosing LiteOS application bug

### 2.8.1 Case Study - I: The LiteOS Bug

In this case study, we troubleshoot a simple data collection application where several sensors monitor light and report it to a sink node. The communication is performed in a single-hop environment.
In this scenario, sensors transmit packets to the receiver, and the receiver records received packets and sends an “ACK” back. The sending rate that sensors use is variable and depends on the variations in their readings. After receiving each message, depending on its sequence number, the receiver decides to record the value or not. If the sequence number is older than the last sequence number it has received, the packet is dropped.

This application is implemented using MicaZ motes on the LiteOS operating system and is tested on an experimental testbed. Each of the nodes is connected to a desktop computer via an MIB520 programming board and a serial cable. The PC acts as the base station. In this experiment, there was one receiver (the base node) and a set of 5 senders (monitoring sensors). This experiment illustrates a typical experimental debugging set up. Prior to deployment, programmers would typically test the protocol on target hardware in the lab. This is how such a test might proceed.

**Failure Scenario**

When this simple application was stress tested, some of the nodes would crash occasionally and non-deterministically. Each time different nodes would crash and at different times. Perplexed by the situation, the developer (a first-year graduate student with no prior experience with sensor networks) decided to log different types of events using LiteOS support and use our debugging tool. These were mostly kernel-level events along with a few application-level events. The built-in logging functionality provided by LiteOS was used to log the events. A subset of the different types of logged events are listed in Table 2.9.

Before presenting the results obtained by our algorithms, we will briefly describe the way a received packet is handled in the LiteOS and the real cause of the problem. In the application, receiver always registers for receiving packets, then waits until a packet arrives. At that time, the kernel switches back to the user thread with appropriate packet information. The packet is then processed in the application. However, at very high data rates, another packet can come when the processing of the previous packet has not yet been done. In that case, LiteOS kernel overwrites
the radio receive buffer with new information even if the user is still using the old packet data to process the previous packet. Indeed, for correct operation, `<Packet_Received>` event always has to be preceded by `<GetCurrent_Radio_Handle>` event. Otherwise, it crashes the system. After running the experiment, “good” logs were collected from the nodes that did not crash during the experiment and “bad” logs were collected from nodes that crashed at some point in time. We subsequently analyzed the logs as follows.

**The Basic Algorithm**

We implemented the basic algorithm used in [5] without incorporating the extensions described in section 2.2 and applied it to generate frequent patterns and perform differential analysis to extract discriminative patterns. For this case study, when we applied the basic algorithm to the “good” log and the “bad” log, the list of discriminative patterns missed the `<Packet_Received>` event completely and failed to identify the fact that the problem was correlated with the timing of packet reception. Moreover, when we applied the basic algorithm to multiple instances of “good” logs and “bad” logs together, the list of discriminative patterns returned was empty. All the frequent patterns generated by the basic algorithm were canceled at the differential phase. This result highlights the weakness of the sequence mining in general when applied for debugging and emphasize the necessity of our extensions as described in section 2.2.

<table>
<thead>
<tr>
<th>Total number of good logs</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of bad logs</td>
<td>3</td>
</tr>
<tr>
<td>Average number of events per log</td>
<td>324</td>
</tr>
</tbody>
</table>

Table 2.10: Log Statistics for the LiteOS Bug
The Extended Algorithm

After applying our discriminative frequent pattern mining algorithm that incorporates the extensions described in section 2.2 to the logs, we provided two sets of patterns to the developer, one set includes the highest ranked discriminative patterns that are found only in “good” logs as shown in Table 2.11 and the other set includes the highest ranked discriminative patterns that are found only in “bad” logs as shown in Table 2.12.

Based on the discriminative frequent pattern, it is clear that in “good” pile, \(<\text{PacketReceived}>\) event is highly correlated with the \(<\text{GetCurrentRadioHandle}>\) event. On the other hand, in the “bad” pile, though \(<\text{PacketReceived}>\) event is present, the other event is missing. In the “bad” pile, \(<\text{PacketReceived}>\) is highly correlated with \(<\text{GetSerialSendFunction}>\) event. From these observations, it is clear that proceeding with a \(<\text{GetSerialSendFunction}>\) when \(<\text{GetCurrentRadioHandle}>\) is missing is the most likely cause of failure.

The Extended Algorithm with Two Stage Mining

As two stage mining is more suitable for bugs that have frequent manifestations such as high number of message losses, we did not apply the two stage mining for this case study as the man-
manifestation of the problem (system crash) was infrequent in this case study. Two stage mining is applied for the case study presented in section 2.8.2 where the manifestation (message loss) of the bug is frequent.

The Extended Algorithm with Progressive Discriminative Analysis

To evaluate the performance improvement due to applying the progressive discriminative sequence mining scheme, we applied the progressive discriminative sequence mining algorithm on the same set of logs. It comes up with twenty six sequences of events as “culprit” sequences of events along with following one:

\[\text{<Get\_Current\_Radio\_Handle>, <Packet\_Received>, <Packet\_Received>}\]

This sequence explains the bug in one step. From this sequence it is obvious that if two consecutive messages are received following a single \(<\text{Get\_Current\_Radio\_Handle} >\) event, the system crashes. Indeed, \(<\text{Get\_Current\_Radio\_Handle} >\) event represents the required handle registration process and it must precede a message receive event.

One thing to note is that our earlier algorithm dropped it due to a particular setting of a threshold parameter that was used to measure the discriminative power of a particular sequence. As earlier we only used the support of a sequence within a file, due to a normalization factor this crucial sequence was dropped mistakenly. Missing this pattern by the extended algorithm described in Section 2.2 highlights the difficulty of parameter tuning that can affect the accuracy of the algorithm. Although with parameter tuning it is possible to capture this sequence, in many cases it is hard to guess the right values apriori. One of the main contributions of progressive discriminative mining is the enhancement in the scalability. To mine for the discriminative patterns, it took 127 seconds with progressive mining whereas the earlier algorithm took 248 seconds which is an improvement of almost 95%. We compare the effectiveness and performance of different schemes in Table 2.13.
<table>
<thead>
<tr>
<th>Algorithm Used</th>
<th>Runtime</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Algorithm</td>
<td>N/A</td>
<td>Need the extensions presented in Section 2.2</td>
</tr>
<tr>
<td>Extended Algorithm</td>
<td>248 sec</td>
<td></td>
</tr>
<tr>
<td>Extended Algorithm with Two Stage Mining</td>
<td>N/A</td>
<td>Effect of the problem is infrequent. Requires manual parameter tuning.</td>
</tr>
<tr>
<td>Extended Algorithm with Progressive</td>
<td>127 sec</td>
<td></td>
</tr>
<tr>
<td>Discriminative Analysis</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.13: Comparison of different schemes for the LiteOS bug

### 2.8.2 Case Study - II: Multichannel MAC Protocol

In this case study, we debug a multichannel MAC protocol [2]. The objective of the protocol used in our study is to assign a home channel to each node in the network dynamically in such a way that the throughput is maximized. The design of the protocol exploits the fact that in most wireless sensor networks, the communication rate among different nodes is not uniform (e.g., in a data aggregation network). Hence, the problem was formulated in such a way that nodes communicating frequently are clustered together and assigned the same home channel whereas nodes that communicate less frequently are clustered into different channels. This minimizes overhead of channel switching when nodes need to communicate.

During experimentation with the protocol, it was noticed that when data rates between different internally closely-communicating clusters is low, the multi-channel protocol outperforms a single channel MAC protocol comfortably as it should. However, when the data rate between clusters was increased, while the throughput near the base station still outperformed a single channel MAC significantly, nodes further from the base station were performing worse than in the single channel MAC. This should not have happened in a well-designed protocol as the multichannel MAC protocol should utilize the communication spectrum better than a single channel MAC. The author of the protocol initially concluded that the performance degradation was due to the overhead associated with communication across clusters assigned to different channels. Such communication entails
frequent channel switching as the sender node, according to the protocol, must switch the frequency to the receiver before transmission, then return to its home channel. This incurs overhead that increases with the transmission rate across clusters. We decided to verify this conjecture. As a stress test of our tool, we instrumented the protocol to log events related to the MAC layer (such as message transmission and reception as well as channel switching) and used our tool to determine the discriminative patterns generated from different runs with different message rates, some of which performing better than others. For better understanding of the failure scenario detected, we briefly describe the operation of the multichannel MAC protocol below.

**Multichannel MAC Protocol Overview**

In the multichannel MAC protocol, each node initially starts at channel 0 as its home channel. To communicate with others, every node maintains a data structure called “neighbor table” that stores the neighbor home channel for each of its neighboring nodes. Channels are organized as a ladder, numbered from lowest (0) to highest (12). When a node decides to change its home channel, it sends out a “Bye” message in its current home channel which includes its new home channel number. Receiving a “Bye” message, each other node updates its neighbor table to reflect the new home channel number for the sender of the “Bye” message. After changing its home channel, a node sends out a “Hello” message in the new home channel which includes its nodeID. All neighboring nodes on that channel add this node as a new neighbor and update their neighbor tables accordingly.

To increase robustness to message loss, the protocol also includes a mechanism for discovering the home channel of a neighbor when its current entry in the neighbor table becomes stale. When a node sends a message to a receiver on that receiver’s home channel (as listed in the neighbor table) but does not receive an “ACK” after ’n’ (n is set to 5) tries, it assumes that the destination node is not on its home channel. The reason may be that the destination node has changed its home channel permanently but the notification was lost. Instead of wasting more time on retransmissions
Table 2.14: Logged events for diagnosing multichannel MAC protocol

<table>
<thead>
<tr>
<th>RecordedEvents</th>
<th>AttributeList</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ack_Received</td>
<td>Null</td>
</tr>
<tr>
<td>Home_Channel_Changed</td>
<td>oldChannel, newChannel</td>
</tr>
<tr>
<td>TimeSyncMsg</td>
<td>referenceTime, localTime</td>
</tr>
<tr>
<td>Channel_Update_Msg_Sent</td>
<td>homeChannel</td>
</tr>
<tr>
<td>Data_Msg_Sent_On_Same_Channel</td>
<td>destId, homeChannel</td>
</tr>
<tr>
<td>Data_Msg_Sent_On_Different_Channel</td>
<td>destId, homeChannel, destChannel</td>
</tr>
<tr>
<td>Channel_Update_Msg_Received</td>
<td>homeChannel, neighborId, neighborChannel</td>
</tr>
<tr>
<td>Retry_Transmission</td>
<td>oldChannelTried, nextChannelToTry</td>
</tr>
<tr>
<td>No_Ack_Received</td>
<td>Null</td>
</tr>
</tbody>
</table>

on the same channel, the sender starts scanning all channels, asking if the receiver is there. The purpose is to find the receiver’s new home channel and update the neighbor table accordingly. The destination node will eventually hear this data message and reply when it is on its home channel. Since the above mechanism is expensive, as an optimization, overhearing is used to reduce staleness of the neighbor table. Namely, a node updates the home channel of a neighbor in its neighbor table when the node overhears an acknowledgement (“ACK”) from that neighbor sent on that channel. Since the “ACK”s are used as a mechanism to infer home channel information, whenever a node switches channels temporarily (e.g., to send to a different node on the home channel of the latter), it delays sending out “ACK” messages until it comes back to its home channel in order to prevent incorrect updates of neighbor tables by recipients of such ACKs.

Finally, to estimate channel conditions, each node periodically broadcasts a “channelUpdate” message which contains the information about successfully received and sent messages during the last measurement period (where the period is set at compile time). Based on that information, each node calculates the channel quality (i.e., probability of successfully accessing the medium), and uses that measure to probabilistically decide whether to change its home channel or not. Nodes that sink a lot of traffic (e.g., aggregation hubs or cluster heads) switch first. Others that communicate heavily with them follow. This typically results into a natural separation of node clusters into different frequencies so they do not interfere.
<table>
<thead>
<tr>
<th>Total number of good logs</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of bad logs</td>
<td>1</td>
</tr>
<tr>
<td>Average number of events per log</td>
<td>40000</td>
</tr>
</tbody>
</table>

Table 2.15: Log Statistics for the MAC channel Bug

**Performance Problem**

This protocol was executed on 16 MicaZ motes implementing an aggregation tree where several aggregation cluster-heads filter data received from their children, significantly reducing the amount forwarded, then send that reduced data to a base-station. When the data rate across clusters was low, the protocol outperformed the single channel MAC. However, when the data rate among clusters was increased, the performance of the protocol deteriorated significantly, performing worse than a single channel MAC in some cases. The developer of the protocol assumed that this was due to the overhead associated with the channel change mechanism which is incurred when communication happens among different clusters heavily. Much debugging effort was spent on that direction with no result. To diagnose the cause of the performance problem, we logged different types of MAC events as listed in Table 2.14.

The question posed to our tool was “Why is the performance bad at higher data rate?” To answer this question, we first executed the protocol at low data rates (when the performance is better than single channel MAC) to collect logs representing “good” behavior. We then again executed the protocol with a high data rate (when the performance is worse than single channel MAC) to collect logs representing “bad” behavior. We subsequently analyzed the logs as follows.

**The Basic Algorithm**

Using the basic algorithm, to generate frequent patterns of length 2 for 40000 events in the “good” log, it took 1683.02 seconds (28 minutes) and to finish the whole computation including differential analysis it took 4323 seconds (72 minutes). We tried to generate frequent patterns of length 3 with the approach in [5] but terminated the process after one day of computation that remained in
progress. We used a machine of 2.53 GHz speed and 512 MB RAM. This highlights the scalability problem.

The Extended Algorithm with Two Stage Mining

With our two-stage mining scheme, it took 5.55 seconds to finish the first stage and finishing the whole computation including differential analysis took 332.92 seconds (6 minutes). After performing discriminative pattern analysis, the list of top 5 discriminative patterns that were produced by our tool is shown in Table 2.16.

<table>
<thead>
<tr>
<th>&lt;No Ack Received&gt;, &lt;Retry Transmission&gt;</th>
<th>&lt;No Ack Received&gt;, &lt;Retry Transmission&gt;, &lt;Data Msg Sent On Same Channel: homechannel: 0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;No Ack Received&gt;, &lt;Retry Transmission&gt;, &lt;Data Msg Sent On Same Channel: homechannel: 0&gt;, &lt;No Ack Received&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;No Ack Received&gt;, &lt;Retry Transmission&gt;, &lt;Data Msg Sent On Same Channel: homechannel: 0&gt;, &lt;Retry Transmission: nextchanneltotry: 1&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;No Ack Received&gt;, &lt;Retry Transmission: oldchanneltried: 1&gt;, &lt;Retry Transmission&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;No Ack Received&gt;, &lt;Retry Transmission&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;No Ack Received&gt;, &lt;Retry Transmission: nextchanneltotry: 1&gt;, &lt;Retry Transmission: nextchanneltotry: 2&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.16: Discriminative frequent patterns for multichannel MAC protocol

The sequences indicate that, in all cases, there seems to be a problem with not receiving acknowledgements. Lack of acknowledgements causes a channel scanning pattern to unfold. This is shown
as the *<Retry_Transmission>* event on different channels, as a result of not receiving acknowledgements. Hence, the problem does not lie in the frequent overhead of senders changing their channel to that of their receiver in order to send a message across clusters. The problem lied in the frequent lack of response (an ACK) from a receiver. At the first stage of frequent pattern mining *<No_Ack_Received>* is identified as the most frequent event. At the second stage, the algorithm searched for frequent patterns in top *K* (e.g., top 5) segments of the logs where *<No_Ack_Received>* event occurred with highest frequency. The second stage of the log analysis (correlating frequent events to preceding ones) then uncovered that the lack of an ACK from the receiver is preceded by a temporary channel change. This gave away the bug. As we described earlier, whenever a node changes its channel temporarily, it disables “ACK”s until it comes back to its home channel. In a high intercluster communication scenario, disabling the “ACK” is a bad decision for a node that spends a significant amount of time communicating with other clusters on channels other than its own home channel. As a side effect, nodes which are trying to communicate with it fail to receive an “ACK” for a long time and start scanning channels frequently looking for the missing receiver. Another interesting aspect of the problem that was discovered is the cascading effect of the problem. When we look at generated discriminative patterns across multiple nodes, we found that the scanning patterns revealed in the logs shown in fact cascades. Channel scanning at the destination node often triggers channel scanning at the sender node and this interesting cascaded effect was also captured by our tool.

The Extended Algorithm with Progressive Discriminative Analysis

Progressive discriminative mining returned in 14 seconds and returned the 59 single events as highly correlated to poor performance. The top events were the following:

*<No_Ack_Received>, <Retry_Transmission>, <Data_Msg_Sent_On_Different_Channel>,<Retry_Transmission: oldchanneltried: 0>, <Retry_Transmission: nextchanneltotry: 1>,<Retry_Transmission: oldchanneltried: 1>, <Retry_Transmission: nextchanneltotry: 2>,
<table>
<thead>
<tr>
<th>Algorithm Used</th>
<th>Runtime</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Algorithm</td>
<td>N/A</td>
<td>Too Slow</td>
</tr>
<tr>
<td>Extended Algorithm</td>
<td>N/A</td>
<td>Too Slow</td>
</tr>
<tr>
<td>Extended Algorithm with Two Stage Mining</td>
<td>333 sec</td>
<td>Effect of the problem (msg loss)is frequent</td>
</tr>
<tr>
<td>Extended Algorithm with Progressive Discriminative Analysis</td>
<td>14 sec</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.17: Comparison of different schemes for multichannel MAC Protocol bug

Although these were all single events, in this case study it would have been adequate to provide the necessary insights to the real problem. The designer of the protocol can readily understand the channel scanning phenomenon that was happening at high intercluster data rate. We compare the effectiveness and performance of different schemes in Table 2.17.

2.8.3 Case Study - III: Directed Diffusion Protocol Bug

In the directed diffusion protocol [3], if a node gets rebooted for some reason, it erases the interest cache completely and would result in a broken path if there is only a single path from the source to the sink node and the rebooted node is on that critical path. Due to this design flaw, there would be a large number of consecutive message losses following a reboot. To compare the scalability
<table>
<thead>
<tr>
<th>Total number of good logs</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of bad logs</td>
<td>3</td>
</tr>
<tr>
<td>Average number of events per log</td>
<td>24000</td>
</tr>
</tbody>
</table>

Table 2.18: Log Statistics for the Directed Diffusion Bug

enhancement due to progressive discriminative algorithm, we collected three good logs (when no node was rebooted) and three bad logs (when a node was rebooted), and evaluated using multiple algorithms as follows.

**The Basic Algorithm**

We applied the basic algorithm used in our earlier work [5] on six logs (three good logs and three bad logs). We configured the algorithm to generate frequent patterns of length up-to 3. The algorithm failed to finish after six hours of computation.

**The Extended Algorithm with Two Stage Mining**

Next, we applied the extended algorithm used in our prior work [6] with two stage mining and configured the algorithm to generate frequent patterns of length up-to 3. The algorithm finished in about 2.5 hours. Unfortunately, it returned several thousands of patterns. Moreover, although the algorithm was able to identify the “Reboot” event as correlated to failure, it was ranked at the very end of the list due to low support and increased the chance of being overlooked by the developer as unimportant pattern.

**The Extended Algorithm with Progressive Discriminative Analysis**

In comparison, progressive discriminative sequence mining finished in about 5 seconds and it returned only seven individual events as correlated to failure. Four of these seven events are listed below:
Algorithm Used | Runtime | Comments
--- | --- | ---
Basic Algorithm | N/A | Too Slow
Extended Algorithm | N/A | Too Slow
Extended Algorithm with Two Stage Mining | 2.5 hr | Failed to rank the Reboot event at the top
Extended Algorithm with Progressive Discriminative Analysis | 5 sec | 

Table 2.19: Comparison of different schemes for Directed diffusion protocol bug

\(< \text{BootEvent} : (\text{NodeId} : X) >, \)
\(< \text{interestCacheEmpty} : (\text{NodeId} : X) >, \)
\(< \text{dataCacheEmpty} : (\text{NodeId} : X) >, \) and
\(< \text{msgDropped} : (\text{ReasonToDrop} : \text{dataWithNoMatchingInterest}) >.\)

The reason for such drastic improvement is that the progressive mining strategy reduced the search space significantly by applying the discriminative analysis at each stage. Another important characteristic is that it reduced the number of final patterns returned from several thousands to only a few which enhances the usability of the tool significantly. We compare the effectiveness and performance of different schemes in Table 2.19.

2.8.4 Debugging Overhead

To test the impact of logging on application behavior, we ran the multichannel MAC protocol with logging enabled and without logging enabled with both moderate data rate and high data rate. The network was set as a data aggregation network.

For moderate data rate experiment, the source nodes (node that only sends messages) were set to transmit data at a rate of 10 messages/sec, the intermediate nodes were set to transmit data at a rate of 2 messages/sec and one node was acting as the base station (which only receives messages). We tested this on a 8 nodes network with 5 source nodes, 2 intermediate nodes and one base station. Over multiple runs, after we take the average to get a reliable estimate, average number of successfully transmitted messages was increased by $9.57\%$ and average number of successfully
received messages was increased by 2.32%. The most likely reason for this minor improvement is writing to flash was creating a randomization effect which probably helped to reduce interference at the MAC layer.

At high data rate, source nodes were set to transmit data at a rate of 100 messages/sec and intermediate nodes were set to transmit data at a rate of 20 messages/sec. Over multiple runs, after we take the average to get a reliable estimate, average number of successfully transmitted messages was reduced by 1.09% and average number of successfully received messages was dropped by 1.62%. The most likely reason is the overhead of writing to flash kicked in at such a high data rate and eventually reduced the advantage experienced at a low data rate.

The performance improvement of the multichannel MAC protocol reported in this dissertation is obtained by running the protocol at the high data rate to prevent over estimation.

We realize that this effect on application may change the behavior of the original application slightly, but that effect seems to be negligible from our experience and did not affect the diagnostic capability of the discriminative pattern mining algorithm which is inherently robust against minor statistical variance.

As multichannel MAC protocol did not use flash memory to store any data, we were able to use the whole flash for logging events. To test the relation between quality of generated discriminative patterns and the logging space used, we used 100KB, 200KB and 400KB of flash space in three different experiments. The generated discriminative patterns were similar. We realize that different application has different amount of flash space requirements and the amount of logging space may affect the diagnostic capability. To help in severe space constraints, we provide the radio interface so users can choose to log at different times instead of logging continuously. User can also choose to log events at different resolutions.

For LiteOS case study, we did not use flash space at all as the events were transmitted to basestation (PC) directly using serial connection and eliminate the flash space overhead completely which makes our tool easily usable for testbeds which often provides serial connections.
2.8.5 Summary

From the above evaluation we can draw the following conclusions. First, the changes as described in section 2.2 that were made to the basic sequence mining algorithm to adapt it for debugging is critical for effective diagnosis. Second, progressive discriminative analysis is extremely fast compared to the two stage mining algorithm presented in Dustminer [6]. Although, in some cases it may return single events as correlated to failure, these events can be used as the clues to begin with and can be further explored if the user of the tool desires. Third, progressive mining has an automatic way of identifying when to stop the mining process. For example, in the LiteOS bug case study, it stopped after generating patterns of length 3 when the set of discriminative patterns became non empty, and in case of the MAC protocol bug it stopped right after mining patterns of length 1. Earlier we had to guess and set the pattern length conservatively which often wastes a lot of time for mining longer patterns and returns too many patterns. Finally, the two stage mining is not suitable in cases where the manifestation of the problem is not frequent (e.g., the bug found in the LiteOS operating system has infrequent manifestation and cause, a single reordering of events followed by system crash).
Chapter 3
Symbolic Bug Patterns

In this part of our thesis, we investigate a novel approach for failure diagnosis by summarizing and generalizing patterns that lead to instances of anomalous behavior in sensor networks\(^\text{1}\). Distributed systems such as wireless sensor network applications typically implement distributed protocols where multiple nodes communicate with each other to collectively perform a collaborative task. Nodes often assume roles such as cluster heads, sensors, or forwarding nodes. Messages have types, usually defined by the respective applications. Often multiple seemingly different event patterns lead to the same type of failure manifestation. A hidden relationship exists, in those patterns, among event attributes that is somehow responsible for the failure. For example, a target tracking system may fail only if a leader node receives a message of certain type irrespective of the absolute nodeIDs involved.

Unfortunately, none of the existing debugging tools and techniques available for sensor networks is capable of troubleshooting symbolic bugs. Dustminer\([6]\), which is described in the earlier part of this thesis, although makes an effort towards using sequence mining for finding interaction bugs, in this part of the thesis we show that analyzing sequences of events based on absolute event attribute values to diagnose such bugs may not be enough. In addition, to make things worse, the resulting patterns can often be misleading and can confuse the application developer.

To address this key challenge, in this part of our thesis, we introduce the concept of *symbolic patterns* that identify the “culprit” sequences of events responsible for failure by capturing the relationships among different event attributes. We generalize from actual observed message ex-

\(^{1}\text{Part of the work presented in this chapter was done in collaboration and was published earlier [7].}\)
changes to the underlying relationships defined on nodes, roles, and message types, that lead to a failure. We call them *symbolic bug patterns*. In the context of this thesis, a symbolic pattern is a pattern where all or a subset of the absolute values of event attributes within the pattern are replaced with *symbols* to generalize the pattern. To perform offline analysis using symbolic pattern extraction, different types of runtime events are logged during program execution and offline analysis is done to identify the discriminative set of frequent symbolic patterns that will contain the “culprit” symbolic patterns that are highly correlated to failure.

### 3.1 A Model for Symbolic Patterns

The logged events in our system can include any operations performed at runtime such as message transmission, message reception, and writing to flash storage. Each recorded event can have multiple attributes. For example, a message transmission event can have senderId, senderType, destinationId and msgType as attributes.

For example, consider the following logged events in a sample log:

\[
< \text{msgSent}, \text{senderId} = 1, \text{msgType} = 0, \text{destinationId} = 3 >
\]

\[
< \text{msgReceived}, \text{receiverId} = 1, \text{msgType} = 1, \text{senderId} = 3 >
\]

\[
< \text{flashWriteInitiated}, \text{nodeId} = 1, \text{dataSize} = 100 >
\]

The above log can be considered a single sequence of three events each with multiple attributes. The frequent sequence mining algorithm used in Dustminer [6] is used to extract frequent subsequences of events. Events in the subsequence do not have to be contiguous in the original sequence. We use the term “frequent (sub)sequence of events” and “frequent pattern” interchangeably in this thesis.

As described before, a *discriminative pattern* between two logs is an ordered subsequence of events that occurs with a different *support* in the two logs, where support refers to the number of times it occurs. The larger the difference in support, the better the discriminative power. Before we formally define
a symbolic pattern, let us consider the following example to illustrate what it means. Say, we have two patterns S1 and S2 where each pattern has two events with multiple attributes as follows:

\[ S1 = \langle \text{msgSent, senderId} = 1, \text{msgType} = 0 \rangle \]
\[ \langle \text{msgReceived, receiverId} = 2, \text{msgType} = 0 \rangle \]
\[ S2 = \langle \text{msgSent, senderId} = 3, \text{msgType} = 0 \rangle \]
\[ \langle \text{msgReceived, receiverId} = 5, \text{msgType} = 0 \rangle \]

where node 1 is the neighbor of node 2 and node 3 is the neighbor of node 5. On the surface, patterns S1 and S2 are different. Now, if we parameterize the relationship that exists between senderId and receiverId and represent it using symbol \( x \) for senderId, S1 and S2 can be represented as follows:

\[ S1 = \langle \text{msgSent, senderId} = x, \text{msgType} = 0 \rangle \]
\[ \langle \text{msgReceived, receiverId} = \text{neighbor}(x), \text{msgType} = 0 \rangle \]
\[ S2 = \langle \text{msgSent, senderId} = x, \text{msgType} = 0 \rangle \]
\[ \langle \text{msgReceived, receiverId} = \text{neighbor}(x), \text{msgType} = 0 \rangle \]

Interestingly, S1 and S2 now become the same pattern which expresses a more general relationship. Note that, if S1 and S2 each has support 1, the symbolic version has support 2 and hence symbolizing patterns increase the visibility of the pattern in the event log.

More formally, in the context of this thesis, symbolic pattern extraction is the task of identifying frequent patterns that satisfy certain relationships, specified by the user or selected from a library of common relationships, defined among event attributes of same or different types of events (e.g., neighborhood relationship, identity relationship, and type relationships). These relationships are then represented using symbols instead of absolute values where appropriate. In this thesis, we present an algorithm for symbolic pattern extraction which, at first, generates frequent patterns using the algorithm used in Dustminer [6]. However, we do not use the two stage mining. Next, it generalizes frequent patterns generated in the first stage by mining for “relationships” in those patterns. In this part of the thesis, we present a hybrid scheme for counting support (similar to the concept of “across support”) which greatly

59
enhances the chance of identifying “infrequent” events that are correlated with failure. Finally, we propose a pattern ranking scheme exploiting the characteristics of symbolic patterns which increases the usability of the tool.

To analyze the performance of symbolic pattern extraction, we simulated several bugs in TOSSIM to generate log files and analyzed them using our new algorithm. We choose simulation to generate log files as it gives us the flexibility to experiment with bugs of arbitrary complexity. We compare discriminative symbolic patterns generated by our symbolic pattern extraction algorithm with the discriminative patterns generated by the algorithm we presented earlier [6] and show that symbolic patterns greatly enhance the diagnostic capability and the usability of the tool.

3.2 Overview

To answer the question “Why do we need discriminative symbolic pattern extraction to debug interaction bugs?”, we provide an example in Section 3.2.1. We then present the symbolic pattern extraction algorithm in Section 3.2.2. We conclude the section by presenting a hybrid support count function and a pattern ranking scheme that have significant impact on the quality of the patterns generated and the scalability of the algorithm.

3.2.1 Motivation for Using Symbolic Pattern for Debugging

Let us assume that, in a particular application, each neighbor of node A periodically communicates with node A and is always expected to send messages of type 0, 1 and 2 in a fixed order, where msgType 0 is followed by msgType 1 and msgType 2, respectively. Also, assume that any violation of this message order from a specific sender crashes the system. Now, let us log a few examples of correct execution (Good Log) and execution that leads to a manifestation of error (Bad Log). Consider the log file presented in Table 3.1 collected from node 1 and node 7, where node 1 did not crash and node 7 crashed. Note that, node 7 crashed as node 8 sent messages violating the
Table 3.1: Sample Log File

<table>
<thead>
<tr>
<th>Good Log (Node 1)</th>
<th>Bad Log (Node 7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. &lt;msgReceived, receiverId = 1, senderId = 3, msgType = 0&gt;</td>
<td>1. &lt;msgReceived, receiverId = 7, senderId = 6, msgType = 0&gt;</td>
</tr>
<tr>
<td>2. &lt;msgReceived, receiverId = 1, senderId = 3, msgType = 1&gt;</td>
<td>2. &lt;msgReceived, receiverId = 7, senderId = 6, msgType = 1&gt;</td>
</tr>
<tr>
<td>3. &lt;msgReceived, receiverId = 1, senderId = 3, msgType = 2&gt;</td>
<td>3. &lt;msgReceived, receiverId = 7, senderId = 6, msgType = 2&gt;</td>
</tr>
<tr>
<td>4. &lt;msgReceived, receiverId = 1, senderId = 2, msgType = 0&gt;</td>
<td>4. &lt;msgReceived, receiverId = 7, senderId = 8, msgType = 0&gt;</td>
</tr>
<tr>
<td>5. &lt;msgReceived, receiverId = 1, senderId = 2, msgType = 1&gt;</td>
<td>5. &lt;msgReceived, receiverId = 7, senderId = 8, msgType = 1&gt;</td>
</tr>
<tr>
<td>6. &lt;msgReceived, receiverId = 1, senderId = 2, msgType = 2&gt;</td>
<td>6. &lt;msgReceived, receiverId = 7, senderId = 6, msgType = 2&gt;</td>
</tr>
</tbody>
</table>

required sequence of message types. If we generate the patterns correlated with failure, a state of
the art algorithm would come up with the following: pattern \( seq_1 \) with support 2 and pattern \( seq_2 \)
with support 1 along with other frequent patterns.

\[
\begin{align*}
seq_1 &= \langle \text{msgReceived}, \text{msgType} = 0 \rangle \\
&\quad \langle \text{msgReceived}, \text{msgType} = 1 \rangle \\
&\quad \langle \text{msgReceived}, \text{msgType} = 2 \rangle \\
seq_2 &= \langle \text{msgReceived}, \text{msgType} = 2 \rangle \\
&\quad \langle \text{msgReceived}, \text{msgType} = 0 \rangle \\
&\quad \langle \text{msgReceived}, \text{msgType} = 1 \rangle 
\end{align*}
\]

If we inspect the logged events presented in Table 3.1 carefully, we can see that there is also a
pattern associated with the senderId, where senderId is the same for \( seq_1 \). For the first and second
occurrence of \( seq_1 \), senderId is 3 and 2 respectively. This pattern is missed due to different support.
For example, logged event \( \langle \text{msgReceived}, \text{receiverId} = 1, \text{senderId} = 3, \text{msgType} = 0 \rangle \) has support 1 and
\( \langle \text{msgReceived}, \text{receiverId} = 1, \text{msgType} = 0 \rangle \) has support 3.

On the other hand, if we parameterize the values of receiverId and senderId and replace the identical
values with symbols where \( \text{receiverId} \) is replaced with \( X \) and \( \text{senderId} \) with \( Y \) in Good Log, the
following pattern will be identified with support 2 where for the first occurrence \( X = 1 \) and \( Y = 3 \) and for the second occurrence \( X = 1 \) and \( Y = 2 \).

\[
\begin{align*}
< \text{msgReceived}, \text{receiverId} = X, \text{senderId} = Y, \text{msgType} = 0 > \\
< \text{msgReceived}, \text{receiverId} = X, \text{senderId} = Y, \text{msgType} = 1 > \\
< \text{msgReceived}, \text{receiverId} = X, \text{senderId} = Y, \text{msgType} = 2 >
\end{align*}
\]

Similarly, if we extract symbolic patterns, we would be able to identify that the following pattern \( \text{seq}_3 \) occurs only in Bad log but not in the Good log -

\[
\begin{align*}
< \text{msgReceived}, \text{receiverId} = X, \text{senderId} = Y, \text{msgType} = 2 > \\
< \text{msgReceived}, \text{receiverId} = X, \text{senderId} = Y, \text{msgType} = 0 > \\
< \text{msgReceived}, \text{receiverId} = X, \text{senderId} = Y, \text{msgType} = 1 >
\end{align*}
\]

Without symbolic pattern extraction there is no way of identifying \( \text{seq}_3 \). A more detailed description of the Symbolic pattern extraction algorithm is presented in Section 3.2.2.

### 3.2.2 Symbolic Pattern Extraction Algorithm

Symbolic pattern extraction is a two step process.

- During the first stage, multiattribute events are converted into single attribute events to reduce the computational complexity. Frequent patterns of events with single attribute are generated using the sequence mining algorithm used in Dustminer [6]. Let us call this set of frequent pattern the base\_frequent\_set.

- At the second stage, the candidate symbolic patterns are generated from this base\_frequent\_set. If the symbolic pattern \( s_i \) has support \( \text{sup}_{s_i} \) which is generated from the base pattern \( p_i \) with support \( \text{sup}_{p_i} \) and \( (\text{sup}_{s_i}/\text{sup}_{p_i}) > \delta \), then \( p_i \) is replaced by \( s_i \). \( \delta \) is the equivalence threshold which is set by the user. If \( \delta \) is set to 0, all the symbolic patterns are retained and if it is set to 1 then symbolic patterns with the exact same support as the base pattern are retained. The generation of candidate symbolic patterns is described below.
Generation of Candidate Symbolic Patterns

To explain the generation of candidate symbolic patterns, without loss of generality, let us assume that $Seq_a$ is a frequent base pattern of three events where each event is of different type and includes a single attribute from each event type.

$Seq_a = (E_x, attr_2 = v_i, E_y, attr_2 = v_j, E_z, attr_3 = v_k)$

Say, event $E_x$ originally has 3 attributes and $Seq_a$ includes only the second attribute of $E_x$. Similarly, we assume $E_y$ and $E_z$ originally have 2 and 3 attributes respectively.

Next, the algorithm reconstructs the equivalent, complete pattern where each event has all the attributes. Now, the equivalent pattern generated from $Seq_a$ would look like as follows:

$$(E_x, attr_1 = *, E_x, attr_2 = v_i, E_x, attr_3 = *)$$

$$(E_y, attr_1 = *, E_y, attr_2 = v_j)$$

$$(E_z, attr_1 = *, E_z, attr_2 = *, E_z, attr_3 = v_k)$$

Here "*" is used for the attributes that are not included in the original pattern which basically says that the "*" attributes are “don’t care”. Next, the algorithm replaces a subset of the "*" attributes with symbols and mine for relationship among those symbolic attributes. The symbolic pattern replaces $Seq_a$ if the support of the symbolic pattern in the original log is “similar” to the support of $Seq_a$.

3.2.3 Challenges

Meaningful Condition Identification

"Which subset of “*” attributes to replace with symbols and what “relationship” to test for?" is one of the key questions in finding meaningful symbolic bug patterns. We need to decide this intelligently.
to avoid useless checking such as “checking if nodeId and timeStamp are equal or not in a particular event”. Our goal is to automate the process as much as possible. To reduce the user involvement, we provide a list of predefined conditions that are especially applicable for wireless sensor network applications. For example, the common attributes expected for wireless sensor network applications are nodeId, message types, sensor data types, timestamps, etc. We tried to come up with the basic conditions that need to be checked. For example, checking if the “neighbor” condition holds between senderId and receiverId makes sense. The user needs to specify the type of the attribute in a header file. For example, if the type of $i^{th}$ attribute of event $E_x$ is “nodeId”, user may specify that information as $(< E_x, attr_i, type : nodeId>)$ from which the tool automatically determines the set of applicable conditions for this attribute. From that information, combinations of conditions of arbitrary complexity such as “Is the senderId always same as the receiverId?”, or “Does the msgType has to be X and sender has to be the immediate neighbor to crash the receiver?” and so on can be generated automatically.

We realize that there may be conditions which are not provided by us. If the user wants to check for conditions that are not provided by a library function, he/she may implement the desired condition and add it to our library. A user may specify a condition that is not provided by the tool as follows:

$$(< E_x, attr_i >, < E_y, attr_j >, Condition_q)$$

where $Condition_q$ is defined and implemented by the user for his/her specific application. A pseudocode of the algorithm is given in Table 3.2.

### Scalability

One of the problems with symbolic pattern mining is that the number of combinations of conditions to check is exponential. For example, consider the following symbolic candidate pattern -
Algorithm: Symbolic Pattern Extraction

**Input:** Set of Good Logs (GL), Set of Bad Logs (BL), similarity measure ($\delta$)

**Output:** Set of discriminative symbolic pattern

1. PatternSetA = GenerateFrequentPatterns(GL)
2. SymbolicPatternSetA = ExtractSymbolicPattern(PatternSetA, GL, $\delta$)
3. PatternSetB = GenerateFrequentPatterns(BL)
4. SymbolicPatternSetB = ExtractSymbolicPattern(PatternSetB, BL, $\delta$)
5. DiscriminativePatternSet = DiffMine(SymbolicPatternSetA, SymbolicPatternSetB)
6. output DiscriminativePatternSet

Function: ExtractSymbolicPattern

**Input:** Set of Frequent Pattern (FP), Set of Logs (L), similarity measure ($\delta$)

**Output:** Set of symbolic pattern (SP)

1. SP = Null; /* set of symbolic pattern */
2. for each pattern $p$ in FP
   2.1 for each check condition $c$
      2.1.1 CSP = GenerateCandidateSymbolicPattern($p$, $c$)
      2.1.2 if (support(CSP)/support(p) > $\delta$) then SP = SP U CSP
2. return SP

Table 3.2: Symbolic Pattern Extraction Algorithm

\[
\begin{align*}
(<E_x, attr_1 = *, <E_x, attr_2 = v_i>, <E_x, attr_3 = *>) \\
(<E_y, attr_1 = *, <E_y, attr_2 = v_j>) \\
(<E_z, attr_1 = *, <E_z, attr_2 = v_k>)
\end{align*}
\]

Now, assume that the applicable set of conditions that need to be checked for this pattern are:

\[
c_1 : (<E_x, attr_1 >, <E_y, attr_1 >, IdentityCondition) \\
c_2 : (<E_x, attr_1 >, <E_x, attr_1 >, IdentityCondition) \\
c_3 : (<E_y, attr_1 >, <E_x, attr_1 >, IdentityCondition) \\
c_4 : (<E_x, attr_2 >, <E_x, attr_3 >, LessThanCondition)
\]

The possible combinations of conditions are $2^{NoOfApplicableConditions} - 1$ where for the above example $NoOfApplicableConditions = 4$.

To reduce the number of combinations to check we apply the following heuristic which is based on the *apriori* property. Informally, *apriori* property states that for a combination of $n$ conditions to be satisfied, any subset of those $n$ conditions must also be satisfied. To exploit this property, at first,
we check for single conditions and try to reduce the number of applicable conditions. For example, if $c_1$ is not satisfied, we do not need to check any combination that includes $c_1$.

Next, we check for conditions in increasing length. For example, assume conditions $c_2$, $c_3$ and $c_4$ are satisfied. We check which combinations of $(c_2, c_3)$, $(c_2, c_4)$, and $(c_3, c_4)$ are satisfied. If all of the length-2 combinations are satisfied, we check if $(c_2, c_3, c_4)$ is satisfied or not.

**Symbolic Pattern Ranking**

The discriminative pattern extraction algorithm often returns patterns with same or very similar support. In the case of non symbolic patterns, there is no clear way to decide which patterns should be ranked as the more important ones. Fortunately, in the case of symbolic patterns, we have a convenient way to rank the patterns. We applied a simple scheme where we give more importance to patterns that are more specific. To do that, we simply count the number of "*" in a symbolic pattern. The higher the number of "*" in a pattern, the lower the rank, as it is likely to be a self-evident generality that does not carry much information. The rationale behind this is due to the fact that "*" implies "don’t care" and hence patterns having more "*" are more likely to have higher support but represent a weaker concept. In contrast, patterns with fewer "*" give more information and should be ranked higher.

**3.2.4 Hybrid Support Count Function**

One of the inherent problems with any discriminative pattern extraction algorithm is that the number of patterns generated as discriminant patterns is overwhelmingly large, which can be in the order of thousands. It makes it “easy” to miss the “culprit” pattern which may end up deep down the list of discriminative patterns. As at each stage $i$, the candidate set is generated by concatenating the frequent patterns generated at stage $(i-1)$ with each of the unique events in the log file, for 100 unique events (e.g. the alphabet equivalent of English language) in a log file, the number of candidate patterns of length 3 is 10,00,000 and so on. To avoid losing crucial events, we have to
set the minimum support threshold to 1 (e.g., a single node reboot event may cause a large number of message losses and setting minimum support threshold larger than 1 will discard the “reboot” event). To address this challenge, in [6] we proposed a two stage approach which first identifies symptoms (e.g., message loss) with setting high minimum support threshold and later tries to identify the cause of failure (e.g., reboot) with lower support. Although this addresses the scalability issue to some extent, the patterns returned in this scheme still fail to return the “culprit” sequence at the top of the list if the cause of failure is infrequent (e.g., large number of message lost due to single node reboot).

The cause of the problem lies in the way support for an event is calculated in the frequent sequence mining algorithm in the data mining domain, which is ill-suited for debugging purposes. The reason is that if we have N log files and an event X exists in only one file 1000 times and does not happen in any of the other files, X will still be considered a frequent event with support 1000. But for debugging purpose, this is “wrong”. As the reasoning behind using discriminative pattern extraction for debugging relies on the assumption that an event correlated with failure should “exist” in at least a majority of the “Bad” log files. Event X in fact violates this assumption and is not a frequent event from a debugging perspective.

To address this problem, we have implemented a support count function that counts the frequency of patterns not only within a single log file but also across multiple log files and uses both estimates to generate support for single attribute events. For example, according to our scheme, if an event X “exists” in only one of the N log files, the across support for X is 1 irrespective of how many times it happened in that single file.

For example, though “reboot” event has a lower support in a single file, it has a higher support across the files (“reboot” exists in all the file for cases that crashed). Using this observation, we discard events from the base set that have across support lower than a threshold $\theta$ which is set by the user (i.e., $\theta = 0.6$ implies that for an event to be frequent, it has to “exist” in at least 60% of the files). We have two sets of frequent events (i.e., alphabet set), one for the set of good logs and one
for the set of bad logs which are subsequently used to generate longer patterns. This reduces the 
execution time significantly and helps rank the patterns that are more correlated with failure higher 
than the other “less” correlated patterns.

3.2.5 Collection of Logs

To use the tool, one must collect runtime logs from the application nodes. As long as the runtime 
logs follow the format specification required by the data analysis back-end, the source of the logs 
does not matter. Logs can be collected from simulation, emulation or from real hardware. For 
example, if the user intends to use TOSSIM, user can log any event inside the application using 
TOSSIM’s “dbg” statement as in \texttt{dbg(“Channel1”, “%d : %d : %d : %d....”, NodeId, EventId, attr1, attr2,....)} as 
described in [5]. The user can also use the data collection front-end designed for real hardware 
described in [6] to collect runtime logs from real deployment or choose to build his/her own data 
collection front-end.

3.3 Evaluation

To evaluate the diagnostic capability using discriminative symbolic pattern analysis, we used 
TOSSIM in TinyOS 2.0 to simulate the nesC code where we used a synthesized bug to create 
the sample log files. We simulated a network of 25 nodes placed on a grid topology with 5 rows 
and 5 columns. For the simulated bug, we compare the generated symbolic patterns with the non 
symbolic patterns generated by the algorithm presented in [6]. We choose to compare our result 
with [6] as [6] is the most related to our work that uses discriminative patterns to diagnose bugs.

3.3.1 Synthesized Bug

In this section we give examples of a synthesized bug to illustrate the strength of symbolic dis- 
criminative pattern extraction for debugging.
• Failure scenario-I: Out of order events, deterministic failure:

Let us assume that in a particular application, each neighbor of node A periodically communicates with node A and is always expected to send the messages of type 0, 1 and 2 in a fixed order where msgType 0 is followed by msgType 1 and msgType 2 respectively. Also assume that message reception in reverse order from a specific sender crashes the system. The discriminative pattern set returned by both algorithms from simulated logs for this failure scenario are given in Table 3.3. The first discriminative symbolic pattern captured the bug perfectly where it expressed the fact that if a particular receiver(X1) receives from a particular sender(X2) messages in reverse order of msgType where msgType 2 is followed by msgType 1 and msgType 0 respectively, there is a problem. Not surprisingly, the algorithm borrowed from [6] generated completely misleading patterns with highest support. Though it returned the pattern "(<msgReceived : (msgType : 2)>, <msgReceived : (msgType : 1)>, <msgReceived : (msgType : 0)>)" at the very end of the list, the crucial condition that this sequence causes a problem only if the messages are received from the same sender is not obvious from the reported pattern.

3.3.2 A Real Bug: Directed Diffusion Protocol Bug

We used the bug reported in [5] where a node experiences a large number of message losses after a node is rebooted in the directed diffusion protocol. For a detailed description of the bug, interested
readers are encouraged to read [5]. Briefly, in the directed diffusion protocol, each node maintains an interest cache entry to keep track of which way to forward a data packet. There can be multiple paths from a single data source to the destination node. If there is no interest cache entry that matches a received packet’s interest description, the receiver node silently discards the packet assuming it is not on the forwarding path. The problem is if a node gets rebooted for some reason, it wipes out the interest cache completely and causes a large number of consecutive message losses. The problem manifests only if there is a single path from the source node to the destination node. This bug is particularly interesting because in [5] the reported discriminative patterns showed the manifestation of the problem rather than showing that the “Reboot” event is the one that is actually causing the problem. For the log generated for this bug, the discriminative pattern set returned by the symbolic pattern extraction algorithm is given in Table 3.4. Symbolic patterns identified the real cause of the problem and correctly correlated the cause of failure and the manifestation. It

Table 3.4: Top Patterns for Directed Diffusion Protocol Bug

<table>
<thead>
<tr>
<th>Patterns reported in [5]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <code>&lt;interestCacheEmpty : NodeId : 3 &gt;</code>, <code>&lt;dataCacheEmpty : NodeId : 3 &gt;</code>, <code>&lt;dataMsgSent : TimeStamp : 20 &gt;</code></td>
</tr>
<tr>
<td>2. <code>&lt;interestCacheEmpty : NodeId : 3 &gt;</code>, <code>&lt;dataCacheEmpty : NodeId : 3 &gt;</code>, <code>&lt;dataMsgSent : NodeId : 4 &gt;</code></td>
</tr>
<tr>
<td>3. <code>&lt;interestCacheEmpty : NodeId : 3 &gt;</code>, <code>&lt;dataCacheEmpty : NodeId : 3 &gt;</code>, <code>&lt;dataMsgSent : msgType : 5</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Patterns generated by Symbolic Pattern Extraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <code>&lt;BOOT_EVENT : (NodeId : X1) &gt;</code>, <code>&lt;interestCacheEmpty : (NodeId : X1) &gt;</code>, <code>&lt;dataCacheEmpty : (NodeId : X1) &gt;</code></td>
</tr>
<tr>
<td>2. <code>&lt;BOOT_EVENT : (NodeId : X1) &gt;</code>, <code>&lt;msgDropped : (NodeId : X1), (ReasonToDrop : dataWithNoMatchingInterest), (TimeStamp : *) &gt;</code>, <code>&lt;interestCacheEmpty : (NodeId : X1) &gt;</code></td>
</tr>
<tr>
<td>3. <code>&lt;BOOT_EVENT : (NodeId : X1) &gt;</code>, <code>&lt;msgDropped : (NodeId : X1), (ReasonToDrop : dataWithNoMatchingInterest), (TimeStamp : *) &gt;</code>, <code>&lt;dataCacheEmpty : (NodeId : X1) &gt;</code></td>
</tr>
</tbody>
</table>
clearly shows that the “Boot” event is followed by the interest cache empty event and message is dropped due to no matching interest cache entry.

3.3.3 Performance Comparison

For the log files collected for the directed diffusion protocol bug, we used three good logs and three bad logs and analyzed using symbolic pattern extraction algorithm to generate patterns. Using symbolic pattern extraction, it took less than 1 hour and returned 188 symbolic patterns of length 3. In comparison, when we applied algorithm from [5], it took more than 3 hours and returned over several thousand patterns. This is due to the fact that in our approach, we were able to discard many unimportant events that had low support across multiple log files and thus reduced the number of the base events that were used to generate longer patterns. Please note that the progressive discriminative sequence mining algorithm described in Section 2.3.2 can also identify the “Boot” event as well. However, the progressive discriminative sequence mining algorithm would return it as a single discriminative event along with other discriminative events in this case as shown in Section 2.8.3
Chapter 4

Identifying “Vicious” Cycles

In this part of our thesis, we focused on a different kind of interactive complexity that often arises due to incompatible composition of adaptive components in distributed systems. Modern networked computing systems are characterized by large-scale, complex design, and high degree of interactive complexity across various system components. To cope with dynamic workload conditions and meet given performance goals, modern large-scale systems often heavily depend on autonomic management and control of various system resources such as CPU, memory, and energy. This part of my thesis addresses troubleshooting challenges that arise from this added complexity, when distributed systems perform poorly due to unintended interactions among system components.

Differently from performance problems caused by component failures or resource bottlenecks, we focus primarily on performance problems caused by self-reinforcing interactions between subsystems that lead to bad states. The cause of performance problems resulting from such bad interactions are often not easily diagnosed by previous debugging approaches geared for detecting single component failures, or those geared for isolating performance bottlenecks. In control theory, such self-reinforcing interactions are commonly known as “instability” or “oscillations” that often arise due to positive feedback loops. Since this thesis does not investigate a control-theoretic solution to the problem, we use the less technical but intuitive

---

1 Part of the work presented in this chapter was done in collaboration with Jin Heo and Shen Li at UIUC, and was published earlier. Jin Heo contributed the implementation of the run-time system based on middleware developed in his PhD thesis, whereas Maifi contributed back-end analysis based on data mining algorithms described in this thesis. Shen Li helped with the evaluation.
term (“vicious cycles”) to describe the problem. This effect is observed in software systems when component interactions produce sequences of events that lead to impaired performance while reinforcing themselves repeatedly, thereby exacerbating the problem even in the absence of failures and bottlenecks. Due to the cyclic nature of these interactions, gradual performance degradation ultimately leads to highly suboptimal behavior. As such problems generally arise due to lack of cooperation between different control actions in complex systems, in principle, it is possible to avoid such problems using architectural solutions such as a hierarchical control design that coordinates control actions. In practice, since systems are often composed from great many components, usually designed by different parties, it is hard to ensure that all components will correctly coordinate their performance-related actions. For such systems, incompatibilities may arise and will need troubleshooting, which motivates this part of our thesis.

Multiple examples of interaction-induced performance problems are reported in early work \cite{41}. For instance, in one case, two independent energy saving policies were identified to interfere with each other, leading to more energy consumption than either of the policies alone. The key in diagnosing the root causes of such performance problems is to identify cyclic event patterns that can potentially explain the problem. Techniques based on classical control theory can identify vicious cycles (unstable loops) when the variables involved are described by difference or differential equations, but when the cycles are composed in part of logical event sequences with no good models of the software systems that generate them, new different techniques are needed.

In this part of our thesis, we develop an online diagnostic service for performance troubleshooting in large-scale systems that tries to explain root causes of problems by identifying bad interaction patterns. It recognizes recurring anomalous sequences of events and links them to potential performance problems empirically, without the benefit of a priori system models. To build the diagnostic service, we leverage the online software service platform, OptiTuner \cite{50}. OptiTuner is developed for tuning and controlling performance of large scale distributed systems. Please note that OptiTuner software is not a contribution of this thesis. The diagnostic service leverage OptiTuner APIs to
log various system measurements periodically. When performance degradation occurs, the diagnosis module identifies “culprit” patterns that are highly correlated with the problem. Although the tool is designed for performance troubleshooting, it can also catch bugs that result in unintended performance problems.

In order to identify cyclic patterns, when the system performs poorly, the diagnosis module uses the progressive discriminative sequence mining algorithm to identify any anomalous sequences of events consistent with “vicious cycles” that may explain the cause of the performance problem. To further reduce the number of such identified candidates and increase accuracy, it focuses on patterns that include semantically conflicting events. Semantically conflicting events may arise due to conflict among different performance management mechanisms. For example, turning off machines by one policy while dropping requests by another policy to compensate for reduced number of machines are semantically conflicting actions.

Our approach offers two key advantages over previous contributions to the problem of identifying adverse interactions among system components [41, 44]. First, our new diagnostic service does not require detailed prior knowledge of the underlying system design to diagnose the problem which is required in one of the earlier effort [41]. Second, contrary to simple statistical techniques, such as correlation analysis used in the Adaptguard [44], our approach identifies semantically meaningful sequences of recurrent events along with continuous variables that may explain root causes of problems. To make the comparison of our scheme to prior work more concrete, we choose to reproduce two real-life problem scenarios reported in earlier literature. In one case, we successfully identified a bad pattern that explains the conflict between two independent energy saving policies, a dynamic voltage scaling (DVS) policy and a machine On/Off policy. Running both policies together end up consuming more energy than either one of the policies alone. In another case, the tool successfully attributes the cause of an anomalously low throughput to bad interactions between an admission controller and a dynamic voltage scaling (DVS) policy. We should highlight that prior techniques required knowledge of a system model in order to identify
these anomalies. We then present a new case study where we investigate failure of load-balancing in a server farm that correlated with cooling subsystem maintenance.

4.1 Overview

The diagnostic service works as follows. During normal system operation, the diagnosis module collects traces of runtime events (defined by the user) from event sensors, and labels them as “good” logs. When system performance is degraded, the module labels the trace as “bad” and performs diagnosis to identify any anomalous sequences of events that are consistent with vicious cycles causing the performance problem. Performance degradation itself can either be flagged manually by a user of the diagnostic tool, or can be automatically identified by specifying limits of acceptable performance (e.g., delay < 3 sec).

To effectively identify repeated sequences of events, we extend data mining techniques called discriminative pattern analysis, previously applied to diagnosing bugs in wireless sensor networks [6, 7]. As described in the earlier part of this thesis, data mining is especially appropriate for diagnosing root causes of non-reproducible behavior in complex systems with a dynamic and time varying nature, because the observed behavior diversity itself enhances ability to learn [6]. Further, discriminative pattern analysis is adequate for an online service as it reduces the search space tremendously and hence the processing time for diagnosis.

Identifying culprit patterns can be perceived as a classification problem. The goal is to identify discriminative patterns (i.e., sequences of events) that can correctly separate the bad logs from the good logs. Observe that it is usually enough to log only basic actions of software components or policy modules that directly affect the performance of the system, because the purpose of our service is to find out what actions cause performance to deviate from the desired goal. For instance, if the problem lies in excessive energy consumption, it is sufficient to log primary actions of policies
that directly affect power consumption (e.g., DVS increase/decrease and Machine On/Off operations).

The diagnosis module first generates frequent sequences of events in both good and bad logs. After frequent patterns are generated, a frequent pattern from the bad log is recognized as discriminative if the pattern is not found in the good log or has a disproportionately low frequency there. If a pattern is identified as discriminative, it joins a set of candidate patterns to consider as possible root causes of the problem. It remains to rank-order the patterns according to the likelihood that they are responsible for a problem.

Our algorithm reports both cyclic and non-cyclic patterns. Since a vicious cycle is necessarily a cyclic pattern, discriminative cyclic patterns are given a higher ranking when patterns are reported to the user. A cycle of repeating events \( A, B, \) and \( C \) can be thought of as a set of attractor states that once entered, repeats indefinitely. Entry can occur at any event in the cycle. Hence, repeated instances of \( ABC, BCA, \) or \( CAB \) are indicative of the same cycle. This equivalence is taken into account when counting frequent patterns. This reduces the number of reported patterns improving the usability of our service.

We further develop a simple heuristic to reduce the number of false positives by focusing on patterns that are semantically conflicting. This requires user help with coloring events such that conflicts are identified based on color. For example, if the user is trying to determine the cause of excess energy consumption, the user can annotate actions that increase consumption (such as “TurnMachineOn” and “FrequencyIncrease”) by the color red whereas those that decrease consumption (such as “TurnMachineOff” and “FrequencyDecrease”) by color green. Our tool can then make more informed decisions regarding the importance of the patterns. Namely, if a pattern consists of events of color Green only, they can be safely ignored. On the contrary, if a discriminative pattern (i.e., one that does not normally occur) consists of events of both colors, it may be one that reflects a conflict among policies that needs to be reported. (Normal upwards and downwards adjustment of controls around a set point will also generate a mixed-color pattern, but it will not be
discriminative since it occurs in normal operation as well.) Using this simple coloring scheme, our diagnostic service is able to prune uninteresting patterns and retain only those that are potentially revealing.

This approach can be extended to more than two colors as follows. The user may specify an \( N \) by \( N \) conflict matrix for \( N \) unique colors, where the \( (i,j) \)th entry in the matrix represents the conflict value between color \( i \) and \( j \). The rank of the patterns can be determined using the following formula:

\[
\text{ranking} = \omega \cdot \text{conflict value} + (1 - \omega) \cdot \text{frequency}
\]  

where \( \omega \) is a user defined constant between 0 and 1. A large \( \omega \) would give more importance to the conflict value defined in the conflict matrix.

Note that, the conflict matrix just represents the subjective knowledge of the user, and is not expected to be exact. This is solely used for the purpose of pattern ranking. Events that are not colored can be thought of as being of a default color that has no conflicts with other colors. Please note that, in the absence of conflict matrix, our tool will still work and will return the patterns that are discriminative. However, using a conflict matrix may decrease the number of false positives.

We expect that in many cases, using two colors is sufficient. One color can be used to represent actions in a direction that should improve the performance metric of interest. Another can be used to represent actions in the opposite direction. Indeed, the evaluation section presents examples where only two colors are used.

Finally, for efficiency purposes, discriminative mining algorithms look for shorter patterns first, and stop as soon as discriminative patterns are found. Hence, if a performance problem occurs because of a cycle involving multiple events, none of which occurs during normal operation, then each individual event in the cycle will be considered discriminative and will be returned as its own individual candidate pattern. This is problematic because it loses order information between events. Hence, if returned event sequences are too short to tell the whole story, an extra stitching
step can be invoked by the user to try and order them. To “stitch” events, the system traverses the “bad” logs once and builds a transition matrix, $M$, between discriminative events (defined as those contained in discriminative patterns) in the log. In this matrix, $M(A, B)$ counts how many times discriminative event $A$ was directly followed by discriminative event $B$ in the log (ignoring all other non-discriminative events). To determine the stitched event pattern, for each discriminative event, $i$, one finds $j$ from the matrix, for which $M(i, j)$ is maximum. This is the event most likely to follow $i$. Frequent common sequences $i, j$ are then stitched whenever they share an event, resulting in a frequent discriminative event graph that may shed more light on the anomaly. For example, if $AB$ and $BC$ were found to be most frequent, then the stitched pattern is $ABC$.

We leverage our Dustminer architecture to integrate the data collection front-end specifically designed to collect system logs from server cluster. The data collection front-end is implemented on top of OptiTuner [50]. OptiTuner provides an abstraction called sensor for collecting system states. Sensors are instrumented in the target system to collect various system measurements such as CPU utilization, delay, and throughput. One instance of OptiTuner process is run on each machine and provides a way to communicate with each other. The diagnosis module is run on the back-end machine for diagnosis.

### 4.2 Evaluation

To illustrate the use of our tool and evaluate its effectiveness, we reproduced two performance bugs reported in earlier work [41, 44], then introduced one new troubleshooting case study. In each case, we were able to successfully identify the “vicious cycle” that was the cause of the problem. The purpose of reproducing examples from earlier work was primarily to illustrate the use of our tool, and to demonstrate (using simple examples) that it reaches the same conclusions regarding root causes of problems as those that were already known to us in advance. Although prior work that reported these anomalies could also find the underlying bugs, it required knowledge
of design of the individual adaptive resource management policies in one case [41], and required that all variables be numeric in another [44]. In contrast, our approach identified the cause of each problem with no a priori knowledge of system design, including when logical event labels were part of the culprit sequence. In addition to reproducing past bugs, a new case-study is included that demonstrates a more interesting use of our tool in a recently encountered situation not previously reported.

### 4.2.1 Case Study - I

In our first case study, we apply our tool to a previously reported problem [44], where performance of a QoS-adaptive Web server was shown to degrade due to an interaction between the admission controller and a DVS policy.

The admission controller was intended to maintain acceptable latency. It would probabilistically drop client requests when utilization was too high. The DVS policy would decide whether to increase or decrease the CPU frequency based on the current CPU utilization to save energy when

<table>
<thead>
<tr>
<th>Variable</th>
<th>Increase</th>
<th>Decrease</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drop Probability</td>
<td>Red</td>
<td>Green</td>
</tr>
<tr>
<td>CPU utilization</td>
<td>Red</td>
<td>Green</td>
</tr>
<tr>
<td>Frequency</td>
<td>Red</td>
<td>Green</td>
</tr>
<tr>
<td>Average Delay</td>
<td>Red</td>
<td>Green</td>
</tr>
<tr>
<td>Request Rate</td>
<td>Red</td>
<td>Green</td>
</tr>
</tbody>
</table>

Table 4.1: Logged Events for Case Study I

Table 4.2: Discriminative patterns due to conflicts between the DVS policy and the admission controller

<table>
<thead>
<tr>
<th>Top Patterns Reported Before Coloring is Applied</th>
<th>Color of the Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. (&lt;\text{DropProbability }= 0.8 - 1, \text{Change in Probability }= 0&gt;)</td>
<td>N/A</td>
</tr>
<tr>
<td>2. (&lt;\text{DropProbability }= 0.8 - 1, \text{RequestRate }= 200 - 250/\text{sec}&gt;)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reported Patterns After Coloring is Applied</th>
<th>Color of the Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. (&lt;\text{DropProbabilityIncreased}&gt;, &lt;\text{FrequencyDecreased}&gt;)</td>
<td>(Red, Green)</td>
</tr>
<tr>
<td>2. (&lt;\text{DropProbabilityDecreased}&gt;, &lt;\text{FrequencyIncreased}&gt;)</td>
<td>(Green, Red)</td>
</tr>
</tbody>
</table>
the system is underutilized. A conflict occurred where a decrease in frequency increased latency and utilization, causing the admission controller to drop more requests, which in turn led to reduced load and a further decrease in frequency, causing a self-reinforcing loop. We already knew the conflict. The question was whether the tool would find it. While the example is very simple, it illustrates how the tool would typically be used.

![Figure 4.1: Conflict Between the DVS Policy and the Admission Controller](image)

To reproduce the scenario, we used one machine (with 2.53 GHz CPU) to run the QoS-adaptive Web server. Another machine (with 3GHz CPU) generated HTTP requests to the Web server using `httperf`. Soon, a large number of denied requests was observed. Our tool was used to “diagnose” the problem. In other words, as far as our tool knew, it was asked to tell why so many requests were being denied. The input to the tool was a set of colored logs. We logged various system variables such as processor frequency, drop probability, CPU utilization, average delay per request, and average request rate that may directly or indirectly influence the power consumption or may help to understand the performance of the system. We also logged events that changed these variables (such as changing frequency of the CPU) and discretized continuous variables. We labeled the logged events as either “Green” or “Red” depending on whether they were desirable or not from the perspective of the system performance. For example, increasing the request drop probability is not desirable and hence is assigned the color Red. In contrast, decreasing CPU frequency is
desirable (it saves energy) and hence it is assigned the color Green. High latency (which may or may not result from reducing frequency) is not desirable and is assigned color Red. The logged variables and the associated colors for the case study are listed in Table 4.1.

Since the tool diagnoses bugs by comparing logs of “good” and “bad” behavior, we fed it logs of each case. In the log repository, we had two traces of good behavior cases. One trace corresponded to the log that was collected when the DVS policy was used alone during testing and performed as expected. The other trace corresponded to the log that was collected when the admission controller was used alone during testing and performed as expected. When the two were used together, the throughput was found to be much less than expected (i.e., requests were rejected even when the server operated below capacity). This case was labeled as the bad log.

The explanation of poor behavior is shown to the reader in Figure 4.1. As shown in figure, initially, when only the DVS policy was in action, the CPU frequency oscillated around 2GHz. However, when we started the admission controller, it immediately increased the drop probability to reduce delay. This eventually reduced the workload and decreased the CPU utilization. As the CPU utilization went down, the DVS policy assumed that the server was underutilized and correspondingly reduced the CPU frequency to save energy. At the lower frequency, the CPU utilization increased and the admission controller increased the drop probability again and so on. Ultimately, the drop probability saturated around 1, the CPU frequency was reduced to the lowest possible speed, and the system got stuck at that point.

**Diagnosis**

To diagnose the bug, we applied our tool to the good and bad logs, and it came up with the patterns shown in Table 4.2 as discriminative. Two sets of patterns are shown. First, we show the top ranked patterns before any coloring was applied. As can be seen from Table 4.2, they correctly identify that drop probability oscillates between 0.8 and 1.0 most of the time when the system performs poorly. This, however, is a symptom of the problem, not a diagnosis. Second, we show
patterns reported with coloring applied. Those are the ones found to be “potentially conflicting”.

The pattern \(<\text{DropProbabilityIncreased}>, <\text{FrequencyDecreased}>\) clearly identifies the conflict: while admission controller is dropping requests, the DVS policy is decreasing frequency thereby decreasing ability to serve future requests. Since this is a repeating pattern, it is implied that the decreased frequency must have been followed by further increases in drop probability, generating the vicious cycle. Hence, the tool explained why so many requests were dropped.

### 4.2.2 Case Study - II

In our second illustrative case study, we applied our tool to troubleshoot an excessive energy consumption problem in a 3-tier Web server farm, reported in previous literature [41]. All servers in the Web server farm were equipped with a DVS policy and a consolidation policy that allowed turning off machines when load was below peak farm capacity. The two interacted adversely, causing energy consumption to increase. The DVS policy, which ran at a higher period, would check CPU utilization and decrease the processor frequency whenever the utilization was lower than a

<table>
<thead>
<tr>
<th>Variable</th>
<th>Increase</th>
<th>Decrease</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machines On</td>
<td>Red</td>
<td>Green</td>
</tr>
<tr>
<td>CPU utilization</td>
<td>Red</td>
<td>Green</td>
</tr>
<tr>
<td>Frequency</td>
<td>Red</td>
<td>Green</td>
</tr>
<tr>
<td>Average Delay</td>
<td>Red</td>
<td>Green</td>
</tr>
<tr>
<td>Request Rate</td>
<td>Red</td>
<td>Green</td>
</tr>
</tbody>
</table>

Table 4.3: Logged Events for Case Study II

<table>
<thead>
<tr>
<th>Top Patterns Reported Before Coloring is Applied</th>
<th>Color of the Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. &lt;Utilization = 0.4 - 0.6, Number of Machine = 12 - 14&gt;</td>
<td>N/A</td>
</tr>
<tr>
<td>2. &lt;Number of Machine = 12 - 14, Average Freq = 949976 - 1266635&gt;</td>
<td>N/A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reported Patterns After Coloring is Applied</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. &lt;FrequencyDecreased&gt;, &lt;MachineTurnedOn&gt;</td>
<td>(Green,Red)</td>
</tr>
<tr>
<td>2. &lt;FrequencyIncreased&gt;, &lt;MachineTurnedOff&gt;</td>
<td>(Red,Green)</td>
</tr>
</tbody>
</table>

Table 4.4: Discriminative patterns due to conflict between the DVS policy and the machine On/Off policy

82
threshold. The consolidation policy, finding resulting utilization high, would turn on an extra machine and rebalance load, thus reducing utilization and causing the DVS policy to further decrease frequency. The vicious cycle continued resulting in more machines turned-on than needed, and hence in highly suboptimal energy consumption. Our tool was to determine why the energy consumption is so high. Again, in this simple example, we already knew the answer, but wanted to see if the tool would pick it up.

To reproduce the adverse interaction, we configured a typical 3-tier Web server farm in our testbed where the first tier received HTTP requests from the user, the second tier executed business logic, and the third tier provided persistent storage for the second tier. For this case study, 17 machines were used: each tier was given 5 machines and two machines were used as load balancers for the first and third tiers respectively. An extra machine with 3GHz CPU was used for generating HTTP requests. We used a well-known Web benchmark, TPC-W, to construct an Amazon-like 3-tier Web site. We enforced the two energy saving policies: the DVS policy and the machine On/Off policy (explained above).

We logged all events of possible relevance to energy, such as frequency, average number of machines in use, CPU utilization, average delay per request, and average request rate. We also logged events that changed values of the above variables such as machine On/Off events. The logged
variables and the associated colors are listed in Table 4.3. Although the goal of both of the policies was to optimize energy consumption, when both policies were used together, the energy savings were in some cases less than when any one of the policies was enforced individually as shown in Figure 4.2. As before, we fed the tool both “good” and “bad” behavior logs. In the repository, we had two traces of good cases. One trace corresponded to the log that was collected when the DVS policy was enforced alone during testing and performed as expected. The other trace corresponded to the log that was collected when the machine On/Off policy was enforced alone during testing and performed as expected. For a bad case, we logged what happened at high load, with both policies present, when the amount of energy consumed was higher than that of either policy alone.

**Diagnosis**

The aforementioned logs were analyzed by the tool. After analysis, the tool returned the patterns listed in Table 4.4 as discriminative. Here, the patterns returned without coloring explain part of the problem. Both patterns show that our system was using 12 to 14 machines (of a total of 14 machines). Moreover, pattern 2 indicates that the machines were operating at a low frequency (949976 to 1266635 whereas the maximum allowable frequency is 2533270).

Patterns returned with coloring applied were more explanatory. The pattern `<FrequencyDecreased>, <MachineTurnedOn>` clearly suggests a conflict as the DVS policy is trying to save energy by decreasing frequency while the machine On/Off policy consequently turns on additional machines. Since the pattern is understood to be cyclic, it implicitly suggests that turning more machines on is followed by further decreases in frequency, and additional machines on, forming a vicious cycle. The pattern explains the suboptimal energy consumption.

### 4.2.3 Case Study - III

In our third case study, we applied our tool to troubleshoot an interesting performance problem observed in our server farm. The symptom of the problem was that a load balancing algorithm
appeared to malfunction when maintenance was done on air conditioning (i.e., when room temperature temporarily increased). The load balancer normally distributed CPU intensive tasks across multiple worker machines for back-end processing. New tasks arrived every interval and were distributed in a bin-packing manner. In our testbed, 15 worker machines were used. Since tasks were CPU intensive, at every interval, \( k \), each machine, \( i \), reported its average CPU utilization, \( u_i[k] \), to the load balancer. The load balancer aimed at a target CPU utilization of 80%. Thus, in each interval, as new tasks arrived, they were distributed over the minimum number of machines that could execute them while maintaining the aforementioned average utilization. The load balancer did not know the execution time of each task. Instead, if machine \( i \) was assigned \( n_i[k] \) tasks in a given interval, \( k \), and reported that it was busy only \( u_i[k] \% \) of the time in that interval, next time the load balancer assumed a capacity \( C_i \) for that machine, computed from:

\[
C_i = n_i[k] \frac{0.8}{u_i[k]}
\]

The load balancer would then sort machines by capacity from largest to smallest and assign \( n_i[k+1] = C_i \) to them in that order\(^1\) then turn off machines where no tasks were assigned.

\[\text{Figure 4.3: Throughput of the System in Different Conditions}\]

\(^1\)The last machine to be assigned may get less than \( C_i \).
Table 4.5: Discriminative patterns due to conflict between the load balancer and the thermal management policy

During normal operation, all assigned machines had comparable throughput. In one instance, however, we observed that the throughput of the system was unusually low, despite normal input load. This case is shown in Figure 4.3 where it seemed like the throughput of the 15-machine farm almost came to a halt. The only other observable difference was a higher-than-usual temperature in the machine room.

**Diagnosis**

To investigate the problem, we compared logs of different variables before and after the anomaly was observed. Our testbed logged per-machine CPU utilization, disk utilization, number of machines in use, average throughput, average temperature across all machines, and maximum temperature across all machines for each second. These variables were actually measured multiple times per second then averaged every second. Given the “before” and “after” logs, our tool came up with the discriminative patterns shown in Table 4.5.

As the returned patterns were too short (all of length 1), we applied the optional *stitching* step to the events in the top 5 patterns, which resulted the 5 by 5 transition matrix shown in Table 4.6 (where events are identified by their numbers from Table 4.5 for brevity). The maximum number in each row is highlighted in bold as it identifies the most common successor for each event type. For example, event 2 in the matrix (row 2) is most often followed by event 3 (largest count in the row) and vice versa. Similarly, event 1 is most often followed by 4 and vice versa. Hence, from the 5 by 5 matrix, two stitched patterns with the highest frequency are extracted. These are (<2.
Table 4.6: Transition Matrix for Patterns in Table 4.5

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>16</td>
<td>21</td>
<td><strong>124</strong></td>
<td>84</td>
</tr>
<tr>
<td>2</td>
<td>40</td>
<td>0</td>
<td><strong>78</strong></td>
<td>28</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>41</td>
<td>75</td>
<td>0</td>
<td>28</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td><strong>38</strong></td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>27</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

SleepEvent>, <3. WakeUpEvent>) and (<1. MaximumTemperatureObserved: 60 - 65>, <4. MaximumTemperatureObserved: 65 - 70>). One is a cycle of sleep and wakeup events. The second is a cycle of oscillating maximum CPU temperature around 65°C. It was easy to verify that the sleep and wakeup cycle was due to a thermal management mechanism that attempted to keep temperature around an operating point. Indeed, in our testbed, such a mechanism was implemented to cap maximum CPU temperature at 65°C, putting machines that exceed that temperature to sleep until they go below that temperature.

![Figure 4.4: Machine Temperature During Normal Operation](image)

The sleep and wakeup cycle was particularly revealing. Sleep implies non-work-conserving scheduling. It was immediately clear that this is a problem, because Equation (4.2), that controls load balancing assumes that scheduling is work conserving (i.e., that a machine will not idle when there
Figure 4.5: Machine Temperature During the Anomaly

is work to do). This was the link between higher temperature and failure of load balancing.

Reflecting on Equation (4.2), it is clear that computing the amount of load that will increase utilization back to 80% does not work when low utilization is attributed to sleep due to emergency thermal management, as opposed to lack of work to do. The low utilization of the machine where emergency management kicks-in would thus cause the load balancer to assign more and more work to that machine in a futile attempt to increase its utilization to 80% until all load is redirected to that machine, causing it to get severely overloaded while the rest of the machines are turned off.

With that insight, the reconstruction of good and bad scenarios was simple. On regular days, when the cooling system in the room was working correctly, CPU temperature remained well below 65°C as shown in Figure 4.4. As a result, the emergency thermal management mechanism was not triggered. All was well. Unfortunately, when the cooling system was turned off or not set correctly, the overall temperature of the room increased, triggering emergency thermal management that caused overheating CPUs to enter a lower duty-cycle (sleep/wakeup) pattern. Once a CPU overheats, and emergency management kicks-in, the reported utilization $u[k]$ would become low for that overheating machine. The load balancer assumed that the machine was underutilized, assigning more tasks to that machine. Since the reported utilization remained low, the load balancer kept increasing that machine’s capacity estimate and load allocation multiplicatively, causing it to rise to the top of the
list (since allocation was done to highest capacity machines first). This machine absorbed more and more load from other machines, until eventually all other machines were turned off, while the overheating one entered severe overload. This is shown in Figure 4.5 where it is seen that one machine maintained a temperature around 65°C, whereas others stopped reporting temperature one after another (because they were turned off).

Figure 4.6 shows the utilization of the overheating machine. Observe that no matter how many tasks were assigned to it, the CPU utilization was always below 80% (around 60% on average), because of the low duty cycle. Figure 4.7 shows the change in the number of machines in use during the anomaly. By the end of the experiment, only one machine was left on. Figure 4.8 shows the typical load per machine. It can be seen that on a good day, the load balancer assigns tasks
Figure 4.8: Average Number of Tasks Per Machine

evenly across all machines, whereas, during the anomaly it eventually assigned all tasks to a single machine (300 was the maximum number of tasks assigned per interval in our experiment).

### 4.2.4 Performance

The troubleshooting delay is bounded by the runtime of the algorithm. Once the problem is detected, troubleshooting takes in the order of seconds in our case studies. For the first case study, we collected traces for approximately 25 minutes and one sample every 3 seconds. It took the discriminative pattern mining module less than 8 seconds to finish and to produce the patterns. For the second case study, we collected samples for approximately 15 minutes at the rate of 1 sample per second. The discriminative pattern mining module took 64 seconds to come up with the patterns. For the third case study, we collected logs for 15 minutes. The discriminative pattern mining took less than 2 seconds as the algorithm did not need to generate longer patterns. The run time is expected to vary with the length of the discriminative patterns and the number of the system components involved in the analysis.
Chapter 5

Troubleshooting “Lack of Interaction” by Diagnostic Powertracing

In the final part of our thesis, to complement our work on troubleshooting interactive complexity, we focus on troubleshooting occasional “lack of interaction” in the deployed sensor system\(^1\). The work described in this part of our thesis is motivated by the need to reduce the cost of troubleshooting remotely-deployed sensing systems. When remotely-deployed nodes become unresponsive, it is generally hard to determine what caused some node to become silent, without sending a person to the field. If the cost of such field trips is high, remote damage assessment becomes highly desirable to assess the need for intervention. For example, if the cause of the problem is energy depletion (in a solar-powered system), there may not be much that can be done about it until the energy source is restored (e.g., weather improves). On the other hand, if the cause is attributed to a transient error (e.g., a system crash), power-cycling the system remotely may fix the problem. If the cause is attributed to a hardware malfunction (e.g., a radio failure), the urgency of repair may depend on whether or not the failure has affected the ability of the application to sample and store data. If the application continues to sample and locally store data, then there may be no need for immediate intervention. In contrast, some failures may require urgent attention. For instance, it is urgent to intervene if there is evidence of water damage that may cascade to other nodes or devices. Another example, experienced by our co-authors on one occasion, was a node that entered a cycle of repeated reboots. The cycle ultimately led to a hardware failure. Early intervention could have saved the node. Our tele-diagnostic system provides strong clues as to what might be wrong with a node, making it possible to plan intervention accordingly.

\(^1\)Part of the work presented in this chapter was done in collaboration and was published earlier \([8]\).
This chapter introduces the tele-diagnostic powertracer, an in-situ troubleshooting tool that uses external power measurements to determine the internal health condition of an unresponsive host and the most likely cause of its failure. We used a low-cost power meter with low-bandwidth radio to report power measurements and findings, hence allowing remote (i.e., tele-) diagnosis. The tool was deployed and tested in a remote solar-powered sensing network for acoustic and visual environmental monitoring. We were able to successfully distinguish between several categories of failures that cause unresponsive behavior including energy depletion, antenna damage, radio disconnection, system crashes, and anomalous reboots. It was also able to determine the internal health conditions of an unresponsive node, such as the presence or absence of sensing and data storage activities (for each of multiple sensors). We explored the feasibility of building such a remote diagnostic tool from the standpoint of economy, scale and diagnostic accuracy. To the best of our knowledge, this is the first attempt that presents a remote diagnostic tool that uses power measurements to diagnose sensor system failures.

While the approach of exploiting power traces to diagnose problems using our tele-diagnostic system is applicable, in principle, to a wide array of sensing systems, we present it and evaluate its performance on a specific deployed platform, called SolarStore [51]. The platform is intended for high-bandwidth sensing applications such as structural, acoustic, or video monitoring. It bundles higher-end solar-powered sensor node hardware with data storage and communication services. It therefore serves as a good example of the types of high-end sensing systems that our powertracer is designed to help troubleshoot. We show that, by remotely analyzing low-bandwidth power traces collected by cheap wireless power meters attached to the deployed SolarStore nodes, it is possible to infer useful information about the nature of node failures when they occur, as well as recognize some coarse-grained application states.

We emphasize that the diagnostic system, described in this thesis, is intended to help a remote operator determine the status of deployed, unresponsive nodes. Nodes that remain responsive can, in general, use other solutions for health monitoring. For example, they can run a local diagnos-
tic routine and report its outcome periodically. Such solutions have been discussed at length in previous literature and hence are not a part of the contribution of the work presented in this thesis.

5.1 System Architecture

Unresponsive behavior can be caused by many reasons, such as router failure, broken antenna (e.g., due to bad weather), software crash, or energy depletion. Once a node become unresponsive, an external measurement tool is needed to collect additional information regarding the node state. Given that the nodes perform different tasks such as sensing, communication, computation, and disk access, the power required for different operations vary quite significantly. This key observation leads to the idea of using external power measurements as a side-channel to infer further information regarding node states. To make the diagnostic subsystem feasible, we had two main objectives. First, we built the power measurement subsystem as an independent subsystem which does not depend on the real system being monitored in anyway. This separation of design reduces the likelihood of correlated failure. This also avoids any change on the monitored system. Second, the diagnostic system cost, in terms of money and power requirement, was kept at minimal. We found that the hardware cost of around 3% was achievable for a single unit.

Figure 5.1 shows the system architecture of the tele-diagnostic powertracer. An external power meter is attached with each sensor node that samples current and voltage periodically, and sends to the base station wirelessly. In our test deployment, the base station has wired Internet access. The diagnosis is done at the remote monitoring station. In our testbed, both the external power meter and the monitored system is connected to the same battery, which is charged by solar panel during day time. As the power meter requires much less power than the sensor node, even when the sensor node shuts down due to power outage, the external power meter can still remain functional and reports the battery depletion. In our case, the power meter remains functional at 6.2V, whereas the minimum power required by the sensor node was 11V.

93
Figure 5.1: A power-based tele-diagnostic system as an “add-on” to a sensing system

Although adding external power meter with built-in wireless communication increases the number of component that may fail in the field, the simplicity of the external power meter makes such probability negligible. For example, residential power meters are expected to operate for 10 years without interruption.

5.1.1 Testbed

We choose to deploy the diagnostic powertracer in conjunction with SolarStore [51], which is developed in our group for monitoring outdoor environment. Please note that this thesis does not claim any contribution towards the development of SolarStore, which is described here briefly for the purpose of completeness. SolarStore is a sensor network service that currently supports acoustic and video recording of wildlife. Two applications are currently deployed on the testbed. The first application records sound of bird vocalizations, and the second one monitors bird eggs using infrared cameras for potential predators. SolarStore manages the collected data in the network, and uploads data to the base station opportunistically. SolarStore uses embedded PC-grade computer as the computing platform, which is equipped with Wi-Fi router for radio communication, and so-
Figure 5.2: (a) Calibrate the power consumption of the computing subsystems indoor for the initial training of our tele-diagnostic system, (b) Outside look of a node in the solar-powered sensor network testbed.

Solar panels for charging batteries. Each node has various sensors such as microphones and cameras for different applications. Figure 5.2(b) shows a sample node.

5.1.2 Design of the Power Meter

Although the thesis does not claim any intellectual contribution towards the design or building of the power meter, we describe it briefly here for completeness only. The meter has two circuit boards. The first one uses an Allegro ACS712 hall effect sensor to measure the current consumption, and an op-amp based difference amplifier to improve the precision of the measurement. The second circuit board is a Digi XBee radio that uses 802.15.4 protocol for communication. It uses 1mW transmit power. The meter requires about 71mA of current. This translates into 871mW power at 12.27V in our testbed. For our experiment, we sampled at 1 kHz from the ADC, and took
an average over 220 samples. This gave us an approximate sampling frequency of 4.5Hz. The cost of the power meter was about $60.

### 5.2 Power-based Diagnostics

This section presents an exploration of different algorithms for diagnosing different node failure states from recorded power traces. The goal is to understand the trade-offs between algorithm complexity and diagnostic accuracy.

#### 5.2.1 Problem Statement

Our goal is to determine whether low-frequency power traces can be used to (i) distinguish among a range of common failure causes and (ii) infer gross-level application state on unresponsive nodes. Typically, the common failure causes in remote deployments are known from past experience. For example, our initial experience with a solar-powered deployment suggests that the most common failure cause is energy depletion. Other causes of unresponsive behavior of nodes include software crashes and communication device failures. We have also encountered cases of infinite loops involving a node reboot, and short-circuit due to water damage (shown in Figure 5.3). We take the above cases as a proof-of-concept portfolio of failures that we purport to distinguish. In general, as more failures are observed during deployment, and their power traces recorded, they can be added to the portfolio. The tele-diagnostic system is trained to recognize failures from their recorded power traces. Common failures can be emulated in the lab prior to deployment for purposes of diagnostic subsystem training. As new failures are encountered after deployment, their traces are used to re-train the diagnostic subsystem to recognize them in the future. Observe that, in the architecture described in Section 5.1, the diagnostic algorithm is run on the operator’s machine, as opposed to in the field. Hence, retraining simply involves updating the classifier at the
operator’s desk using data received from the field. It does not entail a need to upload new software to remotely deployed nodes.

Some failure modes, such as system crashes, entail application failure. Others, such as radio failures, do not give information on application status. It is therefore desired that the diagnostic subsystem can tell, upon occurrence of such failures, which applications are still running (i.e., are able to save their sensed data to disk). We exploit the fact that sensor networks do not typically run a wide range of different applications concurrently. Remotely deployed networks often have very specific purposes. Hence, the application count is limited. This significantly simplifies the diagnostic task. Indeed, the techniques presented in this paper are not likely to scale to a large number of applications. However, in a sensor networks context, they may still be useful for the cases of dedicated deployments. For example, only two applications are running in our current deployment.

To test the accuracy of the diagnostic techniques, we therefore set, as a benchmark, the goal of distinguishing among the twelve failure states shown in Figure 5.4. These include router failures (radio device is out), antenna failures (radio device is on, but the antenna is damaged), operating system crashes, solar energy-depletion, short-circuits (presumably induced by water damage but emulated in our tests by shunting power inputs using a small resistor), and infinite loops involving a system reboot (since, unlike other infinite loops, these would interfere with both the application

Figure 5.3: One node drowned in the flood by heavy rains.
execution and operating system functions, causing the node to potentially become unresponsive.

The above failures were chosen because they had been observed in the field. Moreover, in cases that do not necessarily entail application failure (namely radio and antenna failures), it is desired to tell which of the installed applications is running. The two installed applications in our system are acoustic monitoring and camera surveillance. This leads to the diagnostic tree shown in Figure 5.4.

![Figure 5.4: Possible failure states in our system. App-I is the application responsible for sensing sound using microphone. App-II is the application responsible for recording images.](image)

When a failure occurs, diagnosing it within a few minutes is considered good enough. We thus use a classifier to determine the state of the system every $\tau$ minutes which we call the detection period. When invoked, the classifier uses a window of size $\delta$ samples, to determine the system state. The following subsections explore the space of possible power trace analysis algorithms from simplest to more complicated, that can be used for classification in order of increasing complexity, as well as hybrid schemes that avoid their individual limitations.

### 5.2.2 Static Power Consumption Features

In the simplest case, we characterize the power consumption pattern for a particular failure state by the parameters of the probability distribution of power in the sampled power time-series observed for this state. These parameters are the mean, $\mu_k$, and the standard deviation, $\sigma_k$, for the time series
of state $k$. In other words, rather than modeling how exactly the values in the time series change, we lump such changes into a feature vector $(\mu_k, \sigma_k)$ for each state $k$.

Assume the series of power consumption measurements for system state $k$ is given by the power samples $x_1, x_2, x_3, ..., x_N$. The mean, $\mu_k$, and standard deviation, $\sigma_k$, over a training window of $n$ samples in state $k$, are calculated as follows:

$$\mu_k = \frac{\sum_{i=1}^{n} x_i}{n} \quad (5.1)$$

$$\sigma_k = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \mu_k)^2}{n}} \quad (5.2)$$

After training, each state is represented using a $(\mu_k, \sigma_k)$ pair and stored for later reference. At run-time, when a node becomes unresponsive, our diagnostic system is launched. The power trace data recorded for the unresponsive node is inspected. The vector $(\mu, \sigma)$ is computed over a sliding time window (e.g., 30 minutes) for the observed data, and matched to the nearest known failure state by finding $k$ such that the Euclidean distance between feature vectors is minimized:

$$\min_k \sqrt{(\mu_k - \mu)^2 + (\sigma_k - \sigma)^2}.$$ The observed state is thus classified as failure state $k$.

To test the accuracy of the above scheme, we collected 80,000 samples at the rate of 4.5 samples per second for each of the system states shown in Figure 5.4. We used the first 40,000 samples to extract the static features and the remaining 40,000 samples to test the accuracy of the model. To investigate the effect of window size on accuracy, we use window sizes of 1, 5, 10, 15, 20, and 30 minutes, respectively. The diagnostic accuracy for different window sizes is given in Figure 5.5. Our experiments show that the improvement in classification accuracy with increased window size diminishes after a size of approximately 10 minutes. In the next section, we seek to remedy the above inaccuracies using more involved classifiers.
Figure 5.5: Effect of the window size on classification accuracy for static feature based classification (Data preprocessing used: Outlier filtering)

5.2.3 Capturing Power Consumption Dynamics

A disadvantage of static features is that they do not capture the dynamics of the sampled power time-series. Fortunately, analyzing dynamic time series data and identifying time-varying patterns are very mature research areas in the machine learning and data mining communities. Several techniques that vary in complexity, accuracy and efficiency can be borrowed from the literature [52, 53, 54, 55]. We explore the use of Markov Models.

Modeling

A popular method for capturing the dynamics of complex time-series is the Hidden Markov Model (HMM). The model determines system states and probabilities of state transitions that best describe a particular time-series. In this case, we use a simplified version of Markov Models, where the states are predetermined. To build a model for the power trace of a given failure scenario, we process the power trace corresponding to the failure scenario using the following three stages:
**Preprocessing:**

As the meter is somewhat noisy, we always perform an outlier filtering step. We collect data for each system state and subsequently calculate the mean ($\mu$) and standard deviation ($\sigma$) for that system state, then discard any value that is outside the range of $[\mu - 5 \times \sigma, \mu + 5 \times \sigma]$ as an outlier. Next, we can perform an optional step where we may perform smoothing or normalization to input data based on system configuration.

**Discretization:**

Power consumption produces continuous-valued time-series data, which are difficult to analyze as the possible values for continuous data are infinite. To address this issue, we discretize the power measurements, reducing the numeric values of power samples to a small finite number of symbols. For example, 0-1 Watts can be represented as “a”, 1-2 Watts as “b”, and so on. These symbols represent measured power consumption levels, henceforth called *power states*. The entire power trace is therefore converted to a string of symbols.

**Model construction:**

We build a state transition diagram that expresses which states are followed by which other states. For example, a substring “ab” in the power trace string represents a state transition from “a” to “b”. By observing how often “ab” occurs in the string, we can determine the probability of state transition $ab$. For instance, in string “aaabbcd”, there are total of 6 transitions (e.g., the first “a” is followed by the second “a”, second “a” is followed by the third “a”, third “a” is followed by the “b” and so on). Hence, the transition probability $p(aa)=2/6$ (i.e., there are two transitions from state “a” to “a”), and $p(cd)=1/6$. Any trace can be summarized by a two-dimensional probability matrix that states the probabilities of state transitions from any power state $i$ to any power state $j$ in the trace. The aforementioned state transition diagram is also known as a Markov Model. For each system state, we build a model that represents that state.
The models built above are subsequently used for classifying system states during runtime diagnosis. When a node becomes unresponsive, we collect $\delta$ power samples and build a transition probability matrix for the collected samples. Next, we calculate the probability that the observed sequence of samples is indeed generated by the Markov Model for system state $k$, using the transition probability matrix generated during the training stage for system state $k$. The state which has the highest probability of generating the observed sequence is returned as the classification result.

To test the accuracy of the above family of classifiers, we used the same training and testing data set that is used for the static feature based classification. We used this approach to answer the following questions regarding the classifier design.

- What is a sufficient number of model states to use?
- What is an acceptable sampling frequency of the power trace?
- What is the effect of the data window size used for diagnosis?
- What are the pros and cons of different data pre-processing techniques?
- What are the pros and cons of improved data discretization techniques?

These questions are addressed below. In the following, for brevity, we refer to the Markov Model as an HMM (although technically the states in our HMM are not “hidden”).

**Effect of HMM Size**

To see the effect of the number of HMM states on classifier accuracy, we varied the number of states as 5, 10, 15, 20, 30, 40, 50, 60 and 70 and tested the HMM with a window size of 30 minutes. The effect of the number of states on accuracy is given in Figure 5.6. For this experiment, we trained the HMM on raw data (after noise reduction). As we can see from Figure 5.6, the accuracy increases with number of states and becomes stable after the number of states reaches
50. More interestingly, the figure highlights the fact that increasing the number of HMM states far beyond that value is a “bad” idea as well, because accuracy starts to drop if the number of states becomes “too large”. This is because with too many states, the amount of data available in the used window might become insufficient to accurately determine all the state transition probability. In the rest of the paper, we use 50 states for HMMs unless we specify otherwise.

![Effect of Number of HMM States on Accuracy](image)

Figure 5.6: Effect of number of HMM states on classification accuracy (Window size=30 minutes, Data preprocessing used: Outlier filtering)

**Effect of Sampling Frequency**

Since reducing the sampling interval increases energy consumption, we evaluate the effect of accuracy of the HMM classifier with various sampling intervals. We train the HMM classifier at the sampling interval of 222 ms, 444 ms, 888 ms, 1776 ms, 3552 ms, 7104 ms, 14208 ms, 28416 ms, and 56832 ms respectively. The lower sampling intervals were obtained from the same data by down-sampling the original time series (i.e., selecting one every \( N \) original samples for \( N = 1, 2, 4, ..., 256 \)). We present the effect of the sampling interval on accuracy in Figure 5.7. As we can see, if the sampling interval is reduced to 444ms, accuracy starts to drop and after that point the accuracy decreases monotonically due to the loss of information on finer-grained dynamics.
Figure 5.7: Effect of sampling rate on the classification accuracy of HMM (Window size=30 minutes, Number of states=50, Data preprocessing used: Outlier filtering)

**Effect of Window Size**

To test the effect of window size on accuracy, we trained the HMM on the original data (after outliers are removed) with 50 states and tested its accuracy with window sizes of 1, 5, 10, 15, 20, and 30 minutes respectively. Regardless of window size, we considered all windows shifted by 1 minute intervals. We show the effect of varying window size on accuracy in Figure 5.8. In general, increasing window size helps increase the overall accuracy. The amount of improvement varies between different failure states.

We also show the confusion matrix to determine the probability of misclassification and illustrate which states are most likely to get confused. Table 5.1 gives the confusion matrix for a window size of 30 minutes. A cell (i,j) in the confusion matrix represents the probability that of system state $i$ (the row) will be classified as system state $j$ (the column). The performance of the HMM with a 30 minute window size is significantly better than the static feature-based classification scheme.

We have 100% accuracy for all the states except f3 and f4. f3 occasionally gets misclassified as f4 and vice versa. It is worth noting that these misclassifications do not affect the ability to recognize which component failed. However, they err in inferring which application is running.
Effect of Data Preprocessing

In this section, we consider techniques for data preprocessing that have the potential to eliminate extraneous information from the sampled power signal, allowing us to focus on essential features. For example, the status of a CPU fan (“on” or “off”) can affect power consumption by adding or subtracting a constant offset. An HMM trained with the fan on may lead to misclassifications if the fan is turned off. To address this problem, we resort to data normalization prior to discretization. We explore two alternatives for normalization; namely, (a) z-score based normalization, and (b) normalization based on relative difference. We describe each of these techniques below.
Normalization based on z-score:

To normalize the data using z-score, we use the following formula:

\[ x'_i = \frac{x_i - \mu_k}{\sigma_k} \]

where \( x_i \) is the raw data, \( \mu_k \) is the mean and \( \sigma_k \) is the standard deviation for the training data for a particular system state. Intuitively, the z-score represents the distance between the raw score and the population mean in units of the standard deviation. The z-score is negative when the raw score is below the mean, and positive when it is above. It is a very common technique for data normalization in data mining literature. In Figure 5.9 we present the impact of varying window size on accuracy of an HMM trained based on z-score data. It turns out that the accuracy of HMMs using z-score normalization is not encouraging and can not be used for diagnosis effectively.

Figure 5.9: Effect of window size on the classification accuracy of HMM(Number of states=50, Data preprocessing used: Outlier filtering, z-score normalization)
Normalization based on difference signal:

As an alternative, we normalize the data using a simpler scheme, that uses the difference signal obtained from the following formula:

\[ x'_i = x_i - x_{i-1} \]

where \( x_i \) is the raw data. Note that this scheme is similar to obtaining the derivative of the sampled signal. Hence, any constant bias (such as the power consumption of an irrelevant fan) is eliminated due to differentiation. In Figure 5.10, we present the impact of window size on the accuracy of the trained HMM. As we can see from Figure 5.10, the window size has a significant impact on HMM classifier accuracy. The accuracy is considerably less compared to HMM when no normalization is done.

The intuition behind such poor performance when normalization is used is that because the absolute power consumption level does play an important role in identifying what is running and what is not. Data normalization causes information loss.

![Accuracy of HMM Classifier Based on Difference Signal](image)

Figure 5.10: Effect of window size on the classification accuracy of HMM (Number of states=50, Data preprocessing used: Outlier filtering, difference between consecutive signal)
5.2.4 Discussion

To summarize, from our evaluation it is clear that the static feature-based classifiers exhibit poor performance. HMMs trained using original signal (after removing outliers) without any normalization schemes are better and give reasonable performance. One way to improve accuracy is to train a separate HMM for the states that are misclassified. For example, in our case we trained one HMM for all the states and another HMM with 300 states only for state f3 and f4. The idea is to first use the HMM with 50 states to classify the sample and if it is classified as any other state except f3 and f4, we know that it is correct. The secondary HMM is used only when any state is classified as f3 or f4 by the 50 state HMM. Using this two stage scheme we obtained an overall classification accuracy of 98.3% for f3 and 86% for f4. Another point to note is that even if we have less than 100% classification accuracy for a particular state, as we are doing remote diagnosis, we can afford to collect data for more than 30 minutes and try to identify the cause of the problem across multiple 30 minute windows and narrow down the cause of the problem. In the future, we plan to explore the idea of ensemble classifier approaches widely used in machine learning for classification \cite{56, 57} where classification accuracy is improved by combining a group of weaker ones, called subclassifiers.

5.2.5 On-site Evaluation

In the previous section, a near perfect classifier is presented that uses low-frequency power samples to identify several problems with sensor network nodes in our lab. Below, we describe the experience of using this classifier in an outdoor deployment.

Our experimental testbed bundles higher-end solar powered sensor node hardware with data storage and communication services. It comprises of 10 nodes, where each node is powered by one 12 Volt deep cycle battery (DEKA 8G31) with a capacity of 98 AH, charged by a set of two solar panels. Aiming at high-end data acquisition, we chose Asus EEE PCs for their powerful comput-
ing capabilities, reasonably large local storage and high power efficiency. In addition, each node is equipped with a Linksys WRT54GL router, which is configured to the ad-hoc mode to support high-bandwidth communication between nodes. To experiment with using power consumption measurements to diagnose node failures, we developed a compact remote power meter discussed in Section 5.1.

To evaluate our diagnostic system, we collected real data from the real deployment described in section 5.1. For testing purposes, we artificially induced problems such as operating system crashes, antenna and router failures. Our scheme was able to identify correctly which component failed. Since in a live system we have access to several 30 minute windows of meter data (each shifted a small interval with respect to the previous one), it is easy to take majority vote. Hence, while individual windows may have led to classification errors, taking majority vote over a sequence of windows compensated for these.

5.3 Limitations and Future Work

In principle, the approach used in this paper to troubleshoot unresponsive sensor nodes remotely based on power consumption characteristics can be applied to a wide range of systems. However, it is crucial to understand the assumptions made in our current work before it can be extended to other contexts.

We developed the tool presented in this paper specifically for high-end sensing systems such as SolarStore [51] that justify the cost of using an additional power meter. For low-end systems that use low-power devices, such as Tmote and MicaZ motes, we may need to develop power meters that can run at a lower voltage (our current meter needs a minimum of 6.2 Volt to operate reliably). For such low-power devices, noise may become a problem. Exploring efficient algorithms for identifying patterns in the presence of a low signal-to-noise ratio will be an important challenge to investigate.
The current system does not consider resource efficiency of upload of meter readings. We simply note that the upload requirements are low due to the low sampling frequency of the meter. Efficient and optimized mechanisms for uploading meter readings could present another avenue of future improvement. Different optimizations may depend on the availability of infrastructure, such as connectivity to wired backbone networks or wireless networks including 3G technologies and WiMax.

From an algorithmic perspective, one limitation of our current diagnostic analysis is that it has no notion of “unknown” state. Currently, it classifies any state as either a “normal” state or a “failed” state depending on its distance measure from the trained states. To fix this, one can use a threshold-based scheme, such that if the distance measure from all the known states is larger than a predefined value, it is classified as an “unknown” state. Exploring classification algorithms that admit explicit unknown states is a worthwhile avenue for future work.

Another algorithmic limitation of the current system is its general lack of scalability. As the number of applications increases, the number of possible system states grows exponentially making them increasingly more difficult to classify. Rather than learning to tell the difference among an exponentially growing number of states, future incarnations of our algorithm will need to find characteristic energy features of each application or failure state that are \textit{invariant} in that they do not change with the introduction of other concurrent applications or failures. Finding such invariant energy features is a non-trivial undertaking that would be of great interest to explore. A related topic is one of possibly adding “energy watermarks” to the execution of different applications in order to create such invariant features where they do not exist naturally.

In the current paper, the authors artificially restricted the system to one meter. Obviously, measuring the power used by different components of the system separately can significantly enrich the set of identifiable energy features, hence making classification more accurate and effective. If the price of meters is small enough to allow using more than one device, this can be an important direction for improving the current scheme.
Finally, it is worth mentioning that we currently assume that all application failure signatures are known to the diagnostic algorithm in advance by virtue of prior training. Hence, once our algorithm is trained for a specific system with specific applications, it may not be used to troubleshoot other systems without re-training. Designing an adaptive algorithm that can learn new system states dynamically is a good direction for future work. Of particular interest is to design predictive algorithms that anticipate failure signatures of a modified system given previously recorded signatures and a description of the nature of modification. This will remove the need, for example, to retrain the system upon every software upgrade.

In summary, the diagnostic powertracer presents an initial proof of concept that demonstrates how energy measurements can be indicative of the nature of failures. We hope that this initial investigation will set the stage for many future extensions and improvements that address the limitations outlined above. We should stress that the above extensions mostly increase the scope of applicability of the approach. In our own outdoors deployment, the approach has already proven adequate and valuable in identifying sources of failures in our system. The powertracer, as it currently stands, is good at diagnosing failures in static, high-end sensing systems, dedicated to a single or to a small number of applications.
Chapter 6

Related Work

We divide this chapter into three parts. The first part explores various existing debugging tools available for troubleshooting wireless sensor network applications. The second part tries to explore the available debugging techniques for general purpose distributed and other systems. Finally, we describe different formal verification techniques used to verify wireless sensor network protocols and their limitations.

6.1 Development and Troubleshooting Tools for Wireless Sensor Network

Most of the early troubleshooting tools for sensor networks was generally geared towards finding local bugs such as an incorrectly written line of code, an erroneous pointer reference, or an infinite loop. Marionette [58] and Clairvoyant [59] are examples of source debugging systems that allow the programmer to interact with the sensor network using breakpoints, watches, and line-by-line tracing. Source level debugger is more suitable to identify programming errors which are contained in a single node. It is difficult to find distributed bugs using source level debugger due to the fact that source level debugging interferes heavily with the normal operation of the code and may prevent the excitation of distributed bugs in the first place. It also involves manual checking of system states which is not scalable. SNMS [60] presents a sensor network measurement service that collects performance statistics such as packet loss and radio energy consumption. Testing-
based systems include laboratory testbeds such as Motelab [61], Kansei [62], and Emstar [63]. These systems are good at exposing manifestations of errors, but leave it to the programmer’s skill to guess the cause of the problem.

Extensive work is also done to develop simulation and emulation based systems. Examples of such such systems may include TOSSIM [31], DiSenS [64], S2DB [65], and Atemu [66]. Atemu provides XATDB which is a GUI based debugger that provides interface to debug code at line level. S2DB is a simulation based debugger that provides debugging abstractions at different levels such as the node level and network level. It provides the concept of parallel debugging where a developer can set breakpoints across multiple devices to access internal system state. However, this remains a manual process, and it is very hard to debug a large system manually for design bugs. Moreover the simulated environment prevents the system from exciting bugs which arise due to peculiar characteristics of real hardware, and deployment scenarios such as clock skew, radio irregularities, and sensing failures, to name a few.

Sympathy [67] presents an early step towards sensor network self-diagnosis. It specializes in attributing reduced communication throughput at a base-station to the failure of a node or link in the sensor network. Another example of automated diagnostic tools is SNTS [4] which analyzes passively logged radio communication messages using a classification algorithm [68] to identify states correlated with the occurrence of bugs. The diagnostic capability of SNTS [4] is constrained by its inability to identify event sequences that precipitate an interaction-related bug. The tool also does not offer an interface to the debugged system that allows logging internal events.

There are prior work that attempt to recreate past from collected data. A mobile actor (computation) based post-mortem debugging tool is presented in [69]. Instead of moving around debugging data from node to node, it moves around the computation. Global States(MEGS) [70] presents an algorithm to recreate the snap shot of (partial) global state using information collected during runtime from the participating nodes using a side channel. Subsequently, user can define and check assertions and identify inconsistencies such as routing cycle. Envirolog [71] presents a system for
sensor network applications that collect runtime traces for offline record and replay and parameter tuning. Envirolog stores module outputs (e.g., outputs of sensing modules) locally and later replays them to reproduce the original execution. While this can also help with debugging such protocols, no automated diagnosis support is offered.

In one of the more recent efforts, PDA [72] presents a passive assertion checking approach where the user can use several predefined commands to upload or store values of interest (e.g., variables). They describe several approaches for collecting traces such as packet sniffing network, logging, and wired testbed. The assertions can be specified over distributed node states using a declarative language. Although this approach can identify the assertion violations but may not reveal the cause of the violation. PAD [73] represents another passive troubleshooting framework for root cause analysis. It uses a probabilistic inference model for determining dependencies among multiple network elements. It generates the network topology based on partial information collected from different nodes using a packet marking scheme. It diagnoses problems in real time and categorizes faults in several categories such as physical damage, software crashes, network congestion, environmental interference and application flaws. But it is not geared to troubleshoot arbitrary protocol bugs.

FIND [74] describes a novel approach for faulty node detection in wireless sensor network. The algorithm used in FIND is based on the assumption that the monitored event fades in intensity with increasing distance from the source (e.g., sound, temperature). Based on this assumption, FIND tries to predict the most likely node sequence for an observed event, and compares it with the reported node sequences to identify the faulty node (if any). FIND attacks a different problem than ours. The goal is to identify the faulty node. But it may not tell the reason behind the faulty node’s anomalous behavior.

Some of the other recent work includes Sentomist [75], T-check [76], and KleeNet [77]. Sentomist effectively uses SVM (support vector machine) to identify the potential buggy execution interval. Once such intervals are ranked, manual inspection is used for fault identification. Sentomist can be
used as a data labeling tool for Dustminer. Sentomist highlighted the fact that, in most cases, only a small part of the execution trace needs to be analyzed to identify the bug. This is a great advantage for tools like Dustminer as this makes the scalability challenge more tractable. T-check aims to identify safety and liveness errors before deployment by using state model checking leveraging TOSSIM simulation framework. But T-check’s diagnostic capability is limited due to simulation environment. Certain kinds of bugs such as timing bug and concurrency bug cannot be identified by T-check. Unfortunately, corner case bugs, which often manifest on real hardware in physical deployment, are the hardest to troubleshoot. Kleent tries to uncover bugs by injecting non-deterministic failures while executing on symbolic inputs. Kleenet attempts to cover as much execution paths as possible. NodeMD is a runtime tool that tries to detect failures before it disables the node completely. However, NodeMD focuses on stack overflow, livelock, and deadlock. But in real life, many bugs cause performance problems which are quite different, and do not fit in any of these categories.

The concept of declarative trace points is presented for efficient collection of runtime logs. But it does not automate the process of debugging. From that perspective, data collection using declarative tracepoint can be thought of as a data collection front end. Other efficient approaches for diagnostic tracing are also being proposed recently. Dustminer can use these tools as data collection front-ends to collect runtime execution traces.

In one of our earlier effort, diagnostic simulation was presented for automated diagnosis of the problem by analyzing simulation output. Dustminer subsequently extended the diagnostic capability by implementing an actual system (as opposed to using simulation) and presenting a better log analysis algorithm that is able to uncover infrequent sequences that lead to failures. The idea of symbolic sequence mining tried to address the challenge of sequence mining based on absolute values. Symbolic sequence mining identified that mining for patterns based on absolute attribute values may not be able to identify certain bugs where the system fails because of a hidden relationship such as hop distance, neighborhood etc. Moreover, we showed that symbol-
izing patterns increases the support count and hence improves the chance of subtle patterns to be ranked higher.

In this thesis, we leverage discriminative pattern mining for automated diagnosis. However, discriminative pattern mining received a lot of attention from the researchers in the data mining community as well. Earlier work in data mining [82] showed that the upper bound for information gain of an event (i.e., the potential discriminative power of an event) increases monotonically with the support of that event. This implies that the events with lower support has less discriminative power in general. DDPMine algorithm [26] showed how the upper bound estimation [82] can be exploited during the frequent item set mining process for effective classification. One of the more recent work is NDPMine [83] that formulates the discriminative pattern mining problem as an optimization problem. NDPMine maps the given datasets to a high-dimensional space, and learns the hyperplane that can correctly classify the input space. Intuitively, both DDPMine and NDPMine search for a combination of features that collectively can classify the input space. In contrary, our algorithm identifies discriminative sequences of events (often more than one) where each discriminative sequence individually can distinguish between successful and failed executions.

6.2 Data Mining Tools and Techniques for Troubleshooting Other Systems

Machine learning techniques have previously been applied to failure diagnosis in other systems [84, 47, 85, 86]. In one of the early work [86], the author presented a concept called Delta Debugging, where the cause of failure is iteratively narrowed down to a set of variables and values. Delta debugging starts with a large set of variables, and removes irrelevant variables. At the end, it comes up with a set of variables and values in different parts of the program, which potentially reveals a cause and effect relationship. However, the number of variables and possible values in a large system can be daunting. Moreover, in many cases, the problem is caused due to a bad design, rather than
by a specific set of variables. In a more recent effort, Sober [85] presents an algorithm that identifies a program predicate as correlated to failure if the evaluation pattern of that predicate differs significantly between correct and incorrect executions. Another work [87] leverages Sober to cluster traces collected from multiple failed executions based on predicted fault locations, which can be subsequently used for debugging. One prior work [88] suggested an algorithm to identify the faulty program regions in case of noncrashing bugs by analyzing the software behavior graphs of correct and incorrect executions, leveraging combination of closed graph mining and SVM classification. Mining control flow abnormality is also suggested [89] for identifying potential logic error in software programs. Bayesian approach is also suggested [90] for debugging where trace form a single buggy execution is available. However, these prior approaches focus on a single program, whereas in our work we make no such assumptions. Rather, we try to identify distributed interaction patterns, potentially involving multiple software and/or hardware components. For example, our proposed Dustminer [6] can capture which functions/events happened more frequently during the execution which the above approach can not. Moreover, analyzing call graphs (function calls) has a fundamental limitation. In many cases, the same function can be called both when the system is working correctly and when failing. Merely logging function calls does not capture what happened inside the function (e.g., a lock was set/unset) when it was called. At different invocations of the same function at different times, different events can take place inside the function. In this respect, logging finer grained events instead of just function calls enhances debugging capability.

In addition to the aforementioned work, significant research effort is spent to develop tools and techniques to identify faulty component in distributed systems. For example, Pinpoint [46] uses a client request Id to track a client request as it travels through the system and later uses data clustering for correlation to faulty component. Pip [91] provides an infrastructure to express expected behavior of the system (e.g., timing behavior, resource consumption) in a declarative language. It also provides visualization and query tools to analyze the log. But expressing expectations can be difficult. D+S [92] lets user specify predicates to check distributed properties (e.g., no two ma-
chines should be the leader at the same time). Once the violation is detected, developer needs to study the log manually to identify what sequence of state changes or events leads to the problem. No automated analysis to identify the culprit sequence of events are proposed. In that respect, D’s is more like a data collection front end for our work. Another work [93] used decision tree to diagnose faults for large Internet sites such as eBay. To diagnose faults, the authors [93] logged client request paths through multi tiered systems and recorded system components and databases accessed by the request. Later they used decision tree to correlate system components (e.g., particular machine, database server) to failures. One of the recent work [94] describes a system where neighboring nodes exchange local information such as AP (access point) and throughput among themselves, and builds a shared information repository to diagnose network level failures such as DNS error and low throughput. But this approach is not suitable to diagnose protocol design bugs. In general, debugging large distributed systems is a challenging task, and attracted significant attentions from the research community. But the approaches proposed so far are limited to decision tree, Bayesian network analysis, protocol verification using formal methods, log analysis, visual inspection. But our approach is significantly different from the prior approaches. To the best of our knowledge, we are the first to use discriminative frequent sequence analysis technique to diagnose protocol design bugs.

6.3 Formal Verification Techniques for Wireless Sensor Network

Verification of embedded systems using formal methods and model checking is rich in terms of existing literature [95]. But only a few [96, 97, 98, 99] of them are related to sensor network domain and tried to verify component correctness or, conversely, identifying which properties are violated. One work [96] presented a component verification method using Interface Automata [100] language. It tries to verify the compatibility in component integration. Slede [97] tries to verify
the correctness of security protocols implemented in nesC language by extracting the model automatically. Another work [98] describes verification of a medium access control protocol using probabilistic model checking. Due to scalability issue, the analysis is restricted to only three-hop network, which is impractical for large systems. It also makes simplified assumption such as circular communication range which often does not hold in real deployment.

In general, formal methods and model checking does not scale well for large scale systems due to exponential search space. Moreover, the high level of concurrency is hard to verify a priori. Hence, even formally verified systems often fail in real deployment due to corner case conditions. In contrast, our tool automatically answers questions that help identify the cause of failure that manifests during run time.
Chapter 7

Conclusions

With ubiquitous adoption and scale, interactive software systems are becoming an integral part of our life. We are using services (e.g., amazon online shopping, air ticket reservation system) that heavily rely on large scale software systems, whether we realize that or not. A software bug can affect day-to-day lives of millions of users in a matter of seconds, and may cost millions of dollar. For example, an hour of downtime costs 6 million US dollars on average for a brokerage operation. This is in addition to other intangible losses such as loss of customer satisfaction, confidence, and company goodwill. Today, although the biggest portion of the IT budget is spent on software maintenance and administration (140 billion US dollar yearly), data centers are struggling to maintain the quality of service. Unfortunately, due to tight coupling of various system components, it is increasingly becoming harder to troubleshoot and manage such systems.

Analyzing and understanding complex, large scale distributed systems such as wireless sensor networks and server farms from an empirical perspective is the primary focus of this dissertation. In my dissertation research, I applied key insights from data mining literature to come up with solutions that may help identify the complex distributed interactions that cause the systems to fail or perform poorly. The ultimate goals of my research are, first, to provide a rich collection of tools and techniques for troubleshooting large scale systems, and second and more importantly, to provide a comprehensive understanding of the different failure modes and reliability challenges towards better and more reliable systems, and help bring a fundamental shift in the way we design, build, and manage large scale systems, with an emphasis on next-generation cyber-physical systems. Towards this end, we made the following key contributions in this dissertation.
7.1 Summary of Contributions

In the beginning, we were motivated by the problem of troubleshooting wireless sensor network applications, and our goal was to build a tool that can answer high level questions such as “Why is throughput low?”, “Why is the communication delay so high?”, or “Why does the localization service failed?”.

We identified that such problems are often caused by unintended interactions due to poor protocol design or missed corner cases. To address this challenge, we developed a gapped subsequence mining algorithm in Chapter 2. We identified key limitations of sequence mining algorithm, and addressed those limitations in Section 2.2. Subsequently, we addressed the challenge of scalability. We developed two alternative algorithms, namely, the two stage mining and the progressive discriminative analysis algorithm. The progressive discriminative analysis algorithm exhibits order of magnitude improvement in terms of scalability and diagnostic capability.

We observed that, interestingly, often multiple seemingly different event patterns lead to the same type of failure manifestation. Pattern mining based on absolute values fails to identify the underlying hidden relationship responsible for failure. In Chapter 3, we addressed this key limitation, and developed the symbolic sequence mining algorithm, which is more capable in generalizing observed patterns by replacing absolute values with abstract symbols.

In Chapter 4 of our thesis, we focused on troubleshooting performance problems in large scale systems such as data centers. In this part of our dissertation, we focused on a different kind of interactive complexity that often arises due to incompatible composition of adaptive components in distributed systems. Differently from performance problems caused by component failures or resource bottlenecks, we focus primarily on performance problems caused by self-reinforcing interactions between subsystems that lead to bad states [42, 41, 43, 44]. In control theory, such self-reinforcing interactions are commonly known as “instability” or “oscillations” that often arise due to positive feedback loops.
The key in diagnosing such performance problems is to identify “vicious cycles” formed by conflicting components that accelerate performance degradation over time. We extended our progressive discriminative sequence mining algorithm to build an effective diagnostic service to successfully recognizes cyclic event patterns that are causing the problem. To reduce false positives, we developed a heuristic to discard patterns that are not semantically conflicting using a simple coloring scheme. We also developed a pattern correlation scheme to identify cyclic patterns if the sequence mining algorithm reports multiple single discriminative events instead of chain of events. Finally, to complement our effort on troubleshooting interactive complexity bugs, we looked into troubleshooting the cause behind occasional “lack of interactions” as well; for example, remotely-deployed sensor nodes that become unresponsive. Troubleshooting the cause often requires field inspections, which can be costly and inconvenient. To address this challenge, we developed an in-situ tool [8] for troubleshooting unresponsive nodes. Based on different power signatures in different execution states (e.g., sensing, communication, disk access), the tool determines the internal health conditions of unresponsive hosts and the most likely cause behind the apparent lack of interactions (e.g., energy depletion, broken antenna, router failure, sensing failure). The tool was applied to diagnose causes of silence in remote sensing installations.

**Immediate Impact of Our Work**

We have identified serious design flaws in widely used systems and protocols such as corner case design bugs in the Envirotrack target tracking protocol [101] which has over 160 citations, kernel level race condition bug in the LiteOS operating system [32] which is downloaded and used by numerous universities around the world, protocol design bugs in the directed diffusion protocol [3] which has over 1400 citations, and corner case performance problems in a multi-channel MAC protocol [2] which has over 40 citations.

Our work was included as a book chapter [102] on “Knowledge Discovery from Sensor Data” in the Lecture Notes on Computer Science published by Springer in 2009. Our research findings are
published in top conferences such as ICDCS [9], Sensys [6], and IPSN [8]. Our work was covered in the prestigious International CONET Summer School in Italy in 2009, advanced classes at the University of Minnesota, UCLA and numerous other classes, graduate seminars, and research groups around the world. My work on debugging has more than 70 citations so far. My top cited paper Dustminer [6], published in 2008, has over 30 citations so far.

7.2 Future Work

My long-term goal is to continue my research efforts in an interdisciplinary fashion to develop new algorithms, tools, and techniques towards building more reliable and fault tolerant large scale distributed systems, with an emphasis on next-generation cyber-physical systems. To achieve this long term goal, I plan to purse the following broad research directions.

First, in the near future, billions of devices with built in sensors and actuators (e.g., Internet enabled televisions to microwaves, interactive multi-party gaming consoles to smart meters) are going to be connected to the Internet, and access various services leveraging the cloud platform. Given that most of the software and hardware are increasingly becoming adaptive to changing execution conditions, both physical and computational, we are heading towards building the largest cyber-physical system ever. Distributed interactions are not going to be contained within a single isolated network, single application, or a standalone data center. Interactions will cross all possible boundaries - both physical and logical. In such settings, the problems of interactive complexity can manifest either in edge devices and/or in the cloud. Although, conceptually, the philosophy behind our work is applicable in such settings, effort is needed to apply it in practice. The current tool relies on a centralized approach where all the logs from different devices are consolidated into one, big sequential log. Unfortunately, given the scale of future systems, this is not a viable option. Pinpointing the cause of failure by mining gigabytes of runtime logs, which are often distributed across multiple networks and devices, is a significant research challenge. Instead of centralized
algorithms, we need a distributed approach for efficient fault diagnosis. Effective solutions will need clever techniques to zoom in on the right subset of components to analyze irrespective of the point of manifestation, and integrate runtime logs from heterogeneous sources optimally instead of pulling all the logs to a central location.

Second, we need scalable mechanisms to decouple the concurrency at different levels of executions for effective identification of causal relationships. Interestingly, there are techniques to extract programming models from the source code automatically. In principle, if the program structure is known apriori, it may be exploited to enhance the scalability of the data mining algorithms by pruning the exponential search space. As many programs often execute concurrently on a single node and interfere with each other, it is not obvious how multiple program structures and other deployment specific information can be leveraged during troubleshooting. It calls for active collaboration between the statistical debugging and the formal methods research community, which I plan to pursue in future. Moreover, this thesis does not provide any guideline regarding what to log in the face of anomalous behavior. Currently, it is upto the developer to decide what events to log. I plan to investigate this significant challenge in future. The ultimate goal is to develop efficient techniques for troubleshooting large systems in a distributed manner, and to fix the problem by taking appropriate actions transparently (e.g., installation of software patch), preferably without any assistance from the user of the device.

Third, the cloud computing paradigm is becoming the dominant platform for providing the necessary abstractions for efficient resource sharing across billions of edge devices, and will carry the major burden of managing the underlying computing infrastructure. Unfortunately, end-users’ lives are going to be easier at the expense of added complexity to the task of system administration and maintenance. For large systems, misconfiguration is a significant source of performance problems, and costs valuable resources. Moreover, misconfiguration can cause applications to crash or perform poorly, which often misleads the troubleshooting effort. Although there are some prior efforts to address such challenges, no general solution exists. I plan to undertake this research challenge,
and come up with systematic and general solutions for (i) automated management of system configurations, and (ii) troubleshooting of misconfigured systems, thereby contributing to the overall reliability.

Finally, troubleshooting large scale systems often requires collaboration among multiple independent parties, crossing company and administrative boundaries. Unfortunately, sharing information (i.e., system logs) across administrative boundaries, which is currently done mostly manually, has potential business risks and legal implications. On the other hand, without the complete end-to-end picture (captured by the system logs), it may be impossible to diagnose the problem. We need automated ways for sharing system logs across administrative boundaries in a privacy preserving manner. At the same time, the delicate balance between the privacy and the utility of the information being shared needs to be maintained. We need to develop sound analytical models and frameworks to evaluate various trade-offs on the fly. Without solving this piece of the puzzle, we may never have a completely automated solution to the problem of troubleshooting large scale systems.
References


[26] H. Cheng, X. Yan, J. Han, and P. S. Yu, “Direct discriminative pattern mining for effective classification,” in *Proc. of Int. Conf. on Data Engineering (ICDE ’08)*, 2008.


[83] H. S. Kim, S. Kim, T. Weninger, J. Han, and T. Abdelzaher, “Ndpmine: Efficiently mining discriminative numerical features for pattern-based classification,” in Proceedings of 2010 European Conf. on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECMLPKDD’10), Barcelona, Spain, September 2010.


[89] C. Liu, X. Yan, and J. Han, “Mining control flow abnormality for logic error isolation,” in Proceedings of 2006 SIAM International Conference on Data Mining (SDM’06), Bethesda, MD, April 2006.


Author’s Biography

Mohammad Maifi Hasan Khan was born in 1978 in Dhaka, Bangladesh. He earned his B.Sc. Engineering degree in Computer Science and Engineering in 2002 from Bangladesh University of Engineering and Technology (BUET). He joined the department of Computer Science and Engineering at Bangladesh University of Engineering and Technology (BUET) as a lecturer in 2002. He was a lecturer there until August, 2004, and taught numerous undergraduate level courses during his tenure. Later, he came to United States of America to pursue the Ph.D. degree in Computer Science in 2004, and joined the department of Computer Science at University of Illinois, Urbana-Champaign (UIUC). He earned his M.S. in Computer Science from University of Illinois, Urbana-Champaign in 2007, and his PhD in Computer Science in 2011.