Fault Diagnosis in Enterprise Software Systems Using Discrete Monitoring Data

by
Thomas Reidemeister

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

Waterloo, Ontario, Canada, 2012

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

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

Abstract

Success for many businesses depends on their information software systems. Keeping these systems operational is critical, as failure in these systems is costly. Such systems are in many cases sophisticated, distributed and dynamically composed.

To ensure high availability and correct operation, it is essential that failures be detected promptly, their causes diagnosed and remedial actions taken. Although automated recovery approaches exists for specific problem domains, the problem-resolution process is in many cases manual and painstaking. Computer support personnel put a great deal of effort into resolving the reported failures. The growing size and complexity of these systems creates the need to automate this process.

The primary focus of our research is on automated fault diagnosis and recovery using discrete monitoring data such as log files and notifications. Our goal is to quickly pinpoint the root-cause of a failure. Our contributions are:

- Modelling discrete monitoring data for automated analysis,
- Automatically leveraging common symptoms of failures from historic monitoring data using such models to pinpoint faults, and
- Providing a model for decision-making under uncertainty such that appropriate recovery actions are chosen.

Failures in such systems are caused by software defects, human error, hardware failures, environmental conditions and malicious behaviour. Our primary focus in this thesis is on software defects and misconfiguration.

Acknowledgements

About six years ago I was still an undergraduate student in Germany. After getting introduced to applied research during an internship in the University of Waterloo I decided to pursue a PhD in order to sharpen my analytic skills, enhance my critical thinking, and contribute to the academic community. In retrospect I am pleased to conclude that these past five years were an awesome journey of personal growth and research ventures that went well beyond just sharpening my analytic skills. Stepping out of the comfort zone of my native language, country, culture, and established methodology enabled me to write this work.

I would like to thank everyone who contributed to it. I would like to thank my supervisor Paul Ward, the members of my committee, Krzysztof Czarnecki, Jim Cordy, Patrick Lam, David Taylor, Lin Tan, and colleagues from IBM for direction, dedication, and advice. I would like to thank the members of the Shoshin Research group, especially Mohammad Munawar and Sukanta Pramanik, for their support and sharing their thoughts.

I would like to acknowledge the IBM Centre of Advanced Studies (CAS) for supporting my research financially through a four-year PhD fellowship. My appreciation also goes to the IBM staff and to the other CAS students for providing countless hours of discussion. I had great pleasure working with such talented people.

I would also like to thank the members of the Hogosha Martial Arts and the Waterloo Kendo club for providing inspiring examples of determination, effort, and commitment; and for creating a positive atmosphere that lifted my spirit in any stage of this process. I would like to thank my friends Akie Matsuyama, Jun Carol Fung, Hans Feige, Diane Inouye, Nam Leung, Ivan Popivanov, Karim Elrayes, Xiong Gang, and Chen Qi for their friendship, support and making my life in Canada enjoyable.

Last but not least, I would like to thank members of my family, my parents Bernd and Anneliese, my sister Eva, Martin Reidemeister, Robert Reidemeister, Hartmut Ziemann, Hans Ziemann, and my grandparents, for continued support that helped me throughout this journey. My gratitude also goes to the countless other people who helped me directly or indirectly at the University of Waterloo to make this work possible.

Contents

List of Tables

List of Figures

Nomenclature

γ	Describes the discounting factor of a Partially Observable Markov Decision Process
$\mathbb{A}$	Describes the set of actions of a Partially Observable Markov Decision Process
$\mathbb{C}$	Describes a set of class labels
$\mathbb{D}$	Describes the set of all documents
$\mathbb{F}$	Describes the set of fault states of a system
$\mathbb{O}$	Describes the a set of observations or symptoms
$\mathbb{S}$	Describes the set of states of a Partially Observable Markov Decision Process
$\mathbb{T}$	Describes the set of timestamps
$\mathbb{V}$	Describes a set of features
$\mathbb{W}$	Describes the set of words of a document collection
$\mathbf{\Omega}(\cdot\|\cdot,\cdot)$	Describes the observation probabilities of a Partially Observable Markov Decision Process
$\mathbf{b}$	Describes the belief state of a Partially Observable Markov Decision Process as vector of individual state probabilities
$\mathbf{B}(\cdot)$	Converts a document into a bit-vector. Each bit corresponds to the containment of a particular word.

$\mathbf{b}(\cdot)$ Describes the belief probability for being in a particular state of a Partially Observable Markov Decision Process

$\mathbf{D}$ Describes an instance of a document

$\mathbf{L}(\cdot,\cdot)$ Computes the Levenshtein distance of two sequences

$\mathbf{L}_n(\cdot,\cdot)$ Computes the normalized Levenshtein distance of two sequences

$\mathbf{R}(\cdot,\cdot)$ Describes the reward function of a Partially Observable Markov Decision Process

$\mathbf{T}(\cdot|\cdot,\cdot)$ Describes the transition probabilities of a Partially Observable Markov Decision Process

$\mathbf{V}(\cdot)$ Describes the value function of a Partially Observable Markov Decision Process that assigns a value to belief states

$\#(\cdot)$ Describes the number of instances having a particular class label

FM$(\cdot)$ Describes the f-measure of a class label

FN$(\cdot)$ Describes the false-negative rate of a class label

FP$(\cdot)$ Describes the false-positive rate of a class label

Prec$(\cdot)$ Describes the precision of a class label

Pr$(\cdot)$ Describes the probability of an event

Rec$(\cdot)$ Describes the recall of a class label

TN$(\cdot)$ Describes the true-negative rate of a class label

TP$(\cdot)$ Describes the true-positive rate of a class label

$|A|$ Describes the number of elements within the set or sequence A

$|S|_a$ Describes the number of times the element a is repeated within the sequence S

A	Describes a set of characters or tokens
$a_\emptyset$	Describes the do-nothing action of a recovery controller
$f_\emptyset$	Describes the fault-free state of a recovery controller
$O(\cdot)$	Describes the limiting behaviour of a function
$o_\emptyset$	Describes the no-error observation of a recovery controller
S	Describe an instance of a sequence
s	Describes the sensitivity parameter of Vaarandi's algorithm
t	Threshold for the normalized Levenshtein distance of single-linkage clustering
w	Describes the window-size parameter of the symptom extraction

Chapter 1

Introduction and Motivation

On April 21st, 2011 several web services of Amazon were unavailable for hours in the midst of a busy week. Their infrastructure did provide facilities to prepare for a widespread outage. However, due to Amazon's relatively high service availability compared to other providers, many users opted out of additional high availability services and were not well prepared for the outage. Several companies that rely on Amazon's services were affected, because they could not provide service either [79]. According to Amazon, their backend system is complex and the failure was caused by a configuration error [6]. This is only one example of how information software systems deemed highly available can fail and contribute to losses of revenue.

The success of many businesses depends on their *information software systems* (ISSes). These systems have high availability requirements, and failure in these systems can lead to loss of revenue, customer dissatisfaction, and may even have legal consequences. Failures in such systems manifest themselves as error pages, unavailable systems, poor performance and unexpected behaviour. In 80% of the cases they are caused by operator error or software bugs [102].

Detecting errors and diagnosing their causes in such systems is usually done by monitoring the system. The goal of monitoring is to observe system changes that occur over time. To do so, most ISSes have different interfaces that can be enabled to obtain data

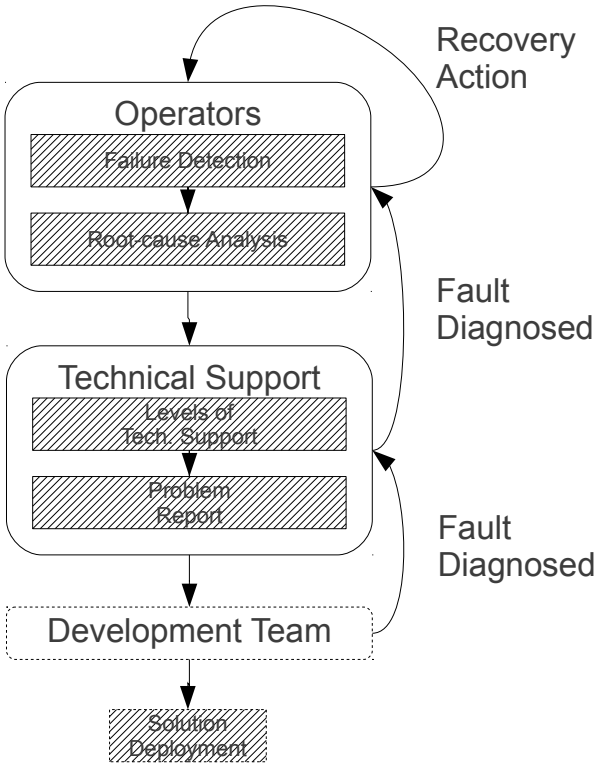


Figure 1.1: Workflow for diagnosing causes of failure

such as log files, application traces, and resource-usage metrics for that purpose [82].

In Figure 1.1 we show an example workflow of the steps through which failures are resolved in ISSes. Operators apply filters, best-practices, and hand-coded rules to monitoring data in an attempt to locate symptoms of failure. Upon failure verification, operators attempt to locate the fault that caused the failure. This process is currently painstaking, automation is limited and success is highly dependent on the operators' skills [98]. The recovery actions available to the operators are mostly limited to hard- and software reconfiguration, system restarts, and reimaging or reinstalling subsystems. This is because the source code of most legacy (i.e., existing) software systems is unavailable to operators,

and it is not their responsibility to debug the software themselves. If the cause of a failure could not be located or the type of fault does not allow a recovery by the operator, operators usually engage vendor support. Technical support has access to the source code and historic problem-management records. Many failures in ISSes are caused by recurrent faults, such that past instances of the same fault have caused failures of the same system or at a different customer site. Furthermore, the analysis process is in general manual as well. Upon locating the root-cause, technical support can recommend a recovery action to the operators (e.g., to install a fix-pack) or engage the development team (e.g., to provide a fix-pack for a validated software bug).

Due to the size and complexity of current ISSes, there is too much monitoring data for humans to manually evaluate and understand [71]. As these systems grow in complexity, size and dynamicity, the number of system administrators required, and the level of qualification for these people will increase. According to projections of the U.S. Bureau of Labour Statistics, the number of employed network and computer administrators will increase from 2006 to 2016 by 27%, from 309000 to 393000 and the number of employed computer-support specialists will grow by 13%, from 552000 to 624000 [138].

Most components of ISSes generate some form of discrete monitoring data, such as log files, for tracking the system history. These files provide valuable information for fault localization for operators and support staff, as shown by the following example.

1.1 Motivating Example

The operator: A system administrator receives an email from a user of the company's intranet website yesterday. The system administrator starts her investigation by visiting the website that provides an interface to the service. The service is still functioning incorrectly, as the corresponding website only loads once for every 10 requests. The system administrator obtains the rotation of the log file that corresponds to the time frame when the user experienced the failure. She uses `grep` [54] to look for known error codes, such as `400`, `401`, `403`, `500` and `503` that constitute to common manifestations of failure on web

servers. Upon discovery of a 503 log record that contains a cryptic error message, she enters it into a popular search engine to get more hints about faults and possible remedial actions. After reviewing a few posts of fellow system administrators, who investigated the same manifestation of failure in the past, she is convinced that the error originates from an application server that provides service to the web server. She resumes her investigation by looking at the transaction log file of the proprietary application server and discovers several Java stack traces in that time period. Upon entry into a search engine, she finds no results for the particular stack traces, except for a generic description for the error code that precedes the stack trace:

```
ERROR1234:  EJB * threw an unexpected exception, as follows *
```

The Enterprise Java Bean appears to map to a proprietary plug-in of the application-server vendor. Having no access to the proprietary source code, she cannot locate the root-cause or perform any remedial action. She finally decides to call the technical support of the application-server vendor, since the service is also an integral part of the company's internal accounting system.

Technical support: A customer support person gets called by the system administrator. At first, the system administrator is guided through a list of questions by the support person to validate that all possible remedial actions were performed and that the problem is actually a software defect on the vendor side. The support person opens a problem-management record (PMR), refers the administrator to actual technical support, and asks her to provide the log and configuration files of the system. A few hours later, the PMR is reviewed by a technical-support person. The support person quickly locates the stack traces that the system administrator found and searches the confidential PMR database for similar cases so that redundant work is avoided. After contacting some of her fellow developers, she is convinced that the discovered problem is new. She continues her investigation by setting up a system in a lab environment that is similar to the customer's system and confirms the failure. She documents each step in a problem-description report for development staff. Development staff react and provide a fix-pack for the problem within two days. The customer-support person provides the fix-pack to the customer and closes the PMR, appending a textual description of the root-cause and the remedial action. The system

administrator applies the fix-pack and the service now appears to run stably.

In the above example the original failure was actually detected by a user. Today, several tools are available that can aid in error detection and can be configured using expert knowledge to send emails when "severe error" records appear in log files. Unfortunately, because expert knowledge of the system is required, and the semantics of severity are in general unclear and defined by software developers [98], this feature is often not used, which makes rule-based analysis non-trivial. Furthermore, those rules may not sustain changes to the configuration of the monitored system, and thus may need to be constantly reviewed [130]. The reader is also referred to Quan [105] for additional scenarios of technical support workflows.

Since the software in use is possibly widely deployed, occurrence of the failure may not be limited to that incident. Since the support technician clearly documented symptoms and the final resolution (i.e., install the fix-pack), subsequent instances of that problem will benefit from the previous work. Research shows that many of the failures reported are actually caused by highly recurrent faults [102]. However, one of the major problems that is also shown in the above example is the lack of operational context in the monitoring data. The operator cannot identify the root-cause just by evaluating the log files. The exhibited exceptions could also be caused by other problems. Different software defects may have identical manifestations in the log data. We refer to this problem as perceptual aliasing.

In order to build an end-to-end process for fault diagnosis and recovery, these challenges need to be addressed. In this thesis we will provide a comprehensive model for automated processing of log files, even if the structure of the individual plain-text records is uncertain (*see* Chapter 5). Currently deriving rules for identifying symptoms of particular faults is a costly manual process. In Chapter 6 we describe our approach to automatically mine such rules from labelled monitoring samples (i.e., as they could be obtained from PMRs). To address the problem of perceptual aliasing, we have designed a model for decision making that integrates probes and recovery actions to validate a fault hypothesis and select appropriate recovery actions (*see* Chapter 7).

1.2 Scope of Work

Information software systems can offer several monitoring interfaces, including performance metrics, event logs, and interfaces to collect trace information. Our work exclusively focuses on diagnosing faults from discrete monitoring data such as system logs or logged event information. These monitoring interfaces are present across a variety of legacy ISSes and are commonly used by human investigators to diagnose faults. Because we focus on diagnosing faults, we assume the availability of reliable external error detection, such that our diagnostic approaches are applied to an ISSes in which a fault is highly suspected. We assume that side-effects of recovery are fully known and that system probes are designed in a way to have no significant impact on the system state.

Our focus is on diagnosing faults that require intervention. Such faults are permanent in the sense that intervention is needed to resolve the faults. Faults are not expected to resolve themselves over time. Further, the faults of interest in this dissertation are assumed to be recurrent, such that the same fault caused a failure in the past on the same or a similar system and that historic data of these occurences exists.

The reader should note that our aim is to make as few assumptions as possible about the structure and behaviour of the monitored system and the structure of the obtained monitoring data so that the proposed approaches are widely applicable.

1.3 Thesis Contributions

In this thesis, we make following contributions:

- A model for diagnosing recurrent faults from discrete monitoring data of ISSes,
- A comprehensive, low-cost approach to process log files for machine learning,
- A decision-making model to validate fault hypotheses and to recover from validated faults.

The state-of-the art research in log file analysis usually requires the availability of extensive expert knowledge, well-structured logging formats or the ability of special attributes

(i.e., correlators, consistent parameters etc.) to perform error detection and fault diagnosis. While this information may be available in some cases (e.g., Liang *et al.* [75, 76]), this work focuses on records of log files that contain unstructured time-stamped plain-text messages. Since the structure and content of log records are usually determined during the development phase of a software system error messages such as core dumps and stack traces are usually logged directly. This feature usually invalidates the assumption that all log records are structured. In addition the error information that was of interest during the development phase of the software system and is logged may not provide any useful clues to the operator as to the cause of the error. Our work adapts to the available structure of the log records. This thesis provides a solution that obviates the need for such expert knowledge. In particular, we do not require any explicit information about the structure of the log records, the presence of special attributes or the availability of expert knowledge of the system-development phase. However, we do require historic samples of faults in order to attribute their recurrence.

Furthermore, the operational context exposed in discrete monitoring data is very limited. Therefore, in many cases the information to be mined directly from the log file is not sufficient to accurately identify the fault that caused an error and then launch automated recovery actions. Also, existing research in self-recovery does not explicitly consider symptoms of particular faults to choose recovery actions. Instead, existing approaches use a cost-model to schedule a generic set of recovery actions to recover from errors. Such models work only for stateless software components. Furthermore, these models do not scale well when more recovery actions are added. Instead, we provide a decision model that takes into account the diagnosis of recurrent faults. Since this diagnosis may be uncertain and not accurately identify the cause of a failure, our decision model can incorporate probes to validate the alleged cause of a failure. Once a fault is accurately identified, our model allows scheduling of appropriate recovery actions to recover the system.

1.4 Thesis Organization

This thesis is organized as follows:

- **Chapter 2** introduces terms and definitions that are used frequently throughout this document. These terms focus on system organization, fault-tolerance techniques, text-mining and discrete decision-making processes.
- **Chapter 3** provides an overview of related work and summarizes the shortcomings of previous work.
- **Chapter 4** describes the problem of interest in this thesis and outlines our approach. We outline how input data is handled and how symptoms of faults are learned and subsequently diagnosed. Because this diagnosis is uncertain, we also provide an overview of a decision-making process that proactively increases the certainty of a diagnosis.
- **Chapter 5** describes our model of monitoring data that is used throughout the following chapters.
- **Chapter 6** describes the symptom extraction of recurrent faults and their subsequent diagnosis from discrete monitoring data.
- **Chapter 7** describes a proactive fault-diagnosis model that is based on the symptom diagnosis and a discrete decision-making process to increase the certainty of the diagnosis. This step should be seen as optional to the rest of the approach and is only applicable in special circumstances.
- **Chapter 8** shows our experimental evaluation, focusing on the model of monitoring data and learning symptoms of recurrent faults.
- **Chapter 9** concludes this thesis. We discuss our findings and outline future work in the area of fault diagnosis and diagnosis-based system recovery.

Chapter 2

Background

In this Chapter we describe background and related work of this thesis. This chapter is organized as follows. In Section 2.1 we introduce key terms of dependable computing. In Section 2.2 we outline causes of failure. In order to detect errors and diagnose faults in ISSes, they need to be monitored. We describe monitoring in Section 2.3. We discuss text-mining approaches that are integral to approach to model discrete monitoring data and symptoms of recurrent faults in Section 2.4. Finally, we describe the concept of the Partially Observable Markov Decision Process (POMDP) that we use for proactive diagnosis in Section 2.5.

2.1 Terms and Definitions

The following terms are used in the field of dependable computing. This taxonomy of terms comes from the definitions given by Avizienis *et al.* [10] and Salfner *et al.* [119].

- System: A *system* consists of subsystems that provide a common service. It usually interacts with its environment. The interface between the system and its environment is referred to as the *system boundary*.
- Subsystem: A *subsystem* is a system that is contained within a larger system.

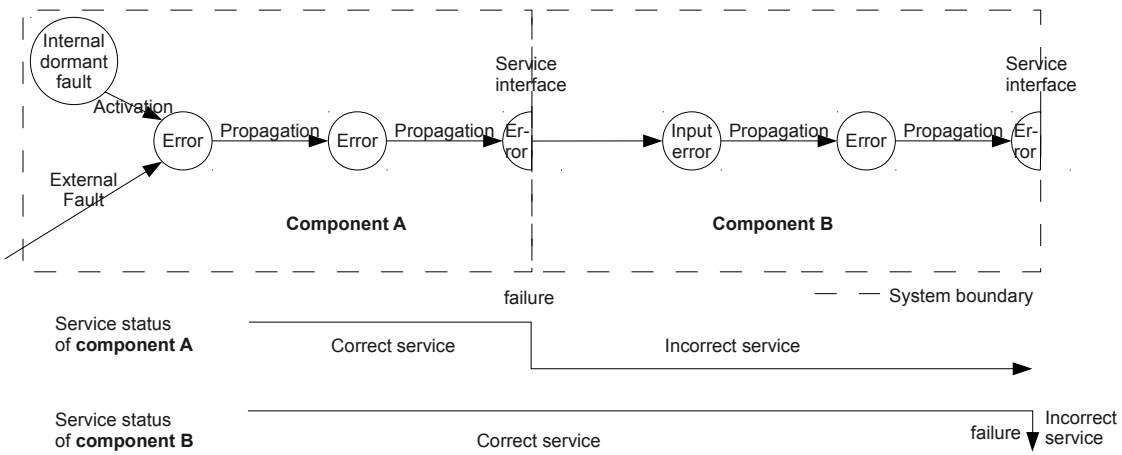


Figure 2.1: The relationship between failure, error, and fault

- Component: A *component* is a subsystem that is not broken down into smaller entities. Note that the granularity of the component definition is adjustable. We state the granularity of the component definition where required.
- Service: The *service* is the *behaviour* of the system that is perceived by its users.
- Users: Users are other systems that receive service from the *provider*. The interface to the user is called the *user interface*. A system provides *correct service* if it implements its prescribed function. Providing service is called *service delivery*.
- Dependability: *Dependability* is the ability of a system to avoid failures that occur beyond acceptability.

We focus our research on the threats to dependability, which encompass failures, errors and faults. The relationship between failure, error and fault is shown in Figure 2.1. The Figure was adapted from Avizienis.

- Failure: A *failure* is an event in which the system does not deliver correct service; that is, the system does not provide its prescribed functionality to its users.
- Error: An *error* is part of the total state of the system that may lead to a failure. Errors also include input errors. The process of errors causing subsequent errors is

referred to as *error propagation*.

- Fault: A *fault* is the alleged cause of an error. Faults can be *internal* or *external*. A fault is called *active* if it causes an error; otherwise it is referred to as *dormant*.
- Symptom: An external manifestation of failure is called a *symptom*.

From our perspective, the monitoring interface is not considered as the user interface. As such, we perform error detection not failure detection at the level of the monitoring data. Unexpected changes in the monitoring data identify errors that could likely indicate a component failure. When a component failure is validated, some of the monitoring records may become part of the symptom of that failure.

To illustrate some of the concepts introduced so far let us revisit the example from Section 1.1. The fault in the system is a software defect that needs to be fixed by the development team. The failure is the transition to incorrect service such that the user cannot use the service on the intranet. This incorrect service of the system is also validated by the system administrator. The error codes in the web-server log file as well as the stack traces in the application-server log file are symptoms of that failure.

According to Avizienis *et al.*, the four measures to reach dependability are as follows:

1. *Fault prevention* prevents the introduction and the occurrence of faults.
2. *Fault removal* encompasses the reduction of the number and severity of faults.
3. *Fault forecasting* covers estimating present number, future incidence and potential consequences of faults.
4. *Fault tolerance* avoids service failures in the presence of faults.

The first measure is used during the development phase of the system. The second measure can be applied during the development and the production phase of the system. During the production phase, fault removal is part of corrective or preventive maintenance. The third measure is accomplished by evaluating the system behaviour during the stages of its life-cycle by estimating fault occurrences and activations. Our research is situated in the area of the fourth measure. Figure 2.2 shows the *fault-tolerance techniques* proposed by Avizienis *et al.*

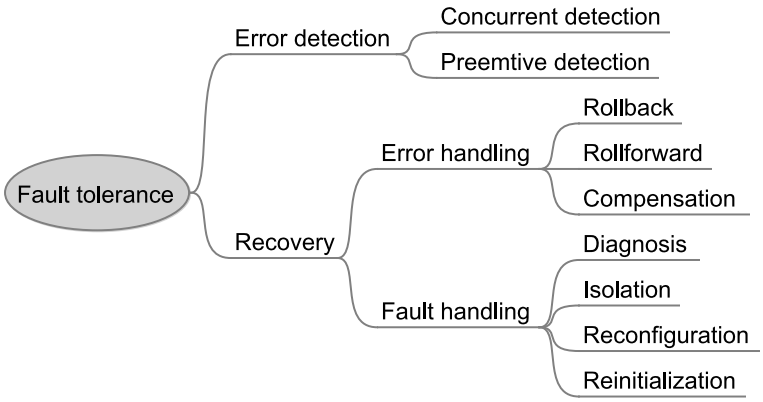


Figure 2.2: Fault-tolerance techniques

In order to achieve fault tolerance, the presence of an error needs to be identified first through *error detection*. This step is followed by *recovery*, which transforms the state containing errors into a state without detected errors.

Error detection can be divided into *concurrent* and *preemptive* detection. The former takes place during the normal service delivery and the latter takes place when the service delivery is suspended. *Error handling* eliminates errors from the system state; this is achieved through one of *roll back*, which brings the system back into a state saved prior to error occurrence; *roll forward*, which skips to a new state without detected errors; and *compensation*, which makes systematic use of redundancy in the system to mask errors.

Fault handling comprises diagnosis, isolation, reconfiguration, and reinitialization. *Diagnosis* identifies and records the cause of an error. *Isolation* excludes the faulty component from participation in service delivery. *Reconfiguration* switches in redundant components or distributes the tasks among non-failed components. *Reinitialization* checks, updates and records the new system configuration (i.e., without detected errors).

2.2 Causes of Failure in Information Software Systems

Amazon's example [6] has shown that even highly dependable systems can fail. In this section we describe possible causes of failures of ISSes. The major causes of failure of ISSes are scheduled maintenance, software bugs, hardware defects, environmental conditions, operator error and malicious behaviour [14]. The focus of this thesis is on detecting and diagnosing the causes of non-malicious software- (*see* Section 2.2.1) and misconfiguration-related failures (*see* Section 2.2.2).

2.2.1 Software-Related Problems

Various studies have shown that software may still contain a number of bugs after rigorous testing (e.g., Gibbs [37] and Oberg [94]). Bugs can be classified into two categories: Bohrbugs and Heisenbugs. *Bohrbugs* describe a phenomenon that can be easily reproduced and is likely input dependent. Their frequency usually decreases as software systems mature; however, they may be introduced during updates. *Heisenbugs* are usually caused by faults that were activated earlier in the execution, such as *buffer-overrun bugs*. For example, if a buffer is overrun and data is written in an adjacent data structure, it will not have an effect on the execution until the adjacent data structure is accessed. Heisenbugs are therefore sensitive to the order of execution. Furthermore, they can be input independent. A failure may occur or not occur even when the program is executed with the same input. Ironically, attempts to debug such failures may inject errors. New programming languages such as Java [41] and C# [56] that enforce strict type checking and application frameworks such as .Net [87], CORBA [95], and J2EE [133] that enforce several reliability constraints can reduce the number of bugs introduced during the software-development process. However, even these tools are not able to completely avoid software-reliability issues [14].

2.2.2 Configuration Errors

Many of the failures that are experienced in distributed ISSes are related to configuration mistakes, as exemplified by Amazon [6]. Due to the complexity of the system it is hard to maintain a model of how the system is configured at any point in time, what impact a configuration change has on the system, and what components are currently being changed; therefore, upgrading, extending, modifying or applying patches to the system is currently a painstaking task and is heavily dependent on operator skill; however, as shown by Pertet *et al.* [102], humans can err. Mistakes during the reconfiguration process can have catastrophic consequences.

2.2.3 Summary

In this section we discussed primary causes of failure in ISSes. The primary focus of this thesis is to diagnose causes of non-malicious recurrent failures in ISSes. Such failures usually stem from software-related problems that we discussed in Section 2.2.1 and configuration problems that we described in Section 2.2.2.

2.3 Monitoring Information Software Systems

The focus of our work is to analyze monitoring data obtained from information software systems for error detection and fault diagnosis. *Monitoring* is the dynamic extraction of information about processes and its presentation to users in a useful manner for debugging, testing, performance evaluation, and management activities: Quality of Service (QoS) management, configuration management, accounting management, and specifically fault management [67, 71].

We describe monitoring in the context of system management in Section 2.3.1. In Section 2.3.2 we describe the types and properties of common ISS-monitoring interfaces. In Section 2.3.3 we describe discrete monitoring data in detail as it is the primary focus of this thesis.

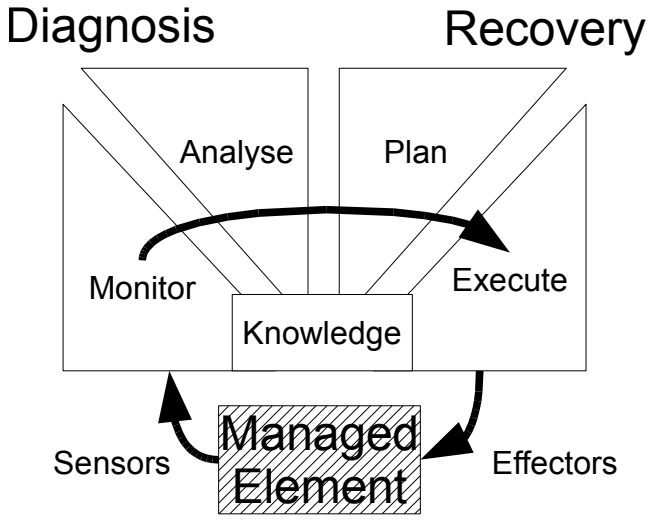


Figure 2.3: MAPE and its relation to fault diagnosis

2.3.1 System Management

A common paradigm used to describe system management of ISSes is the MAPE model [71] shown in Figure 2.3. *MAPE* stands for Monitor, Analyze, Plan and Execute. MAPE models the system as individual elements that are subsystems or components of the system.

Monitoring is a passive processes that is needed to obtain information about an element in order to make management decisions. The analysis encompasses extracting the relevant information, specifically for error detection and fault diagnosis. These steps result in the planning of decisions that are executed as control actions to the element. Such decisions involve testing and correcting faults. In the next section we describe the types monitoring interfaces that are exposed by ISSes.

2.3.2 Monitoring Interfaces

In practice many components of the system produce monitoring data. As ISSes are usually composed of many components and are distributed and dynamic, analyzing the data is non-trivial and creates the following challenges (*see* Joyce *et al.* [67] and Mansouri-Samani [82]):

- Concurrency,
- Communication delays between components of a distributed system,
- Non-deterministic, asynchronous behaviour,
- Dynamic changes,
- The probe effect, and
- Collecting, communicating, and storing monitoring data.

In addition to the technical challenges pointed out by related work, our experience shows that monitoring data lacks sufficient context to reason about the state of the system reliably. In particular the type, format and content of monitoring data are decided during the development phase of a software system mostly by software developers. In many instances the assumed operational context of monitoring records may differ from the actual context. In some cases the software developer takes no consideration about the operator and includes debugging information, such as stack traces and core dumps, in the monitoring records. While such information may be valuable for the developer, it provides little use to the operator, as she may not have access to the source code of the application or the means to debug and remove the fault.

Figure 2.4, adapted from Munawar *et al.* [90], shows an example of a system that exposes different monitoring-data formats. It comprises an e-commerce application that is running on top of a J2EE web server, stores its business data in a relational database system and provides a service interface using an Hypertext Transfer Protocol (HTTP) server (*see* RFC2616 [32]).

Many users will access the system at the same time. This means that the application will spawn multiple threads on the application server that process the users' requests concurrently; therefore, the monitoring data generated for each request is generated *concurrently*. The application server, web server, and database server may run on different physical machines in the network. The application may even be distributed across different application servers. *Communication delays* between those servers will also manifest themselves in the generated monitoring data. Application servers could be added to or removed from the system, which also leads to a *dynamic creation* of monitoring data. All monitoring interfaces have an impact on the performance of the system. The overhead of

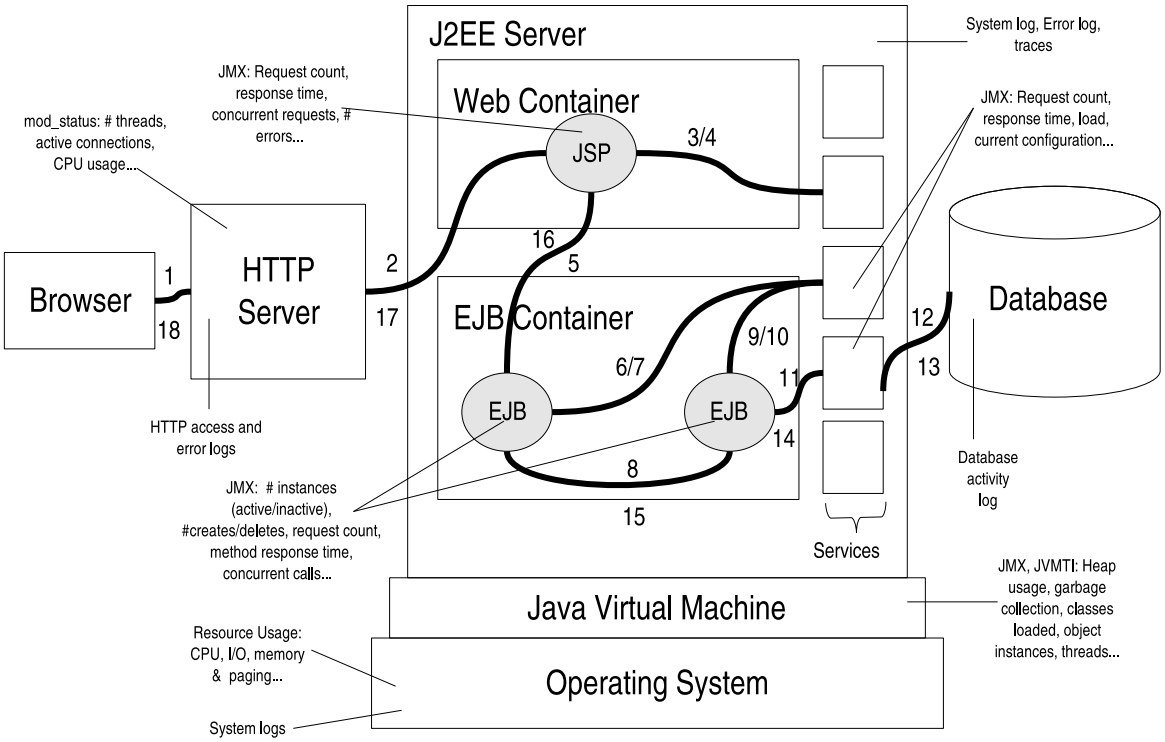


Figure 2.4: An example of different monitoring interfaces exposed by an information software system

monitoring is usually specified as part of the system's service. However, some monitoring interfaces, such as application traces, could cause a unexpected performance degradation that disturbs the system's service. This is referred to as the *probe effect*. The monitoring data of the individual components must be *collected, communicated, and stored* for analysis.

Each of the software components shown in Figure 2.4, offer different types of monitoring interfaces which in turn generate data of different natures (e.g., nominal or ordinal, *see* Stevens [131]). Also these interfaces provide different system coverage and varying level of detail. For example *application traces*, such as Application Response-time Measurement (ARM) [65], obtained from the web application provide detailed information to the method level, but only cover the methods of the individual application. *Resource usage metrics*, such as provided by the Windows Management Instrumentation (WMI) [88], provide a wide coverage of the operating system and the running processes, but only very coarse data about the processes themselves.

The different monitoring interfaces can be categorized into discrete or continuous monitoring data. *Discrete monitoring data* corresponds to monitoring data that is generated at discrete points in time, such as log records in log files and application traces. *Continuous monitoring data* can be requested at all points in time, such as resource usage metrics. Our focus is on analyzing unstructured and structured discrete monitoring data that, we describe in the next section.

2.3.3 Discrete Monitoring Data

Discrete monitoring data is generated by the monitored component at discrete points in time. We refer to one atomic item of discrete monitoring data as a *monitoring record*. We assume any type of discrete monitoring data as a sequence of time-stamped records, considering the timestamp to be part of the record itself.

In the context of ISSes, we categorize discrete monitoring data into unstructured-, structured- and transactional monitoring data; see Table 2.1 for details. This distinction is based on what we can assume about the individual records.

Description	Discrete Monitoring Data		
	Unstructured	**Structured**	**Transactional**
Timestamp	yes	yes	yes
Description	yes	yes	optional
Multiple and consistent attributes	no	yes	yes
strictly sequential	no	no	yes

Table 2.1: Categories of discrete monitoring data

We refer to our most general class of discrete monitoring data as *unstructured monitoring data*. Monitoring records of this class are only composed of a timestamp and a plain-text attribute. Although instances of the plain-text attribute might contain individual parameters and attributes, the structure is not consistent across all monitoring records. Examples of that type of monitoring data are Linux kernel ring buffer log files [78].

We refer to monitoring data that exposes multiple attributes that are consistent across the individual monitoring records to as *structured monitoring data*. Examples of such attributes are message identifiers, process names, component names and session identifiers. The reader should note that log-normalization technologies, such as the common-base-event format (CBE) [96], the WSDM Event Format (WEF) [93], and the Common Information Model (CIM) [29], attempt to transform unstructured monitoring records into a structured format. Those technologies assign the content of the plain-text attribute to a fixed and consistent set of attributes. In many cases, however, the source monitoring data is not rich enough to make an assignment, such that external data or expert knowledge is necessary to assign them. Examples of that type of monitoring data are BSD Syslogs [80] and Apache access log files [7]. In Apache access log files, the address of the requestor, the requested operation, the status code and the size of the returned object are consistent attributes across the majority of monitoring records. The BSD Syslog interface is usually configured to include the hostname, the process/module that created an event, the uptime of the

system and the description of the event.

Fault diagnosis using structured and unstructured discrete monitoring data is challenging in particular, because individual monitoring records do not generally contain sufficient operational context, small changes to the monitored system can generate a majority of the monitoring records and faults can have ambiguous manifestations in the monitoring data. A fault-localization technique also has to cope with inconsistent record types [98].

We refer to structured monitoring data as *transactional monitoring data* if it allows serializing individual monitoring records into an unambiguous sequence. This is done using a correlator attribute that enables correlating multiple records into a transaction. In many cases response time and component information is included to track the path and response time of individual transactions. Examples of this type are ARM and monitoring records obtained from IBM Tivoli Composite Application Manager (ITCAM) [51]. Usually enabling such detailed monitoring incurs high overhead in the system and is, as such, not the focus of this thesis.

2.3.4 Summary

In this section we discussed relevant concepts of monitoring software systems in general and ISSes in particular. Monitoring is a crucial part of system management, as we described in Section 2.3.1. We discussed various sources of monitoring data and their properties in Section 2.3.2. Because the primary focus of this work is on discrete monitoring data, we described it in Section 2.3.3 in detail. Although we included structured-discrete- and transactional monitoring data in the descriptions the primary focus of this thesis is on analyzing unstructured discrete monitoring data.

2.4 Text Mining

Our work focuses on analyzing unstructured records of discrete monitoring data. We need to utilize text-mining approaches to turn discrete monitoring data into a feature

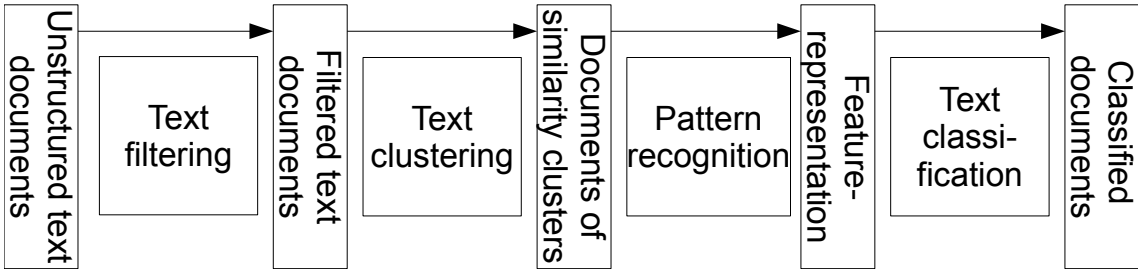


Figure 2.5: Conceptual workflow of relevant text-mining tasks

representation that can be easily processed. In this section we describe the tasks of text mining relevant to this thesis.

Text mining is a process to derive information from text. The process involves structuring, filtering, and devising patterns from the structured data. Tasks include text clustering, text summarisation, and topic detection. While the application areas of text mining are usually opaque, we use text mining for text clustering and supervised document classification. Figure 2.5 shows the conceptual workflow of the text-mining tasks that are relevant to us in order to model and classify discrete monitoring data. We describe the applications of the concepts discussed in this section to our work in detail in the Chapters 5 and 6, and describe related background to the individual tasks in the following sections.

The reader should note that the area of text mining also covers semantic approaches, such as topic modelling that models the underlying semantics of a text data set. These approaches are mostly unsupervised in establishing the semantics of the data set. Because our focus is on diagnosing recurrent faults and we assume the existence of labelled samples, unsupervised approaches are not in the scope of this thesis. The reader is referred to Hervé *et al.* [1], Dumais [30] and Blei *et al.* [15] for more information on this area of text mining.

2.4.1 Text Filtering

The filtering stage of text processing is a pre-processing step that removes observations from the input data that contain noise, missing data, or features that are deemed not relevant for the following steps of the analysis [13]. If no additional information is available the input data to this step is a set of documents that are represented as variable-length strings of characters. In order to apply meaningful filtering approaches the documents are structured. This structuring process varies by application area. For example, applications in web mining structure documents representing websites into collection of words. This is done by splitting the string that represents the document by a known set of delimiters. We refer to all string fragments that are not considered to be delimiters as tokens. Tokens that are deemed noise or deemed irrelevant are excluded from that representation.

Depending on the application area, the sequence of tokens is preserved or a bag-of-words model (i.e., the sequence of tokens is ignored, *see* Witten *et al.* [144]) is used. Other proposals include the use of q-grams [31]) or weighted relative frequencies of tokens [120]. For our application area we assume that the input data consists of individual records, we split the document first into records and then split the records into individual tokens, while preserving the order of these tokens and marking removed tokens with a place-holder. The output of this step is a sequence of tokens for each monitoring record.

2.4.2 Text Clustering

Once the data is structured in a representation that suits further processing, text clustering is applied in order to identify similarities in the data set. The general problem of clustering is to find a partition of data points such that all data points in one cluster are similar to each other. Traditional clustering algorithms use a distance function to assess the similarity of two points. Another class of algorithms is density based; the aim is to discover dense regions in the data set without the direct use of a distance function [140].

Traditional clustering methods used include k-means clusters that clusters the data set into k clusters, and hierarchical clustering, such as single-linkage clustering that seeks to

build a hierarchy of clusters. The complexity of k-means clustering is NP-complete [5] and the complexity of hierarchical methods is quadratic [124] with respect to the number of data-elements. Distance functions that are often used for text-clustering algorithms are derivatives of the Levenshtein distance [72].

Traditional clustering methods depend on a representation of points that have many categorical attributes, such as the vector-space model [120]. In natural-language processing this model is typically sparse such that not every attribute is present in each document. In addition, such a representation is often high dimension for natural language, and thus many traditional methods that were designed for low dimension clustering tasks fail to detect natural clusters [140]. For high dimensional data density-based approaches developed. These approaches first identify a seed of low-dimensional subspaces and gradually increase the dimensionality and verify if the cluster candidates from the low dimension subspaces are still valid clusters in a higher dimensionality. Examples of these type of algorithms include Vaarandi's algorithm [140], CLIQUE [4], MAFIA [38], CACTUS [35] and PROCLUS [3].

The reader should note that the data point representation largely depends on the text-mining objective. For example, web-search applications that aim to detect similarities between websites use a vector-space model [120] of the entire web documents to cluster the web sites by similarity. Our objective for this step is to detect and recognize patterns of the discrete monitoring records. Hence, we cluster the records rather than entire documents in order to perform pattern recognition on the clusters of records in the next step. For our approach, the output of this step is clusters of sequences of tokens.

2.4.3 Pattern Recognition

The general objective of pattern recognition is to assign a label to given input data. Our specific objective is to find patterns within the text clusters that were derived in the previous step. We refer to this step as encoding. Note that in the steps discussed so far we assumed that the sequence of elements was preserved. The objective of this step is to identify the longest-common subsequence (LCS) that describes each cluster, and thus the individual monitoring records of the input data.

A LCS of a set of sequences of tokens is the longest sequence of tokens that is a subsequence of all sequences of the set. The LCS problem is NP-complete [81] and is also shown to be difficult to approximate [63]. One of the traditional algorithms to solve the LCS is the Wagner-Fischer algorithm [12] that has a complexity of $O(m^k)$, where m is the length of the longest sequence of the input set and k is the number of sequences in the input set. The actual running time of this algorithm is not affected by other parameters of the input data, for example, similarities of the input sequences. A more recent proposal to approach the LCS-problem was proposed by Melchiar *et al.* [85]. Although the complexity of their proposal is still NP-complete, in general the running-time of their algorithm is heavily influenced by the properties of the input data and their memory complexity is better than the Wagner-Fisher algorithm. The factors that influence the running time are shown in Formula 2.1.

$$O\left(|A|\left(\sum_{i=1}^{m}|S_i| + m\sum_{a\in A}\Pi_{i=1}^{m}|S_i|_a\right)\right) \tag{2.1}$$

The parameter $|A|$ denotes the size of the alphabet of all tokens of the input sequences. The value of m is the number of sequences of the input data, $|S_i|$ is the length (i.e., in terms of tokens) of the sequence S_i and $|S_i|_a$ is the number of times the token a repeats within S_i.

The proposal by Melchiar *et al.* constructs the LCS of tokens for each sequence following these steps:

1. For each sequence in the cluster, construct a Directed Acyclic Subsequence Graph (DASG) that accepts all sub-sequences of tokens of that sequence (*see* Croucher-more *et al.* [27]).
2. Intersect all DASGs of the input data to form a Common Subsequence Automation (CSA) that accepts all common subsequences of the input elements (*see* Tronícek [137]).
3. Assign negative weights to each edge of the CSA and apply Dijkstra's algorithm [28] to find one solution for the LCS (*see* Melichar *et al.* [85]).

We illustrate the steps in Figure 2.6 for a set of two sequences. The alphabet consists of

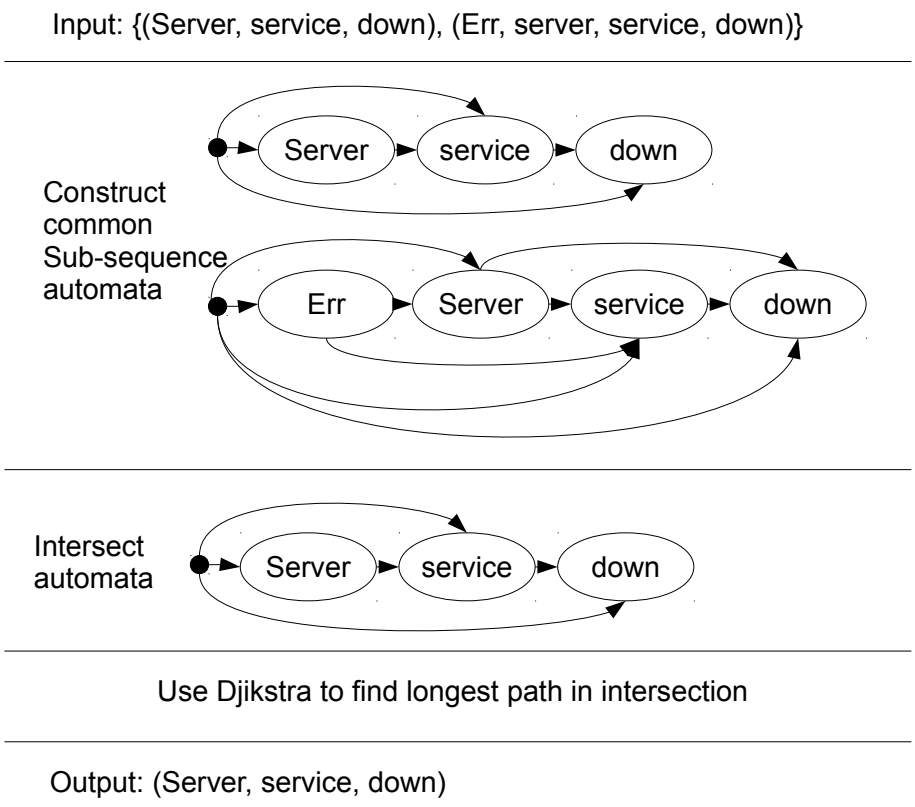


Figure 2.6: Illustration of LCS algorithm by Melchiar *et al.*

{ Err, Server, service, down }; therefore $|A| = 4$. The length of the sequences are $|S_1| = 3$ and $|S_2| = 4$. Since no tokens are repeated inside the sequences $|S_i|_a = 1$ for each sequence i and each token a. Because there are two sequences in the example, $m = 2$.

The reader should note that LCSs can be used to form a vector-space model [120] of the documents of the input data. If the input documents have labels, this representation can then be used to apply supervised learning and classification techniques that we discuss in the next section.

2.4.4 Text Classification

The final step that is relevant to this thesis is knowledge representation. Given a set of labelled documents that are now represented as documents having processable features, we need to find a representation of the knowledge that associates the labels with the documents such that new samples can be classified with a high confidence. Such knowledge is represented as a trained classifier that was previously trained from a set of labelled samples. Possible representations of such a trained classifier include decision trees, classification rules, probabilistic representations, and instance-based representations.

Decision trees consist of nodes that represent testing particular features of the input data. Leaf nodes assign a class label to all instances that reach the leaf. To classify an instance, it is routed down the tree according to the values of the attributes tested. For example, let the features of a set of text documents be represented as each document containing a subset of words of the entire corpus of words of the sample. The inner nodes of the learned tree are then represented as tests for the presence of particular words. In order to classify a document, it is routed through the inner nodes according to the words that it contains. The leaf node is then the assigned class label [144].

Classification rules are an alternative to decision trees. The rules consist of a set of conjunctions of tests that are applied to an instance to reach a conclusion about a class label of an instance. Classification rules can be directly inferred from decision-tree classifiers by creating a conjunction of nodes for each leaf (i.e., class label) to the root of the tree [144].

Probabilistic representations model the classifier as a probability distribution. This distribution assigns a probability for every class label to an instance. A popular example is used in document classification, where the labels are topics. For each instance the classifier assigns a probability for that instance to belong to individual topics [144].

Instance-based representations consist of a set of instances that have been memorized. To classify a new instance, the training set is search for an instance that most strongly represents the new one. In order to assign a class label to the new instance, the class label of the found training candidate is used [144].

Of relevance to our work are decision-tree classifiers. Since many log-reconciliation tools operate using rule-based representations to identify monitoring records [11, 97, 98, 103, 136] that are relevant for system management, focusing on this type of classifier allows us to integrate the classifier we learned into such existing tools. We have selected the C4.5 classifier [106] for this purpose. C4.5 recursively subdivides the instances of the training set to generate tests for relevant attributes. These tests represent the nodes. In order to decide which attribute to test for at the top level, the algorithm calculates the information gain for applying each test. The test that produces the highest information gain with respect to the training sample is chosen. This process is repeated recursively for the two partitions of the training set that remain after applying the test. The recursion terminates when there are no more attributes to test for, or the partition is pure (i.e., all instances only belong to one particular class label).

This type of naïve recursion, however, produces complex decision trees that tend to over fit the data [106]; therefore, C4.5 implements post-pruning techniques, including sub-tree raising, sub-tree replacement, and reduced-error pruning, that attempt to simplify the classifier after it was learned. The reader is referred to Quinlan [106] for the details of these techniques. The pruning heuristic relevant to us is reduced-error pruning. This heuristic attempts to simplify the learned tree by replacing sub-trees with leaf nodes. The decision to prune a sub-tree is based on an error estimate that is measured from the partition of instances that matches the sub-tree. Given a confidence, c, one can estimate the confidence limits, z, such that:

$$\Pr\left(\frac{f-q}{\sqrt{q(1-q)/N}}\right) = c \tag{2.2}$$

where N is the number of instances matching the sub-tree and f is the observed error rate (i.e., the ratio of instances not having the most frequent class label) and q is the upper confidence limit. Using that confidence limit, a pessimistic error estimate for each sub-tree can be computed as shown in the following equation.

$$e = \frac{f + \frac{z^2}{2N} + z\sqrt{\frac{f}{N} - \frac{f^2}{N} + \frac{z^2}{4N^2}}}{1 + \frac{z^2}{N}} \tag{2.3}$$

If the error estimate, e, in the top node of a sub-tree is smaller than the error estimates of the child nodes, the sub-tree is replaced with a leaf node that has the most prevalent label of the instances matching that sub-tree [144]. The computational complexity of creating the C4.5 tree is $O(mn \log n)$, where m is the number of attributes and n is the number of instances of the training set. The complexity of pruning is $O(n)$ [144].

An alternative to decision-tree classifiers are probabilistic classifiers such as the Naïve Bayes classifier [144]. The classifier uses a probabilistic measure to express the likelihood of an instance $\mathbf{D}$ belonging to a particular class. The naïve model assumes independence of the features contained in the feature vector; as such, the classification problem can be expressed as follows. Let $\mathbb{C} := \{c_1, c_2, \ldots, c_l\}$ be the set of classes, and $v_i \in \mathbb{V}$ the set of features that are associated with the samples.

$$\Pr(c_i|\mathbf{D}) = \frac{1}{Z}\Pr(c_i)\prod_{j=1}^{m}\Pr(v_j|c_i) \tag{2.4}$$

Prior knowledge that is modelled as the disribution $\Pr(c_i)$ and the independent probability distributions $\Pr(v_j|c_i)$ for the features of each sample $\mathbf{D} \in \mathbb{D}$ can be estimated by the relative frequencies from a labelled sample set. Z is a normalizing constant.

The reader should note that the classification approaches discussed are not limited to the area of text mining and are also commonly used to classify samples with numeric attributes. The most important task of text mining is the data representation as features that is performed in the previous steps.

2.4.5 Summary

In this section we described the relevant background needed in text mining to comprehend the methods that we use in order to model unstructured monitoring data (*see* Chapter 5) and learning and identifying symptoms of recurrent faults (*see* Chapter 6). The reader is referred to Chapter 8 for a discussion of performance measures relevant to the classification. We describe quality and performance measures for the classification as needed.

We acknowledge that this overview of text-mining tasks and approaches only represents a narrow subset the field of text mining that is relevant to this thesis.

2.5 Partially Observable Markov Decision Processes

In this thesis we propose an approach to recovery-driven monitoring that can incorporate knowledge of recurrent faults (i.e., their prevalence and symptoms), as well as relationships among the individual recovery actions. Our proposal leverages a model of a Partially Observable Markov Decision Process (POMDP) [69, 125] to accomplish this task. In this section we briefly describe POMDPs. A POMDP models a sequential discrete decision process of an agent with its environment. The POMDP is a tuple of $(\mathbb{S}, \mathbb{A}, \mathbb{O}, \mathbf{T}, \mathbf{\Omega}, \mathbf{R})$: $\mathbb{S}$ is the finite set of states of the environment that are hidden from the agent, $\mathbb{A}$ is the finite set of actions available to an agent to influence the environment, $\mathbb{O}$ is the set of all observations that can be emitted after an action has been performed, $\mathbf{T}$ models the transition probabilities of the environments' states, $\mathbf{\Omega}$ models the observation probabilities, and $\mathbf{R}$ models the rewards.

The environment is modelled by a set of states $\mathbb{S}$ that are hidden to the agent. An action $a \in \mathbb{A}$ taken by the agent causes the environment to transition into a new state $s' \in \mathbb{S}$ from $s \in \mathbb{S}$ with probability $\mathbf{T}(s'|s, a)$ and exposes an observation $o \in \mathbb{O}$ with probability $\mathbf{\Omega}(o|s, a)$ to the agent. Finally, the agent receives a deterministic reward $\mathbf{R}(a, s)$ for committing the action and the process repeats. Since the current state of the environment is hidden from the agent, the agent maintains a so-called belief state, $\mathbf{b}$, a probability distribution over states of the environment at any point in time. After committing $a \in \mathbb{A}$ and observing $o \in \mathbb{O}$, the given belief state $\mathbf{b}$ is updated as shown below [125].

$$\begin{aligned} \mathbf{b}_a^o(s') &= \nu \mathbf{\Omega}(o|s', a) \sum_{s \in \mathbb{S}} \mathbf{T}(s'|s, a)\mathbf{b}(s) \\ \nu &= \frac{1}{Pr(o|\mathbf{b}, a)} \\ Pr(o|\mathbf{b}, a) &= \sum_{s' \in \mathbb{S}} \mathbf{\Omega}(o|s', a) \sum_{s \in \mathbb{S}} \mathbf{T}(s'|s, a)\mathbf{b}(s) \end{aligned} \tag{2.5}$$

$\mathbf{T}$ and $\mathbf{\Omega}$ are assumed to be first-order stationary random processes and $\mathbf{R}$ is assumed to be deterministic. Astrom [9] has shown that under these assumptions the belief state is a sufficient statistic for all previous actions of the agent and referred to as being Markovian [125]. Since the belief state is Markovian, the optimal policy can be obtained by solving the optimal value function $\mathbf{V}(\mathbf{b})$ over the decision horizon. This function returns the optimal action $a \in \mathbb{A}$ for any given belief state $\mathbf{b}$. Note that the notation used in Equation 2.6 is a simplified from Kaelbing *et al.* [69]. The value function is linear and scalar. It assigns value for individual belief states; such that, $\mathbf{V}(\mathbf{b}) = \sum_{s \in \mathbb{S}} \mathbf{b}(s)\mathbf{V}(s)$.

$$\mathbf{V}(\mathbf{b}) = \max_{a \in \mathbb{A}} \left[\mathbf{R}(\mathbf{b}, a) + \gamma \sum_{o \in \mathbb{O}} \mathbf{\Omega}(o|\mathbf{b}, a)\mathbf{V}(\mathbf{b}_a^o) \right] \tag{2.6}$$

The parameter γ controls the influence of future rewards. Values close to zero of γ causes the agents optimal policy to be opportunistic; a value close to one maximizes expected future rewards. The value function for an infinite decision horizon is stationary [126]. Cassandra *et al.* [21] propose an efficient algorithm to approximate $\mathbf{V}$ for the infinite decision horizon. In order to implement a feedback loop, the model only makes an observation after an action has been performed and, therefore, only considers state changes in response to agent actions. This property implies that the controller has to have full controllability over the system, such that no other event than the controller actions can change the system state.

2.6 Summary

In this chapter we described the background relevant to this thesis. We introduced key concepts and terms in Section 2.1, outlined causes of failure in Section 2.2 and discussed the concepts of monitoring ISSes in Section 2.3.

An integral part of our contribution is the model of discrete monitoring data discussed in Chapter 5 as well as the learning and identification of symptoms of recurrent faults discussed in Chapter 6. Both of these contributions rest firmly on advances in text min-

ing that we described in Section 2.4. Further, we described the concept of the Partially Observable Markov Decision Process (POMDP) that we use to model system recovery in Section 2.5.

Having introduced the basic concepts of dependable computing, as well as algorithms that we use for our approach, we discuss related work in the next Chapter.

Chapter 3

Related Work

The concepts described in the previous chapters are generic and not all tolerance techniques apply to our problem domain of Information Software Systems. Furthermore our work focuses on learning recurrent faults from discrete monitoring data. In this section we survey related approaches that use discrete monitoring data.

The primary motivation to use discrete monitoring data such as log files is their availability in most existing ISSes. Most legacy ISSes log status changes into some form of discrete monitoring data by default. Therefore, this type of monitoring of ISSes does not create additional overhead to the running system. However, processing, transmitting, and storing discrete monitoring data incurs overhead depending on where the data is collected and processed. A principal drawback of relying on such monitoring interfaces is that the context of any form of error detection and diagnosis is limited to whatever the developers decided to log. Therefore the coverage of that interface may be limited and the context of the monitoring records during operations may not correspond to the assumed context of those records during the development phase of the system [98].

A lot of work as been done in the area of error detection and failure prediction using logs. Error detection is necessary to assess the operational status of a system. If the system is deemed erroneous, diagnosis and recovery are attempted. Although discrete monitoring interfaces may only provide a limited coverage of the overall system state, machine-learning

methods can be devised relatively easily that approximate the overall behaviour of the system and identify potential errors using anomaly detection or failure prediction. From our perspective using such interfaces as the only source for error detection may not provide enough coverage to reason about the overall health of the system, precisely because the state coverage of these interfaces is limited.

In order to contribute to the research community, our focus is primarily on fault diagnosis, as such we assume the existence of a reliable error-detection mechanism that indicates that the system has failed when the diagnosis takes place. This error detection is not restricted to log analysis, but could also be based on continuous-monitoring interfaces such as metric monitoring [62, 91] or monitoring based on other interfaces and, possibly, active probes [40, 55].

From our perspective the number of contributions that focus on fault diagnosis using discrete monitoring data is limited. This may be due to the issue of limited operational context. Many proposals we encountered rely on structured discrete monitoring data to leverage special attributes, such as component identifiers, that may aid fault localization only. Fault localization approaches attempt to localize the component that experienced the fault but usually do not attempt to attribute the type of failure.

We have chosen a different and novel approach to fault diagnosis by focusing on diagnosing recurrent faults from discrete monitoring data. Research has shown that failures experienced by ISSes are often caused by recurrent faults. Learning from previous instances of a fault may provide more context to the diagnosis of that fault. Our experience with IBM has shown that this reasoning is consistent with the work of support personnel. Our aim is to automate this process by analyzing discrete monitoring data that is available at low system overhead.

Although error detection using discrete monitoring data is out of scope for our approach we survey a selection of approaches in Section 3.2 to provide the reader with context of existing work and explain why these approaches are not relevant to our proposal.

In Section 3.3 we describe approaches to fault diagnosis. We distinguish these approaches by the type of discrete monitoring data used (*see* Section 2.3.3). In this section

we also briefly survey approaches that use structured discrete and transactional monitoring data that are not directly relevant to our proposal because our focus is on unstructured monitoring data. We highlight the respective differences where needed. Our objective is to make only minimal assumptions about the structure of the monitoring data.

Since our approach also entails automated system recovery, we survey related approaches in Section 3.4. Existing research on automated system recovery focuses on applying corrective maintenance actions that may involve isolation, reconfiguration, or reinitialization of system components.

3.1 Modelling Monitoring Data

In this section we describe relevant approaches to model unstructured monitoring data. Modelling unstructured monitoring data entails two principal problems:

1. How are individual samples of monitoring data (e.g., separate log files) consisting of those records modelled.
2. How are the individual discrete records modelled.

The first issue depends on the objective of the modelling and constraints imposed by the monitoring interface. As we will show in Section 3.2, many approaches that use structured and unstructured monitoring data for error detection typically model separate samples as sequences of monitoring records. The monitored systems are typically single threaded, or the record format contains thread identifiers that allow a sequencing of individual records. The reader should note that our approach to unstructured monitoring data does not assume the availability of additional attributes. We do not consider the records in a strict sequence, but rather consider the mixture of records over fixed-time periods. A related approach that follows the same objectives is Vaarandi [140]. The approach is used to summarize the behaviour of the system and ignores the ordering of records. The output is a summary of patterns that describe the records of the sample.

The second objective depends on the structure of the monitoring records itself. The objective of record modelling is to assign a nominal type to each of the records. Many

approaches in error detection as well as fault diagnosis assume the availability of special attributes in the records. Therefore the application of such approaches is limited to monitoring interfaces that actually expose these attributes. We consider such approaches to fall in the class of structured monitoring data. The type of the record is determined by the value of one or more attributes that are consistent across all log records.

Unstructured approaches model the monitoring records as time-stamped base strings with varying parameter attributes. Typically the records are time-stamped when they are logged by the component and then forwarded to the monitoring interface. It could, however, also be the case that the monitoring facility performs the time-stamping of records. In large monitoring infrastructures where an event is passed through several components before it is logged and time-stamped this may create a clock skew; such that the event receives a significantly different timestamp from its actual occurrence. This issue can also cause events to be recorded in a different order than their creation. The problem of clock skew needs to be closely investigated by approaches that model discrete monitoring data as a sequence of records.

The type of a record is determined by the base string that formed the record. The following records may have been the result of the base string `Server %i service %i down`. All records belonging to the same base string are considered to be of the same type.

```
Server 1 service 4 down
Server 2 service 5 down
Server 3 service 6 down
```

Such base strings can, in certain cases, be inferred automatically from the source code of the application. For example, Xu *et al.* [146] propose an approach that extracts the record-types of console logs from the source code of applications. Because we do not assume the availability of the application source code their approach is not relevant to this thesis. Other approaches [86, 134] entail the use of grammar-inference techniques to learn a grammar of dominant record types and flag outliers as anomalies.

Closest to our proposal are the works by Fu *et al.* [34], Jiang *et al.* [64] and Vaarandi [140]. All of these approaches employ text mining (*see* Section 2.4) to approximate the base strings

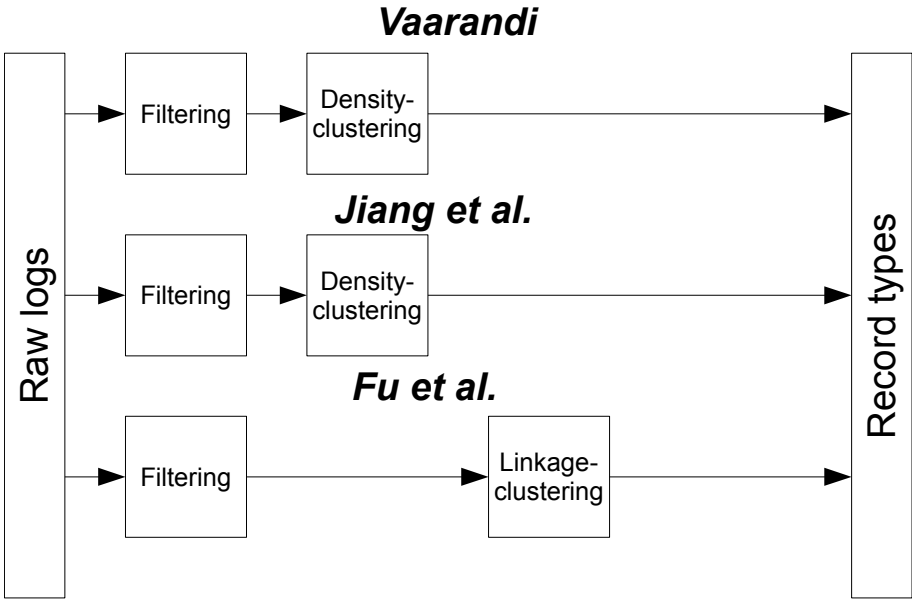


Figure 3.1: Conceptual comparison of unstructured log modelling approaches

of monitoring records. A conceptual comparison of these approaches is shown in Figure 3.1. All approaches first tokenize the individual monitoring records and filter out tokens according to predefined rules. All of these proposals model the features of log files as base strings and parameters of individual log messages. Conceptually all proposals employ clustering algorithms to cluster the monitoring records into clusters of similar log records. Fu *et al.* decided to use single-linkage clustering using a special variant of the Levenshtein distance. Since single-linkage clustering requires the use of a cut-off threshold for the distance measure, they propose a heuristic to determine it automatically. In order to determine the cut-off threshold they cluster the entire data set first using 2-means clustering and then using the inter-cluster distance as cut-off threshold for the single-linkage clustering. The reader is reminded of the discussion of Section 2.4 that both of these algorithms have quadratic complexity. As we will show later, the approach by Fu *et al.* incurs substantial overhead. Surprisingly, the authors did not state the size of their input data set.

Jiang *et al.* use a custom density-based clustering algorithm that has linear complexity. Instead of using a distance measure, their approach defines a number of instances (by default 10) of similar filtered log records that occur to be considered as a cluster. The output of this algorithm is a set of clusters of filtered log records. Each cluster approximates a certain record type.

Vaarandi proposed a very effective algorithm to summarize individual log files. The objective of the original algorithm was to reduce the amount of data a support person has to evaluate in order to assess the monitored-system's behaviour. It is a density-based clustering algorithm that infers frequent patterns from log files. Vaarandi's density-based clustering algorithm has linear complexity with respect to the number of filtered records. The algorithm has two sensitivity parameters as tunable parameters. The user can configure the sensitivity toward frequent tokens and also the number of similar filtered log records to occur to be considered as a cluster. The unique property of Vaarandi's algorithm is the representation of clusters as regions of density. Instead of representing each cluster as a selection of similar filtered records, the algorithm provides a tuple of tokens and spaces as cluster. A record matches that particular type when its tokens appear at the same positions as the tokens of the cluster. Spaces are used as a place holder to match any token. Because the algorithm considers the tokens at absolute positions, it is very sensitive to the choice of delimiters that are used to convert the raw unstructured records into their filtered representation.

3.2 Error Detection

Fu *et al.* [34] leverage unstructured log files to perform error detection. They model the features of log files as base strings and parameters of individual log messages. They use a finite-state machine model to model the sequence of monitoring records. If an outlier occurs (i.e., a log record is seen that cannot be matched by a transition), their model flags it as anomaly.

Salfner [118] applies a modified Hidden Markov Model (HMM) (*see* Rabiner [107]) to

pre-processed log files in order to detect errors and identify different failure modes. The author's method takes annotated samples of log files that contain the time and duration of the failure and trains an HMM according to the sequence of observed events. Salfner uses spatial-temporal filtering similar to Oliner *et al.* [98] in order to limit the amount of training data taken into account. They evaluate their model on log files obtained from telecommunication networks. In order to perform record-type approximation they apply Levenshtein-distance-based clustering to cluster similar monitoring records into particular failures. Log files obtained from telecommunication, unlike log files from ISSes, are rich in structure and the logged events are well chosen and described. Therefore, the logged events preceding a failure in that context are highly correlated with failure. ISSes log files, for example application server log files, are written to by many concurrent processes. The log-file records are not necessarily restricted or well chosen. The log file may contain several monitoring records that precede a failure but are totally unrelated. Since Salfner's log model requires the use of special attributes and expert knowledge, we do not consider that model relevant for this thesis.

Furthermore, most of the model parameters of Salfner's approach, such as the number of states of the HMM and the threshold for the Levenshtein distance are expert-chosen. Therefore, they need to be hand-picked for different types of log files.

Yamanishi *et al.* [147] propose a generative HMM-mixture model for Syslogs. They model Syslogs as being generated by concurrent sessions. Each session corresponds to one hidden Markov model. They propose a training algorithm based on the Expectation-Maximation (EM) algorithm (*see* Hastie *et al.* [43]) to train their model from historic data. They assign an anomaly score to individual observed sequences that is based on the inverse log-likelihood for the sequence to occur (i.e., unlikely sequences of events are flagged as being abnormal). The authors do not provide an exact definition for the term "session" and do not describe how the session labels were established for the training data or how the threshold for attributing abnormal sequences was chosen. In addition, their dynamic model selection algorithm described in the paper appears to maximize the likelihood for an observed sequence for an individual HMM of the mixture. In other words, each sequence of events is assumed to have come from an individual HMM. Therefore the accuracy and

robustness of their approach is highly sensitive to concurrent sessions in the log file. This issue is not directly addressed by the authors. In order to consider the events in sequence they have to rely on special attributes that identify sessions. Since our objective is the analysis of unstructured monitoring data, that model is not relevant to us.

More work has been done in the domain of intrusion detection systems to diagnose root causes of potentially malicious failures. Julish [68] describes a method similar to that of Hellerstein *et al.* [45], and attempts to cluster similar manifestations of malicious failures by identifying episodes of events that appear to correspond to a particular failure. His input data is structured and obtained from intrusion detection systems. Since our focus is on diagnosing non-malicious faults, and the monitoring data obtained from information software systems is inherently different from intrusion-detection systems, his methods are not directly applicable.

3.3 Fault Diagnosis Using Log Data

In this section we describe selected approaches in fault diagnosis that use various forms of discrete monitoring data. Our focus is recurrent fault diagnosis using unstructured monitoring data. However, in order to provide the reader with context in this research area, we also survey approaches that use structured monitoring data. We begin by describing approaches that use unstructured monitoring data in Section 3.3.1, approaches using structured monitoring data in Section 3.3.2 and approaches using transactional monitoring data in Section 3.3.3.

3.3.1 Unstructured Monitoring Data

Xu *et al.* [146] propose a method for error detection and fault localization for unstructured console log files. Their method consists of two stages. They first perform source-code analysis in order to approximate different classes of monitoring records and their parameters. After that, they employ Principal Component Analysis (PCA) (*see* Joliffe [66]) in order to

detect errors. Their proposed approach only works post-mortem, requires a large amount of monitoring data to yield significant outcomes, and assumes the availability of the source code.

Vaarandi [140] analyzes several clustering techniques for high-dimension data to approximate record types from samples of discrete monitoring records. Vaarandi's objective is to summarize the dominant behaviour of a system by analyzing the log files. This summary is than reconciled by the operators of the system to detect and diagnose potential problems. While Vaarandi's objective was primarily data summarisation, its low running time makes it very attractive for log clustering. We use Vaarandi as part of our approach for that particular task.

Peng *et al.* [101] attempt record-type categorization using a Naïve Bayes classifier and require labelled samples. Furthermore, they need to incorporate temporal dependencies in order to yield acceptable record-type prediction accuracy. Their evaluation only takes into account eight different record types. Stearley [128] also attempts record-type categorization using the Teiresias algorithm, but the paper indicates that Teiresias incurs a very high overhead and is, therefore, not considered relevant for our approach.

Oliner *et al.* [97] focus on fault detection using unstructured monitoring data from supercomputer logs. They split the monitoring data obtained into so-called node hours, in which each node hour contains one hour of monitoring data from a particular machine of the supercomputer. They propose an entropy-based scoring function that assigns a higher score to words that are only contained in a small subset of node hours. In more recent work of Stearley and Oliner [129], they also take into account the size in bytes of each node hour. This technique can also be applied online by analyzing the events of the last hour of the monitored system. The technique is straightforward and does not require a structured log file. Unfortunately, the authors do not elaborate how heterogeneous workloads and varying configurations of particular nodes affect the diagnostic accuracy. As such we expect this technique only works in environments that have replication and homogeneous workloads and configurations. Such a configuration would allow the peer comparison proposed by the authors. Furthermore, it requires excessive filtering and data transformations before it can be applied.

3.3.2 Structured Monitoring Data

Bodik *et al.* [16] propose a dynamic model-based visualization of Apache log files. They identify failed requests based on HTTP status codes [32]. Their focus is limited to the web logs. They correlate failed requests with individual web pages and employ Pearson's χ^2-two-sample test [92] to localize web pages that show a significant failure rate. Their approach is limited to fault localization and does not maintain a history of previously localized faults to improve the prediction accuracy over time.

Hellerstein *et al.* [45] propose a model for creating correlation rules applied to events and evaluate their approach using SNMP [42] events to identify actionable patterns, including symptoms of failure, in a distributed system. They describe pattern-discovery methods for events to identify temporally correlated events. Specifically they seek to identify event-bursts, partially periodic patterns and mutually dependent patterns. The discovered patterns are encoded as Event-Relationship-Networks (ERNs) [135] and presented to an expert audience for review. Specifically, they propose a grouping algorithm over multi-attribute events to reduce the complexity of the following pattern-discovery methods. The set of attributes that is used for the grouping needs to be chosen by an expert. Furthermore, this approach requires expert attribution for each discovered pattern. Their described pattern-discovery methods require an expert to set one or more parameters. In addition, expert knowledge is needed to attribute the type of the discovered patterns. The author does not make any claims whether he discovers failures or other patterns.

Li *et al.* [73,74] describe another method for grouping monitoring records exposing multiple attributes. They identify groups similar to Hellerstein *et al.* [45] and then mine those groups for temporal dependencies in order to leverage patterns from log files for system management, including problem detection. They group individual monitoring records by their state or situation and component attribute and then seek to identify pair-wise dependencies among those groups. Although they claim that they perform text mining, their method only relies on the use of CBEs [96], which transforms the raw text-log files into a format that exposes multiple attributes. Their method has tight configuration requirements, requiring log files (or CBE adapters) that at least expose a timestamp, component,

and state or situation attributes. Furthermore, they seek to merge log files from multiple components within a system and, as such, require synchronized clocks of these components or post-fact synchronization. The biggest shortcoming of the method proposed by Li*et al.* from the perspective of fault diagnosis is that each discovered pattern needs to be manually attributed as faulty or not. They provide no elaboration on how, for example, novelty detection in that context could be used to spot potential failures or configuration changes.

Furthermore, Li *et al.* [74] describe an incremental improvement that incorporates the method described by Li *et al.* [73] into a systems-management tool. They use their previous method to identify errors from network traffic. The traffic is visualized in a management console as a histogram, as well as the time-line of statistical measures, such as the Shannon entropy (*see* Shannon [123]), defined over this distribution. The past samples for measures are gathered from historical data stored in a database. A system administrator using that tool can set thresholds over these measures that are then used to identify errors. In addition to the weaknesses of the approach by Li *et al.* [73], the thresholds set may not accurately identify the fault. Furthermore, the Shannon entropy may be hard to interpret by a system administrator.

Liang *et al.* [75] propose a method for error detection and localization for structured discrete monitoring data. They survey a collection of logs obtained from the BlueGene/L supercomputer. The log format of this system is structured and events expose individual fields, including time, type, severity, component and location. Assuming that events corresponding to faults occur in bursts, they propose a spatial-temporal filtering scheme to limit the number of events emitted by different components. They only consider monitoring records of failure or fatal severity. In more recent work [76], the authors describe how to predict failures of individual components based on the filtering scheme, but their focus is mainly on predicting failures caused by hardware faults and environmental faults that can be fit to stable distributions. Oliner *et al.* [98] propose a static rule-based method similar to the ones proposed by Liang [75, 76] for error detection and fault localization for structured monitoring data. They survey a collection of logs obtained from five different supercomputers, describe existing fault-diagnosis mechanisms based on system alerts, and

propose improvements. Their study only surveys monitoring records that expose explicit attributes, such as severity (e.g., fatal, failure, severe, error, warning and info) and source component. They conclude that existing fault diagnosis is done manually by system administrators who tag offending records through log view tools. Individual events are filtered by their severity and by correspondence to a failure manually encoded as a regular expression. Unfortunately, the approaches described by Liang *et al.* [75,76] and Oliner *et al.* [98] require an event format that exposes at least the attributes timestamp, severity and source component, but many structured monitoring records generated by different ISSes components do not expose any or only a subset of these attributes. In addition, by applying spatial-temporal filtering multiple independent faults activated in a particular component within a short time frame may be impossible to detect. Furthermore, their methods focus on detecting and diagnosing hardware and environmental-related failures that are an issue in the field of supercomputers but only account for a small fraction of failures experienced by ISSes and cannot, from our perspective, be used directly to diagnose failures caused by misconfiguration or software bugs.

Sabato *et al.* [115] propose a ranking scheme for individual monitoring records to aid in error detection and fault diagnosis. Their method operates on structured monitoring data that can be easily associated with record types. Using a large baseline sample of log files they establish a cumulative distribution function for the occurrence of individual record types in a log file. Record types with lower occurrence probability are ranked higher than record types with high occurrence probability. In order to limit the effect of workload changes on their approach, they group the baseline into log files that have similar use cases. The biggest disadvantage of their method is a large number of training samples they need in order to establish a valid cumulative distribution function of record types. Furthermore, minor configuration changes or upgrades could invalidate the previously established baseline and, hence, the CDF. This issue could lead to concept-drift in a realistic setting.

Sahoo *et al.* [116,117] analyze several machine-learning and pattern-discovery methods in order to predict critical events and system performance from collected resource usage metrics and system log files from supercomputers. They analyze the statistical properties of failures in distributed computer systems and conclude that only a small fraction of

monitoring records correspond to symptoms of failures, stochastic failure and error processes exhibit strong auto-correlation and possibly periodic behaviour. Furthermore, they find that failures are not uniformly distributed among different machines and that a small number of machines experience most failures. Furthermore, they compare the performance of rule-based methods with Bayesian networks [43] in predicting critical events. In their model, however, critical events translate to exceptional conditions that need to be specified or attributed manually. In contrast to other studies they propose this technique for life-monitoring and show that it is possible to predict some (hand-picked) events with a high accuracy using time series analysis over a fixed sliding window. However, their model does not solely rely on discrete monitoring information. They gather resource usage metrics of the system and integrate them into their prediction model. Based on the collected performance metric samples and the log files, they create a joint monitoring record that includes CPU utilization, record type and the inter-arrival time. This format is very particular and can only be applied in that setting.

More recent work by Liang *et al.* [77], based on the outcomes of Sahoo *et al.* [116, 117], attempts to predict failures using the monitoring data from the past hours from supercomputer remote-serviceability-logs (RAS). They compare several supervised learning techniques, including support vector machines (SVM) (*see* Witten *et al.* [144]) and RIPPER (*see* Cohen [25]). Their method requires a distributed system that exposes failure rates that can be fit to a stationary distribution. Furthermore, their method requires a monitoring format that includes a severity attribute. Their training set uses the severity attribute to identify fatal events, although, the work done by Oliner *et al.* [98] has shown that the semantics of the severity attribute are limited. Some events marked as fatal do not necessarily correspond to failures and some events marked as being informational could indeed correspond to actual failures.

Chieslak *et al.* [24] seek to analyse the root cause of failed jobs in the context of grid-computing. They assume knowledge of whether a job has failed and then employ C4.5 decision trees [106] over the logged properties of the job to aid in fault diagnosis for the failed jobs. The tree leaves correspond to failed or successful jobs and the edges to values of particular properties known for the jobs. Although they do not explicitly state monitoring

as source for this data, the properties could be inferred from structured system logs that store the return codes of individual jobs.

Huang *et al.* [49] propose a data-stream system to analyze the health of a database system. They place monitoring probes into the database software to gather event data. In contrast to all approaches we discussed so far the event are stored in memory for rapid processing. As it is our objective to not introduce additional probes into the system such approaches are not of relevance to us.

Kavulya *et al.* [70, 143] describe techniques to model the behaviour of telecommunication log files in order to diagnose chronics (i.e., system failures that do not trigger error detection) and events of interest to system management. In their model chronics are system failures that to not necessarily get detected by conventional error detection approaches. Chronics are often transient. Although the outcomes of this approach are promising the authors depend on very detailed system log files that expose consistent attributes such as component identifiers, caller identifiers and defect codes. The application of their approach, therefore, appears to be limited to telecommunication systems that expose this set of attributes. Because our objective is to make as little assumptions as possible about the nature of the input data we do not consider this work relevant to this thesis.

3.3.3 Transactional Monitoring Data

Maruyama *et al.* [83] propose a model-based method to localize faults by comparing function-call traces obtained from presumably fault-free executions of programs against traces that were obtained from cases where a fault is present. By comparing the traces they seek to highlight functions that are frequently called during normal execution but absent when failure is present and functions that are predominantly invoked when a failure is present. They propose a scheme that groups different consecutive function executions into execution blocks. Their model, however, requires a specific instrumentation of the code that is not present in most legacy systems and not available from ARM [65]. Furthermore, monitoring at the level of detail that the authors require incurs performance overhead. In

addition their proposed approach only covers fail-stop cases of jobs that have a limited run-time. It is hardly applicable to ISSes that are running continuously.

Sandeep *et al.* [121] propose a model-based approach for structured monitoring data that attempts to diagnose performance problems in network-attached storage environments (NAS). They analyze historic performance logs of the NAS software that expose resource usage metrics in order to classify and predict the workload experienced by the NAS. Although they claim that they perform error detection, their system requires expert knowledge of when a new failure occurred. After the cause of a failure is attributed they identify the workload when the environment was deemed fault-free (i.e., before the fault) and identify those counters that violate the hypothesis of the identified workload after the fault has happened. Their approach is limited because it requires labelled samples of historic workloads and a particular format of log files that exposes resource usage metrics. Their analysis focuses on a single NAS instance and they do not address how to cope with clustered NAS instances that would be present in general distributed system environments. Fault diagnosis in a replicated environment would require manual error detection.

Chen *et al.* [23] propose a dynamic model-based approach to locate faults using transactional monitoring data. They instrument J2EE applications to trace individual requests through the software components that are used for web applications. They inspect the HTTP traffic in order to identify failed requests from HTTP status codes [32]. For each request they identify the component types that the request passed through. After many failed requests have been identified they highlight the components that show the strongest correlation with failed requests. Their approach is accurate in locating the root cause in the intended use environment, but requires the extraction of the entire transaction path. We have shown [91] that extracting the complete path information from individual monitoring records can create an analysis challenge because in such systems the individual components are invoked by different requests concurrently and individual traversals can be logged out of order. In addition, monitoring interfaces are sometimes unable to log failed requests, so the fault coverage may be limited in a practical scenario. Furthermore, the authors need a large amount of data in order to locate the cause accurately. Furthermore, Chen *et al.* [22] propose an approach that automatically infers rules identifying faults for

transactional monitoring data. They focus their evaluation on fault diagnosis within the eBay monitoring infrastructure. The infrastructure monitors the flow of web transactions across different application servers. The records are harvested at a central location. Each monitoring record can be associated with a particular user transaction. From the monitoring records it is also possible to infer the start and stop of a transaction. Furthermore, the records expose the attributes request type, request name, host name, pool name, version, timestamp and status of the request. The authors build a multi-layer decision tree classifier over the attributes of the monitoring records. The individual nodes correspond to particular attribute values and leaves correspond to failed or successful requests. When pruning the successive branches of the tree, the tree shows attribute values for failed requests that aid in further root-cause analysis. The authors assume that they can accurately determine the outcome of a request as failed or succeeded, which works in their setup because the monitoring format explicitly exposes a request status attribute.

Gao *et al.* [36] propose a codebook-based approach for transactional monitoring data. They attempt to construct a set of transactions that covers all monitored components. Assuming that a transaction would fail whenever a component that is traversed by the transaction fails, they attempt to isolate the intersection of components of failed transactions and attribute them as root-causes. In order to cope with the NP-completeness of an optimal codebook solution, they propose two heuristic approximations. One of the main drawbacks of codebook-based approaches, in general, is that they usually provide good system coverage but a relatively poor state coverage. A failure in a system may only be caused if certain parameters are set. In addition, this method can only diagnose total failures. For example response-time degradation will be hard to catch with a codebook.

Brodie *et al.* [17] and Modani *et al.* [89] developed models to diagnose recurrent faults by indexing and matching call stacks. Call stacks provide very detailed information about the software system such as the sequential invocation of functions and system calls within the monitored software. Continuously monitoring such traces incurs substantial overhead therefore they are usually generated when the system has failed. In order to be able to process such data the structure of these call-stacks needs to be known. We do not consider the approaches relevant to our proposal because it requires us to make additional

assumptions about the availability and the structure of such stack traces, but their findings are in so far relevant to our proposals because the authors show that a majority of system failures is caused by recurrent faults.

3.4 Proactive System Recovery

In the past decade the vision of autonomic computing [71] gained attention, promising self-managing data-centres that minimize human intervention and, therefore, operating cost. The proposed dimensions of self-management are self-configuration, self-optimization, self-protection, and self-healing. The self-healing dimension consists of automating error detection, root-cause analysis, system recovery and retest.

The application of the autonomic-computing paradigm mandates a system model that includes autonomic elements that are managed using autonomic agents in accordance with predefined policies. An autonomic manager continuously monitors the state of the managed element and its environment. The monitoring data is analyzed to develop and execute plans to manage the element in accordance with predefined policies. For example, an autonomic agent could monitor the log files of a managed web server to detect and diagnose a failure. Using the available information, the agent would match the diagnosis against known fix-packs and install the appropriate fix-pack or alert a system administrator. Unfortunately, the operational context (i.e., the expected system behaviour) logged by many forms of monitoring, such as log files, is too limited to diagnose causes of failure with acceptable confidence [98]. Therefore other proposals [18,40,55] focus on recovering the system rather than on accurately diagnosing the cause of a failure. We denote this domain as recovery-driven monitoring. Recovery-driven monitoring, in contrast to self-healing, only relies on accurate and prompt failure detection and not on an extensive root-cause analysis [18]. Implementations of this paradigm reach from design-for-reboot [19, 20] to policy-driven system recovery [40, 55]. Isard [55] proposes a model that comprises recovery agents for elements of the software system. His model defines the states as healthy, failure, and probation; and contains a small set of recovery actions. Upon error detection, using a

watchdog, an agent is placed into a failure state and a recovery action is attempted; after that, the agent transitions into a probation state. It remains in this state for a period of time. If within that period no other error is detected by the watchdog, the element is placed into the healthy state. If, however, an error occurs inside the probation state, the agent is placed into the failure state again and another recovery action is chosen. According to the author, the choice of the recovery action depends on the history of previous recovery actions and the cost of the recovery action. From our perspective this model does not scale with the number of possible recovery actions. Assume, for example, a system fails and can only be recovered by a costly recovery action. Until this recovery action is chosen a long trial of other recovery actions will be performed. Furthermore, symptoms of known faults, which could be obtained from support databases of software vendors and tested against monitoring interfaces exposed by the system, are not considered. Furthermore, the recovery actions performed are modelled to be independent of each other. In many cases, recovery actions may be subsumed by other recovery actions. For example a complete system reboot subsumes any micro-reboots of its components. When deriving an optimal recovery policy such relationships should be taken into consideration.

3.5 Shortcomings of Prior Log Analysis Work

In this section we summarize the shortcomings of prior work in the area of fault diagnosis using discrete monitoring that are relevant to our work. At this point we want to reiterate our objective to make only minimal assumptions about the monitored system and the structure of the monitoring data. Furthermore the monitoring interfaces we consider relevant do not involve additional system probes or monitoring data other than discrete monitoring data.

Many approaches to system modelling require knowledge of the system structure: In related work to our research, many existing approaches do require knowledge about the system structure to locate a fault. These approaches assume the availability to compensate for the lack of operational context in the monitoring data. Unfortunately,

data about the system structure may be invalid, outdated, or incomplete. In addition collecting such data through reverse engineering maintaining it creates additional system management overhead. We want to avoid introducing additional system management overhead.

Our solution approach, however, only requires a mechanism to identify components from the monitoring data. Several log formats, such system log files, already capture component identifiers. In other cases, log files are only written by individual components and hence the component association is implied. Unfortunately, a log-modelling technique needs to be aware of such attributes to take advantage of that correlation.

Meta-data needed for log-file analysis: Existing work relies on the presence of special attributes, for example to identify services and sub-components from the stream of monitoring data. While the log format may be well defined for some services under normal operation, established logging formats are frequently violated to include error data (e.g., stack traces and core dumps) if the system fails. Such approaches may characterize the behaviour of a system under normal operation and may flag an error; unfortunately, they cannot process exceptional log records and, hence, are not suitable to fingerprint such behaviour accurately.

Models derived from log files do not scale well with the amount of log data collected: Existing work on log-file processing, with the exception of Vaarandi [140], usually requires a small data set to return results in acceptable time. Such approaches operate against well-defined logging interfaces that impose rigid constraints on what events are logged and in which format. Therefore, the amount of log data to process is usually small compared to our scenario.

Existing bias on manifestations of failure: Some approaches already formulate a bias on how failures manifest in the monitoring data. Such assumptions include, for example, periodicity of events or bursty event generation. While existing work on hardware failures in supercomputers [98] or chronic events in telecommunication logs [70, 143] has shown that some of these assumptions may be valid, our experience shows that such assumed patterns may not be valid for software defects.

Fault diagnosis approaches still require manual effort: Most of the existing approaches we surveyed on error detection and fault diagnosis may flag an error or a likely root-cause respectively, but do not steer recovery based on these findings. Yet most of them claim a reduction on manual diagnosis effort. We believe that by combining recovery and diagnosis we could reduce manual effort even more. Even if an automated recovery attempt fails it provides a more in-depth insight into the system behaviour than current approaches that rely only on the analysis of monitoring data.

Automated system recovery does not take into account symptoms of recurrent faults that could be obtained by log analysis: We are not aware of existing work that considers symptoms of recurrent faults to steer automated system-recovery efforts. Most approaches we have surveyed employ a rule-based cost model that acts upon failed or succeeded recovery attempts, without taking into consideration monitoring interfaces of the system. Whenever a recovery attempt fails a recovery action that is equally or more expensive is chosen. We believe since those actions are aimed at recovering from highly recurrent faults, the approach can benefit by taking into consideration log-file analysis for recurrent fault detection. That could enable recovery actions to be scheduled more efficiently.

3.6 Summary

In this chapter we discussed several approaches to modelling unstructured monitoring data, error detection and fault diagnosis using discrete monitoring data. Furthermore we surveyed related proactive system recovery approaches. At this point we want to remind the reader that our primary objective is to do recurrent fault diagnosis using unstructured discrete monitoring data. We acknowledge the existence of other approaches that use other types of monitoring data (for example, [49, 91]), but they are not relevant to this thesis because they require expert knowledge about the system structure, the monitoring format, the source code of the application, or additional probes being introduced into the system.

One of the main contributions of this thesis is a model for unstructured monitoring

data. We discussed the concepts of log modelling and surveyed related approaches.

After having introduced log modelling, we surveyed error detection techniques to exemplify how such log models are used to reason about system health. While our focus is exclusively on fault diagnosis this section provides some insights to where related log models, in particular [34,64], are used. In contrast to fault diagnosis, error detection only yields a binary outcome, whether the system is experiencing and error or not. Such approaches do not attempt to attribute the root-cause of a problem.

After that we surveyed approaches in fault diagnosis. Based on our investigation, the majority of proposals require knowledge of external models or the existence of particular attributes in the monitoring data. While our focus remains unstructured monitoring data, we included selected approaches that use structured and transactional monitoring data to provide the reader with a broader context of the area and highlight limiting assumptions that related work makes.

Since discrete monitoring interfaces only provide a limited coverage of the operational state of the system a diagnosis that is only based on discrete monitoring interfaces may not be accurate. This limitation motivated us to investigate approaches that attempt to recover the system having only limited insight into the actual system state.

Finally, we summarize issues of related proposals that provide the motivation for our problem statement in the next chapter.

Chapter 4

Problem Definition

In this Chapter we provide a more precise problem definition and highlight our solution approach. We address the problem of complex software monitoring. Our primary objective is the use of discrete monitoring data obtained from existing monitoring interfaces to diagnose recurrent faults.

We assume that our approach is applied to a system that has already failed, or where failure is assumed to be very likely; as such, we do not address the problem of error detection.

As we already described in the background and related work sections, using existing discrete monitoring interfaces as samples are readily available. Furthermore, as our experience with IBM and research by other authors [45, 70, 143] has shown it enables the post-mortem diagnosis of faults that did not previously trigger error detection because the data was already generated previously.

Furthermore by not introducing any additional probes into the system, our approach can be oblivious to expert knowledge of the system implementation. By focusing on unstructured monitoring data our approach can be oblivious to the structure of the monitoring data.

Obviating the need to make assumptions about the system structure and the structure

of the monitoring data and the need to place additional probes in the monitored system, reduces the management burden on the operator of the system.

While reducing the management overhead of the operator, making such simplifications reduces the contextual information available to the operator when attempting to diagnose the cause of a system failure. We therefore focus on diagnosing faults that were shown to be highly recurrent [17]; in some instances up to 90 % of the reported problem cases were caused recurrent faults [89]. Using the lessons learned from resolving previous instances of recurrent faults provides detailed context to the resolution of a fault when it is diagnosed again.

Furthermore, as we described in the background section, data from discrete monitoring interfaces may only provide a limited state coverage of the system. As a result several faults may not be diagnosed from such monitoring interfaces, or semantically different faults may have identical manifestations in the monitoring data. We address this problem by proposing a probabilistic decision process that can be used to improve the diagnosis of such cases. The reader should note that this step is optional in our approach, and many cases do not need to be resolved in such a manner if the diagnosis is accurate.

In order to address the problem of recurrent-fault diagnosis our approach addresses the following sub-problems:

1. Modelling of discrete monitoring data,
2. Learning and diagnosing symptoms of recurrent faults from discrete monitoring data, and
3. Proactive recovery from perceptually aliased recurrent faults.

In the following sections we describe these problems and our solution approach to these problems in detail.

4.1 Modelling of Discrete Monitoring Data

The first problem that needs to be addressed in order to diagnose recurrent faults from discrete monitoring is that of modelling the discrete monitoring data for automated pro-

cessing. In order to model the monitoring data we leverage approaches from text mining that scale well in the face of huge amounts of discrete monitoring data.

Our principal focus is on analyzing unstructured discrete monitoring data such as log files. As we described in Section 3.1, the process of modelling log files entails the two conceptual steps:

1. Modelling samples of discrete monitoring data and
2. Modelling records of discrete monitoring data.

Since our objective is to diagnose symptoms of recurrent faults, we assume that the data for analysis is available as individual samples that contain sequences of monitoring records. Such samples can be log files that have been collected from faulty systems. Because these samples may vary substantially in size and time-span, we use windowing to select an area of interest within these sequences. Because we assume error detection that has a high confidence, we select such areas of interest as window of set time-span around the time the error was first detected. We refer to this time-span as window of interest and denote that parameter as w.

Furthermore, because our focus is on diagnosing recurrent faults we assume the existence of historic labelled training data. These labels can be derived from PMRs or other sources of data that provide context to the historic fault when it was first discovered. We want to emphasize at this point that we do not require individual records to be labelled, we only assume that we have a distinct label for each historic window if interest. These labels form the set of possible recurrent fault states $\mathbb{F}$ of the system that are a subset of the set of all possible system states $\mathbb{S}$.

Since our objective is to make further assumptions as to the internal structure of the system, we do not model the monitoring records within the windows of interest to be in sequence. Instead we use a frequency-based representation of the record types that expresses each window of interest as mixture of different monitoring record types.

Our primary focus is on analyzing unstructured monitoring data, therefore we need to infer the structure of the monitoring records automatically. Since we assume no particular structure of the monitoring records other than a timestamp and a plain-text portion,

we use text-mining approaches to approximate the set of possible record types as base strings (*see* Section 3.1). As a result we obtain a vector-space model of labelled historic windows of interest that consist of a mixture of possible base strings. This model is used to identify symptoms of recurrent faults based on the labels and the set of base strings.

4.2 Diagnosing Symptoms of Recurrent Faults from Discrete Monitoring Data

Having a vector-space representation of the samples of the historic monitoring data enables us to use classifiers to identify symptoms of recurrent faults inside the sample. In order to have a discrete representation of symptoms of recurrent faults we have chosen to use decision-tree classifiers. Using the C4.5 [106] classifier we can accurately associate sets of record types with particular recurrent faults.

We refer to the set of record types that maps to a particular recurrent fault as a symptom. The set of symptoms is denoted as $\mathbb{O}$. The reader should note that because of the limited state coverage of the discrete monitoring interface there may not be a bijective mapping of $\mathbb{O}$ to $\mathbb{F}$. As a result a symptom may identify multiple faults, faults may have symptoms that contain no record types (i.e., they have no manifestation in the historic samples), or symptoms of more different faults may have a non-empty intersection of record types. We refer to such issue as perceptual aliasing. When the association of historic faults is significantly aliased, other approaches are needed in order to perform fault diagnosis confidently.

4.3 Proactive Recovery of Perceptually Aliased Faults

As we described in the previous section, a core problem of analyzing monitoring data from one type of monitoring interfaces only is that the coverage of the system state may be

limited. As we described in the previous section this may lead to significant perceptual aliasing.

Related work [40, 55] in the area of system recovery appears to be oblivious to the possibility of diagnosing recurrent faults and instead only relies on error detection. We believe that although the mapping from the set of observations $\mathbb{O}$ (i.e., symptoms) to the set of recurrent faults $\mathbb{F}$ may not be ideal, establishing that mapping to reason about possible recovery actions is useful. As research by Oliner *et al.* [98] and our experience at IBM has shown, if faults are manifested as symptoms in discrete monitoring data they tend to have significant and very distinct manifestations.

We want to remind the reader at this point, that we believe that the semantics of faults are particular to the observer. Different users of the system have different objectives for evaluating the monitoring data. For example, during the development phase of a system the information of logs are primarily used by developers for debugging and testing purposes. The objectives of the developer include localizing sections of code that triggered particular core dump. As such it is in the interest of the developer to localize problems at the source code level. The semantics of faults for the developer are based on issues that are introduced at the level of the source code of the application.

On the other hand the operator of a system that is composed of mostly proprietary software components is usually oblivious to such monitoring information because she has no access to the source code. From her perspective the semantics of faults are defined from an operational perspective for example, which software components cause failures in the system or which hardware component has failed. The level of detail required for identifying software bugs is limited to identifying fix-packs that need to be installed to turn the system back into an operational state.

We want to emphasize that because the semantics of fault diagnosis and recovery are from the perspective of the observer, the set of faults $\mathbb{F}$ the set of system states $\mathbb{S}$, and consequently the set of observations $\mathbb{O}$ (i.e., symptoms), also depend on the observer. In order to minimize the overhead on different observers, we want to develop a model that can automatically infer symptoms of recurrent faults $\mathbb{O}$ and a mapping from symptoms $\mathbb{O}$

to faults $\mathbb{F}$ for different observers. Consequently, different observers may have different severities of perceptual aliasing. We see our solution to this problem using a probabilistic decision process as an optional tool that can be used by users of a system that experience a significant degree of perceptual aliasing to refine the diagnosis of recurrent faults made by a classifier that was trained against discrete monitoring data.

4.4 Model Assumptions

Our proposed solution operates within the following assumptions:

1. Prompt and reliable error detection is handled by an external process.
2. Non-recurrent faults are exceptional; hence the set $\mathbb{F}$ only describes highly recurrent faults.
3. Faults are mutually exclusive.
4. The set $\mathbb{F}$ is finite and known.
5. Active faults are persistent until explicitly recovered.
6. Every active fault can be recovered with a finite sequence of recovery actions.
7. Effects on the system state by recovery actions are stochastic.

The first assumption is a necessary restriction of scope for this thesis. Most distributed systems define a set of service-level objectives, to describe the normal operation of the system. By monitoring these service-level objectives correct system operation can be assessed.

The second assumption is based on the observation that many faults are highly recurrent. The third and fourth assumptions are needed to limit the scope of the diagnosis. For example, if the states describe individual components that have independently failed in a computer system, we assume that all those components that can fail are known beforehand. When fault diagnosis is limited to identifying failed system components, we also speak of fault localization.

The subsequent assumptions are used in Chapter 7. The fifth and sixth assumptions assure that we have full controllability of the system state while the system is failed. Full controllability can be attained by modelling human intervention as part of the recovery

actions. For example, catastrophic system failures can always be resolved by manually decommissioning the system itself. The seventh assumption is made because successful outcomes of recovery actions cannot be always guaranteed.

4.5 Solution Approach

Having described the core problems to address when diagnosing recurrent faults from discrete monitoring data, we provide an overview of our solution approach in this section. Our approach to diagnosing recurrent faults makes two main contributions. Our first contribution diagnoses failures from recurrent faults from log files. The second contribution refines a perceptually aliased diagnosis using proactive recover approaches. We show the stages of the first contribution in Figure 4.1 and the conceptual overview of the second contribution in Figure 4.2.

We apply the sequence of operations shown in Fig. 4.1 to approximate the base strings and, based on that, learn symptoms of recurrent faults from historic labelled log files. As we discussed in the previous sections labels can be defined from different observer perspectives and may include failed component, type of failure or recommended remedial action. The approach to log files has three phases.

In the first phase we model the structure of the individual samples of discrete monitoring data. The output of this phase is the set of approximated record types from the historic sample. We describe our approach to modelling samples of discrete monitoring data in Chapter 5.

Based on the learned record types and the labelled samples of historic faults we train a decision-tree classifier to predict faults in samples of discrete monitoring data in the second phase. Furthermore, because we use a model that represents the samples of discrete monitoring data as mixture of record types we can infer discrete symptoms of recurrent faults from the historic samples. Such symptoms model each fault as the presence or absence of particular monitoring record types over a fixed window of interest. We describe that approach in the first part of Chapter 6.

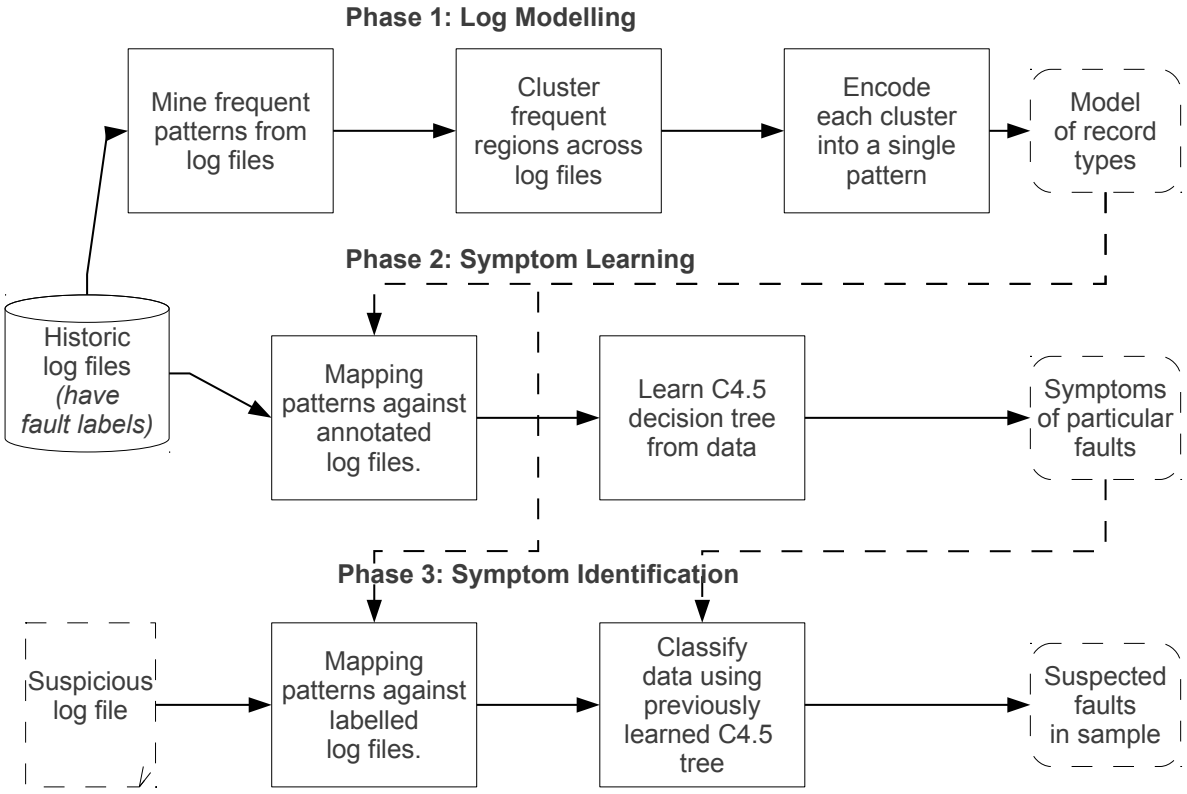


Figure 4.1: Overview: diagnosing recurrent faults from discrete monitoring data

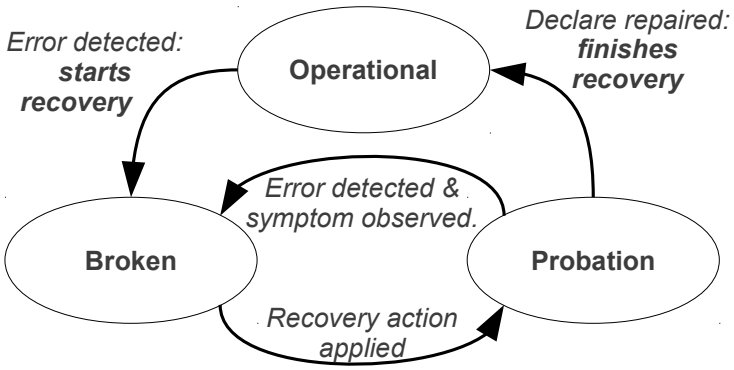


Figure 4.2: Overview: proactive system recovery

After the classifier is trained and a rule-base to identify symptoms of recurrent faults is established, we predict failures of recurrent faults in new samples using the classifier. We want to remind the reader that at this stage we assume the availability of reliable error detection such that the sample to be analyzed is coming from a system that has likely failed. We describe this approach in the second part of Chapter 6.

Since the state coverage of system that is described by the historic samples of the discrete monitoring data is limited certain fault classes may be aliased. In cases where this perceptual aliasing places a big burden on the system management, proactive approaches can be used to improve the diagnosis and recover from the failure. While automated recovery approaches have gained recent attention [40, 55], they do not incorporate symptoms of recurrent faults [109]. In order to improve a diagnosis made by the approach shown in Figure 4.1, we propose a probabilistic decision model based on Partially Observable Markov Decision Processes (POMDP). The overview of this approach is shown in Figure 4.2.

The approach entails error detection and proactive recovery. Error detection is used to start the recovery process and check if attempted recovery succeeded. In our work we assume the existence of an external reliable error detection mechanism that indicates whether the monitored ISS is working or not. When our ISS experiences an error, we transition it into the broken state. Whenever a recovery action is attempted, we transition

the ISS into the Probation state for a fixed period of time and use error detection to check if the recovery succeeded.

Upon initial error detection, we attempt fault diagnosis based on the approach described previously to validate the cause of the error. Whenever an attempted recovery fails (i.e., as indicated by the error detection), we rerun fault diagnosis to update our belief about the fault. This process is similar to hypothesis testing; given a hypothesis from the log analysis approach about a likely recurrent fault, the recovery approach validates or rejects the hypothesis. Upon rejection a new hypothesis is selected among the remaining candidates. Although so far we conceptually described our approach for the whole ISS scope, it is actually applied to individual serviceable components of the ISS. The granularity of the components depends on the granularity of the error detection and monitoring data. We describe this approach in detail in Chapter 7.

4.6 Summary

In this chapter we described the core problems our thesis addresses. Those problems are the modelling of unstructured discrete monitoring data, the diagnosis of recurrent faults based on that model, and the proactive recovery from perceptually aliased faults. Our solution approach consists of two contributions that address these problems. The proactive recovery can be seen as an optional step that can be applied in the face of significantly aliased faults.

The main contribution is our model of unstructured discrete monitoring data that we discuss in Chapter 5. We discuss recurrent fault diagnosis, without closing the feedback loop, in Chapter 6. We describe proactive approaches that integrate with the fault diagnosis in Chapter 7.

Chapter 5

Modelling of Monitoring Data

It is crucial for the automated analysis of log files to develop an appropriate system model. As log files record the state changes of a server over time [82], we use them to reason about the state of the system. Unfortunately, automatically reasoning about the system state is non-trivial and requires a model of the monitoring data to establish that reasoning. In this chapter we present the first phase of our approach. We show the conceptual overview in Figure 5.1.

In order to develop a model from the monitoring data, two main problems have to be addressed:

1. Modelling the occurrence of monitoring records over time and
2. Modelling the structure of monitoring records itself.

Addressing the first problem largely depends on the nature of the monitored system as well as the objective of the log analysis. As we described in Section 3.1, many approaches to error detection model the monitoring samples as a sequence of records over time. Many approaches to fault diagnosis model samples of monitoring data using different models that can express the mixture of monitoring records over a finite period of time. Such models, for example, include vector-space models [120] and association rules [45]. A key issue for this modelling step is to deal with the problem of time-skew. Timestamps associated with the individual log records may not reflect the true order of the occurrence of the events.

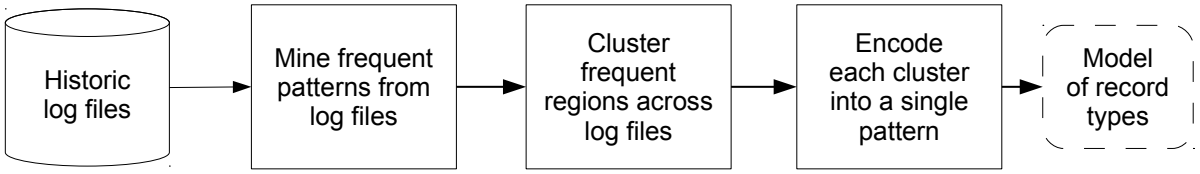


Figure 5.1: Modelling of Monitoring Data

We describe our approach to modelling the occurrence of monitoring records over time in Section 5.1.

Addressing the second problem depends on the nature of the monitoring data itself. The objective of this task is to associate individual records of the monitoring data with log record types. For structured monitoring data such types are sometimes directly logged by the system itself [53] and are typically defined during the development stage of application software. However, in most cases such type information is not directly available at the monitoring interface and needs to be approximated. The workflow shown in Figure 5.1 comprises our approach to approximate such types from unstructured monitoring data. We describe our approach to record type approximation in Section 5.2. Our primary focus is on modelling unstructured monitoring data.

5.1 Modelling Discrete Monitoring Data over Time

In this section we describe how we model discrete monitoring data over time. Our objective for this model is to obtain a representation of the records that can be used by classifiers as part of text mining that introduced in Section 2.4. As part of this model we structure a data set of discrete monitoring model into the following entities.

- Individual samples (e.g., instances of individual log files) and
- Record types that are contained in each sample.

Consistent with related work in text mining, we denote each sample in the data set as $\mathbf{D} \in \mathbb{D}$. Each instance of this set could be an instance of a log file.

Each record within the samples is of a specific type. Let $w \in \mathbb{W}$ denote elements of the alphabet of record types. For structured monitoring data such types are usually derived from special attributes contained inside the records and for unstructured monitoring data such types are typically modelled as base strings of the log records (*see* Section 3.1). The individual record types, $w \in \mathbb{W}$, need to be known apriori, specified, or approximated from the available samples. In the following sections we describe ways to approximate the record types from the available data.

In any case we assume the log records to consist of the following structure. Each element, $r = (t, w)$, is composed of a timestamp, $t \in \mathbb{T}$, and a record type, $w \in \mathbb{W}$. A sample of the set of all monitoring samples $\mathbb{D}$ is accurately represented by the following expression:

$$\mathbf{D} := \{r_1, r_2, \ldots, r_n\} \quad \forall \mathbf{D} \in \mathbb{D} \tag{5.1}$$

For example, for unstructured monitoring data we assume that the records are logged as a set of time-stamped base strings. Each of the samples $\mathbf{D} \in \mathbb{D}$ can be of variable length. The individual records may be subject to time skew; such that, the order of records does not correspond to the true order of occurrence.

In the next chapter we discuss transformations to extract the relevant record-types from this representation and use it for identifying symptoms of recurrent faults. The rest of this chapter describes the modelling of individual record types.

5.2 Record-Type Identification

After having described how we model samples of discrete monitoring data we describe our approach to the components of these samples in detail in this section. We model each sample as a set of time-stamped record types. Such types are written by software developers who may not have the overview of the operational context of the application once

it is deployed. The semantics of log records emitted during the operation of the application may be different from the development environment. For example, an event deemed critical at the development phase may only be deemed informational during operation. Also, an event deemed informational may indicate a serious problem during the operation of an application.

Furthermore, a software system today may be composed of many different components such that a software developer may be unable to attribute potential causes of failure at the time of development of an application. Therefore many applications seem to log the occurrence of an error rather than attributing the cause of it. In some cases causes of errors are logged that are useful for the software developer to locate potential software bugs (stack traces and core dumps). Such log records are, however, not useful to the operator since she usually does not have the means to debug the source code of the software. Since log files log the activity (state changes) of an application over time, minor changes may generate a substantial amount of logging data that needs to be analyzed. Neither the number of log records over time nor the distribution of log records may be sufficient to reason about the magnitude of the state change of the application [97, 98].

In addition the source code of the application may not be accessible or log normalization may change the format of the records, making it impossible to model all possible log records that could be emitted during operation upfront.

Structured and unstructured monitoring data expose timestamps. Unstructured monitoring data exposes only a single additional plain-text attribute and structured monitoring data exposes multiple attributes.

Attributes can be of different scales of measurement. We adopt Steven's model [131] to characterize scales. Attributes can be on a nominal, ordinal, interval, or ratio measurement scale. A special case, from our perspective, of a nominal scale is plain-text.

We model record types as a distribution over the values of a subset of available attributes of monitoring records. In order to derive distinct record types, the range of values of selected attributes needs to be quantized.

- Nominal attributes are implicitly quantized. Individual values are used directly.

```
Server myserver1 service XYZ down
Server myserver1 service ABC down
Server myserver1 service 123 down
```

Figure 5.2: Example log records

- Values of ordinal, interval and ratio attributes are mapped onto equidistant intervals.
- Plain-text attributes are clustered into classes of string similarity.

We assume that unstructured monitoring data primarily consists of records that have a timestamp attribute and only one additional plain-text attribute. We consider structured monitoring data that contains records that have an explicit type attribute or where the types can be easily inferred from the scales of measurement not as part of our evaluation. The focus of the next section is to describe how to approximate types from plain-text attributes.

5.3 Modelling Unstructured Monitoring Records

We model unstructured monitoring records as consisting of a base string and potential parameter values. The example log records in Figure 5.2 are instances that could be derived from the base string `Server %s service %s down`. We do not assume any knowledge of such base strings; instead, we approximate these base strings. This assumption is reasonable, because source code for proprietary software components is usually unavailable. Furthermore, many layered architectures have events passed through several layers of monitoring and potentially log them in a different form than that specified in the source code. Moreover, error handling may log unpredictable defects that cannot be modelled easily.

We approximate the base strings from the historic data. Sequences of these base strings are used to fingerprint the behaviour of each log file. We represent each approximated base string as the longest common subsequence of tokens from similar monitoring records.

Our approach to model the record types consists of several independent tasks that can be done in parallel. In Figure 5.3 we show the independent tasks of our approach.

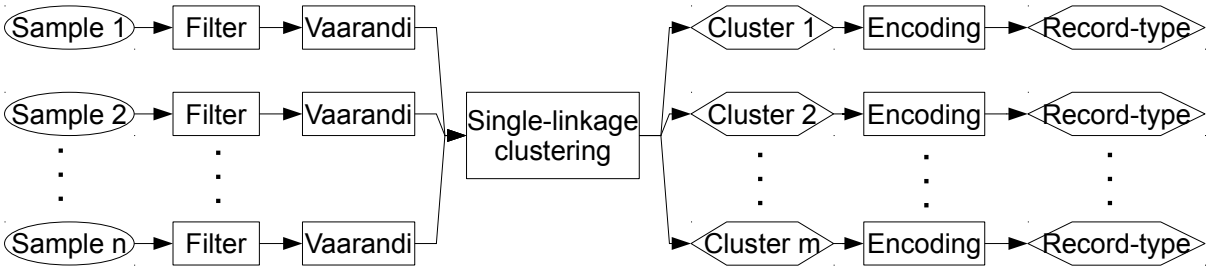


Figure 5.3: Parallel tasks of unstructured monitoring record modelling

The inputs to our model are samples of records of unstructured monitoring data. For example, a sample could correspond to a log file that is stored in a PMR database. At the first stage each sample is filtered independently. That filtering entails breaking the plain-text blocks up into tokens and filtering out tokens that are not of relevance to the record type modelling; for example, filtering out tokens that consist entirely of numeric characters or timestamps. The output of the filtering step is a set of token sequences, each sequence, contains the tokens of the corresponding plain-text record. The filtering tasks can be parallelised in two dimensions. The first dimension entails the parallel filtering of individual samples (e.g., log files). Each filtering task can be parallelised further to transform individual monitoring records into sequences of tokens.

The second step uses a density-based clustering algorithm to cluster similar sequences of the individual samples into regions. This task can be parallelised at the level of filtered samples. The outputs are so called regions of density. A region is a sequence of tokens that frequently occurs in the individual sample. We use a modified version of Vaarandi's algorithm [140] for this task and describe this step in detail in Section 5.3.1.

The third step entails a further refinement of the regions using single-linkage clustering. A big drawback of Vaarandi's algorithm is that it requires tokens of the sequences to be considered at absolute positions. This makes the algorithm very sensitive to the choice of delimiters that are used to break the unstructured records into tokens. The objective of this step is to aggregate similar regions into clusters. While there have been approaches

to parallelise single-linkage clustering for specific applications [99], we regard it outside of the scope of this thesis to evaluate the options in detail and regard it part of our future work. We describe this step in detail in Section 5.3.2.

The outcomes of the third step are clusters that contain sets of similar regions. Because a set representation of such regions may create additional overhead when matching it against discrete monitoring data, we encode each cluster as the longest-common subsequence of tokens of the contained regions. This step can be parallelised to the level of encoding individual clusters. We describe this step in detail in Section 5.3.3.

In addition to parallelism, the objective of this task structure is to chain tasks by increasing complexity. Each task reduces the input size of the following tasks to make our approach scalable to large data sets of discrete monitoring data.

5.3.1 Mining Frequent Regions of Monitoring Records

We modified Vaarandi's algorithm [140] to approximate the record types of individual log files. Vaarandi proposed a very effective density-based clustering algorithm that infers frequent patterns from individual log files, referred to as *regions*. A region is a set of tokens with their absolute positions in monitoring records. Its basic element, one token at a particular position, is referred to as a *one-region*. We modify the original algorithm using relative, instead of absolute, scales and heuristic approximations of some of its parameters.

The algorithm operates in two steps. In the first step it approximates frequent tokens in the data set. In the second step those frequent tokens are matched against the monitoring records, again to form regions. Of those regions the patterns are retained that occur above a set threshold. Unfortunately, the approximation of frequent tokens and the selection of regions to retain are parameterised, leaving it to the operator to specify these parameters on an absolute scale. Vaarandi, however, gives no heuristic or formal model for their specification. We supply heuristics in the remainder of this section.

The algorithm begins by mining frequent one-regions. Each monitoring record of the data is split into tokens according to a defined set of delimiters. The frequency of all tokens

of the data is measured to estimate frequent one-regions. Because measuring the frequency of all possible tokens is prohibitive, since the alphabet size is usually large compared to the frequency of tokens, Vaarandi uses a hash table of tokens to minimize the memory overhead. The number of buckets, k, is a user parameter. The non-empty buckets in that table are used to form the elements of the alphabet. From our perspective, the hash table should only grow on a logarithmic scale when a higher sensitivity s is needed. This sensitivity parameter controls how many similar instances of tokens are needed to establish a frequent token. Let b_s be the anticipated bucket size of the hash table. Based on examples [140] and our own observations, we propose the use of $b_s = 50$. The following equation shows how the number of bins for that hash table is approximated using p and b_s:

$$k = -\frac{b_s}{\log_{10}(s)} \tag{5.2}$$

Consider the log records in Figure 5.2 for illustration. The following one-regions are obtained by applying Vaarandi's algorithm, if the sensitivity is $s = 10\%$: (`Server`, 1), (`myserver1`, 2), (`service`, 3), (`down`, 5).

In the next step the input log file is processed again. Each log record is matched against the set of one-regions. All one-regions matching the log record are retained, forming a so-called *cluster candidate*. All cluster candidates that occur more than $s \times N_c$ times are retained, where N_c is the total number of cluster candidates. For the previous example this would result in the candidate: (`Server`, 1), (`myserver1`, 2) (`service`, 3), (`down`, 5).

Note that we apply Vaarandi's algorithm independently to each monitoring sample. The objective is to minimize the impact of large frequency shifts of individual records. Some samples may contain a large number of occurrences of error messages and be substantially larger than smaller files that only contain a few occurrences. Our objective is to reduce the input size to the following steps of the record type approximation as much as possible. For that step to be effective we have to establish a lower-bound limit for s. Since s corresponds, approximately, to the relative frequency of a particular record type to occur to be a region, s should have a lower-bound limit to not consider all individual plain-text instances as

independent regions. We propose the following heuristic.

$$\min(s) = \frac{1}{\text{median}\left(|\mathbf{D}_i| \quad \forall \mathbf{D}_i \in \mathbb{D}\right)} \tag{5.3}$$

The smallest value that can be chosen for p should be limited by the inverse of the median length of historic samples. Note that we use $|\mathbf{D}_i|$ to denote the length, as the number of monitoring records, for the monitoring sample $\mathbf{D}_i$.

The major advantages of this approach compared to existing work [34] are as follows:

- The absolute positions of the tokens are retained, making the application of string similarity measures against the regions more accurate.
- Region composition occurs in linear time with just three passes over the input data [1].

Fu *et al.* [34] use pair-wise string clustering using the weighted edit distance. Not only is the edit distance $O(l^2)$ with respect to the length of the input strings l, but nominal clustering incurs a complexity of $O(N^2)$ with respect to the number of data points N as well. Memon [86] proposed an approach that uses grammar inference techniques to infer a grammar for the log-records. It has been shown [39] that the problem of grammar inference is NP-hard in general. In order to apply grammar inference other preliminary steps need to be taken to minimize the input size. Such steps may introduce additional expert knowledge requirements that we want to avoid. In addition grammar-inference techniques have high data dependencies and are hard to scale in a parallel environment.

5.3.2 Clustering Frequent Regions

Transforming the monitoring records into regions reduces the number of data points. Unfortunately, we cannot use such regions directly as features because, as we have shown [111], the regions may still contain static parameters (e.g., hostnames, IP addresses) or parameters that occurred in more than $s \times N_c$ instances of particular log records. Applying

[1] Region composition is referred to as "clustering raw log keys" in Fu *et al.* and regions are referred to as "initial groups" by the authors.

Vaarandi's approach to individual log files avoids a bias towards extracting patterns from relatively long log files over smaller size log files.

We mitigate this problem by clustering the extracted patterns using a variant of the Levenshtein distance [72] as follows:

$$\mathbf{L}_n(x_1, x_2) = 1 - \frac{\mathbf{L}(x_1, x_2)}{\max(|x_1|, |x_2|)} \tag{5.4}$$

The function $\mathbf{L}(x_1, x_2)$ computes the edit distance in terms of different tokens between the regions x_1 and x_2. The notation $|x_1|$ represents the number of tokens in the region x_1. It allows quantifying the maximum allowable variation of two regions, x_1 and x_2, on a normalized scale. If a position in one region is not allocated in the other, we do not count this as a difference, favouring the most general region.

We cluster the regions using complete-linkage clustering with a threshold of t against the metric shown in Equation 5.4. We initialize the clustering by accepting all unique regions as individual clusters and then iteratively merge individual clusters until the pairwise distance to all elements in the cluster is less than or equal to t. The clustering threshold (t) can be defined automatically using the approach proposed by Fu *et al.* [34]. For records sharing the same base string, the inner class distance is usually small compared to different base strings. We can apply 2-means clustering to the data set. The distance between the two groups would approximate the inter-class distance. The outcome of this step is clusters of similar regions.

5.3.3 Encoding Region Clusters as Patterns

Although using the region clusters as features yields high classification accuracy for recurrent faults [111], maintaining the clusters of regions as record types is impractical. We encode the clusters as patterns that enable efficient matching against log files and ease the work of human investigators, because they only have to review individual patterns instead of region clusters.

Assuming all regions within individual clusters are derived from the same base string, they share a longest common subsequence (LCS) of tokens (*see* Section 2.4.3). In our view the LCS is a compact representation of strings that can also be easily converted to POSIX regular expressions to provide backward compatibility with existing rule-based log file analysis tools.

Through the use of our previous clustering step we ensure that the strings are similar and vary by only a few parameters depending on the clustering threshold (t) of the previous step. Since the regions within one cluster vary only by a few static parameters we expect the alphabet size to be approximately proportional to maximum length of the strings inside the cluster.

We adopt the proposal by Melchiar *et al.* [85] to create an LCS for each cluster (*see* Section 2.4.3). Our application of Melchiars approach for each cluster is based on the following three steps:

- For each region inside the cluster construct a DASG,
- Intersect all DASGs to create a CSA that accepts all sequences of tokens common to the regions of the cluster and
- Find the longest path using Dijkstra's algorithm to obtain a LCS solution.

The output of this step is a LCS of tokens for each cluster of the previous step. The approach is consistent with the example shown in Figure 2.6.

The decision to use this algorithm instead of traditional approaches like the Wagner-Fisher algorithm [142] is due to the better memory and time-complexity of the proposal by Melchiar *et al.* [85]. The reader is reminded of the issue that we are operating against many regions inside the individual clusters. Typical application areas of the traditional algorithm only consider binary comparisons (i.e., two strings), for example `diff` [33] and are as such perceived to be of quadratic complexity. In other words, the problem is bound to at most two stings, while in our case the problem is bound by the number of regions inside a cluster. Applying the general form of Wagner-Fischer [142] results in the complexity that is shown

below.

$$O\left(m\prod_{i=1}^{m}|S_i|\right)$$

The parameter m identifies the size of the cluster (i.e., number of regions). The notation $|S_i|$ denotes the length of each region (i.e., number of tokens). Assuming the length of all regions is equal the Wagner-Fischer solution has an exponential running time with respect to that length only. Other parameters such as the alphabet size (i.e., the set of unique tokens for each cluster) or the similarity of the regions have no influence on this algorithm. As such the algorithm in general may yield prohibitive computational overhead if the cluster consists of a large number of sequences.

By comparison, the running time of the proposal by Melchiar depends on other parameters too. Based on Equation 2.1, these properties lead to an efficient running-time of Melchiar's algorithm. Given a cluster that has an alphabet A of tokens and assuming the optimal case, where the cluster comprises m equal regions S that have no repeated tokens (i.e., $|A| = |S|$), the time complexity of the algorithm will be proportional to $O(|S|(m|S| + m)) = O(m|S|^2)$. Equation 2.1 shows that the algorithm scales geometrically with an increased alphabet size. Our observation indicates that individual log records are relatively short and their base strings contain only a few or no repetitions of tokens, so this is not a limiting factor for our approach. We also want to highlight that we process tokens instead of individual characters, thus the input size to the algorithm is kept small. Furthermore, the quantities that influence the running time can be measured directly to avoid extreme cases or to apply approximate algorithms (*see* [46, 57, 63]).

The outcome of this step is one common subsequence of tokens for each cluster. Depending on the nature of the input data, more than one cluster can have a common subsequence and some clusters do not have a common subsequence. We only retain unique subsequences of a length greater than one. This set now establishes the set of approximated record types $\mathbb{W}$. We use the notation $\mathbb{W}$ for the set of record types that are semantically equivalent with the set of words in text-mining problems.

```
Server myserver1 service XYZ down
Server myserver2 service ABC down
Server myserver3 service 123 down
Err Server myserver4 service 1 down
Err Server myserver5 service 2 down
```

Figure 5.4: Example plain-text messages

```
{(Server, 1), (Err, 1), (Server, 2), (myserver1, 2), (myserver2, 2),
(myserver3, 2), (myserver4, 3), (myserver5, 3), (service, 3),
(service, 4), (XYZ, 4), (ABC, 4), (123, 4), (1, 5), (2, 5),
(down, 5), (down, 6)}
```

Figure 5.5: Example one-regions

5.4 Example of Record Approximation

In this section we show an end-to-end example of the application of our approach to record approximation. Suppose we obtained the plain-text messages shown in Figure 5.4. In the first step of the record approximation we use Vaarandi's algorithm to mine the regions of that sample. In the first stage of Vaarandi's algorithm we would obtain the one-regions shown in Figure 5.5. Based on the configuration of s rare items are discarded from this set. Without the loss of generality let us assume that only the one-regions shown in Figure 5.6 remain.

In the next stage Vaarandi's algorithm matches the set shown in Figure 5.6 against the unstructured monitoring records in order to form the regions. Based on the configuration of s frequent regions are retained. Based on the example this would yield the regions shown in Figure 5.7. Note that `*` is a placeholder for an unmatched token.

As shown in Figure 5.7, the use of absolute positions to express the density of tokens makes the approach very sensitive to the choice of delimiters and preliminary filters. Since our objective is to provide a robust solution, we cluster the outcomes of Vaarandi's algorithm again using a linkage-based clusterer that uses the Levenshtein distance to consider

```
{(Server, 1), (Err, 1), (Server, 2),
(service, 3), (service, 4),
(down, 5), (down, 6)}
```

Figure 5.6: Filtered example one-regions

```
Server, *, service, *, down
Err, Server, *, service, *, down
```

Figure 5.7: Filtered example regions

similarities with respect to shifting positions as well. As a result all elements of Figure 5.7 are clustered in a single cluster, based on the configuration of the parameter t. While in this example we only have to consider two elements inside this cluster, in general there may be many more elements inside these clusters. Therefore we use the LCS algorithm proposed by Melchiar to encode each cluster as a sequence of tokens. For the sequences shown in Figure 5.7 that would result in the LCS: `Server, service, down`.

5.5 Summary

In this section we have shown our approach to building models directly from discrete monitoring data. We provided an efficient and low-overhead solution to modelling unstructured monitoring records that only expose a plain-text attribute and a timestamp. We presented a method to establish types of records as longest common subsequences. This representation offers an intuitive and compact way to model features of a log file that can be easily understood by support operators. Furthermore, this model enables computing measures to estimate the overhead of the feature extraction. These routines can be used adaptively to enable approximate algorithms in place of the proposed exact algorithm if the overhead is deemed prohibitive. We are proposing a compact feature representation of log files of ISSes. Furthermore, this approach can be applied to any plain-text log file without relying on the presence of a rich set of additional attributes in the record types. Finally, we have

described that the individual tasks of the modelling process expose only a small amount of data dependencies, such that most of the tasks of the modelling process can be implemented in parallel to improve the scalability of the approach. We are not aware of other solutions to the problem of modelling unstructured monitoring data that expose such a high degree of task parallelism.

In the next chapter we will show how to use this model for extracting symptoms of recurrent faults from collections of labelled log files.

Chapter 6

Extracting Symptoms of Recurrent Faults

In this chapter we describe our approach to symptom extraction. The approach uses text mining and is based on the data model described in the previous chapter. The conceptual workflow of this part of our approach is shown in Figure 6.1.

The approach entails two phases: learning and observation. The inputs to the learning phase are the approximated record types that we discussed in the previous chapter. In addition, we need a set of labelled samples of discrete monitoring data that we can use for the supervised learning of faults. The historic samples only contain monitoring records and a label per sample that identifies the recurrent faults. Our approach does not rely on any further annotation of the samples. These samples are converted into a vector-space model using the model of the records described in the previous chapter. This vector-space model is then used for supervised learning. The output of the learning stage is a trained decision tree classifier that can classify samples where a fault is highly suspected. We describe the vector-space model in detail in Section 6.1 and the learning of the symptoms in Section 6.2.

In the second phase this classifier is used to classify monitoring samples of the system where a failure is highly suspected. In this phase we convert the sample into a vector-space representation using the model that was described in the previous chapter. This vector

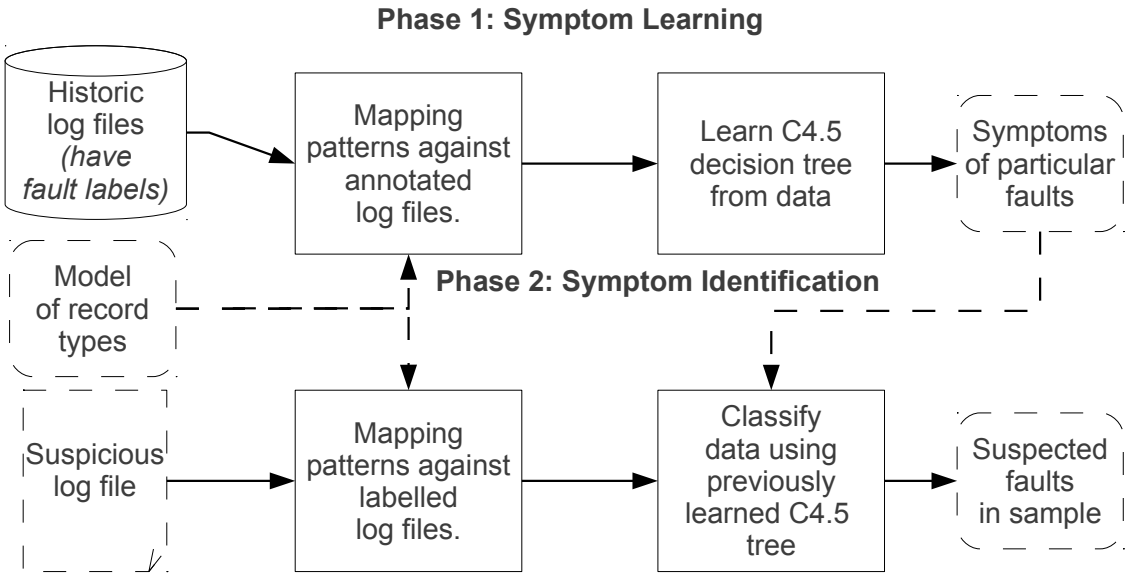


Figure 6.1: Learning and identifying symptoms

is classified by the trained classifier to attribute likely recurrent faults. We describe this phase in Chapter 6.3.

6.1 Vector-Space Representation of Samples

In the previous chapter we introduced a model of discrete monitoring data that expresses each sample, $\mathbf{D} \in \mathbb{D}$, as a set of discrete monitoring record. Each monitoring record $r = (t, w)$ consists of a timestamp and a record type $w \in \mathbb{W}$. The previous chapter was concerned with the establishment of $\mathbb{W}$ and in this section we describe how to transform this model into a representation that can be fed into a classifier.

As we described previously, the samples, $\mathbf{D} \in \mathbb{D}$, can vary in size and duration. Moreover, the period where the fault was active may differ from the period of the sample. From our experience the duration of monitoring data that can be collected when a fault is suspected in the system largely exceeds the duration of many active faults. In addition, the timestamps that are associated with the individual log records may not accurately reflect

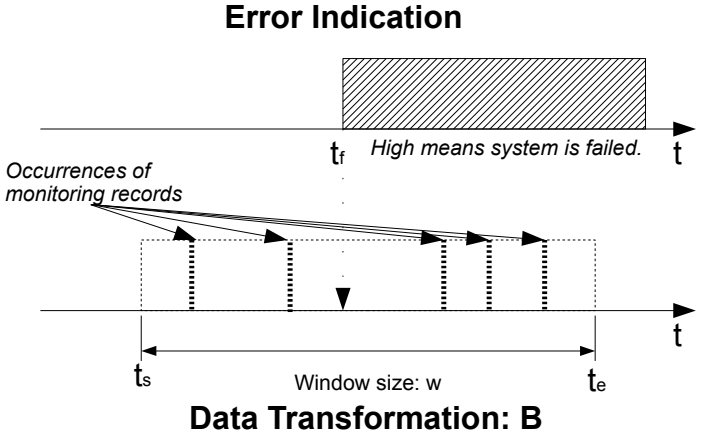


Figure 6.2: Selecting records of interest

the time at which the logged event occurred. In addition, the timestamps may not accurately identify the order of the events. Without the use of special attributes, such as a sequence counter, the order cannot be assumed to be accurate in general. We previously described this issue as time skew. Our design objective is to find a robust model that is oblivious to this problem and selects a reasonable period of monitoring data to learn and identify the symptom of the fault. We denote this model as the function $\mathbf{B}$ that applies a data transformation to each sample $\mathbf{D} \in \mathbb{D}$ and show the transformation in Figure 6.2.

We assume the existence of reliable error detection. The error can be flagged as being active by an indicator function over time that we show in Figure 6.2. The monitoring records that are of interest to the symptom learning are those that immediately precede or succeed the system failure (i.e., error indication becomes high). Since ISSes comprise many components, it may take some time until the error of an active fault is propagated to the system's service interface. As such, the records preceding the error indication are as important as the ones succeeding the indication. For that selection we define a window of interest of w seconds that spans across the indicated time of failure t_f. Unless stated otherwise the window is symmetric around t_f such that $w = 2(t_f - t_s) = 2(t_e - t_f)$. Because the records inside the window are selected based on their timestamp the window size needs to be large enough to minimize the effect of time skew. Depending on the system usage

and assumed system reliability the window size may vary. For example for discovering actionable events in network management related work [45] have chosen w on the order of minutes; related work [97, 129] in supercomputers have chosen w on the order of hours to investigate potential faults manually.

In order to be oblivious to the event order, we only consider the types of records, $w \in \mathbb{W}$, that occurred in this time period, instead of the events itself. If a record type occurs multiple times it is only counted once. Counting the type only once has the objective to treat faults that are only indicated by a few record types equal to faults that are manifested as bursts of monitoring records in the data. In addition, this model is less prone to overfitting to workload shifts that could vary the frequency of individual record types within w for different instances of the same recurrent fault. The reader should note that because of the preliminary filtering, the use of Vaarandi's algorithm and the LCS encoding, the set of record types may be relatively small and not every record in the original monitoring sample will be associated with a record type.

$$\mathrm{B}(\mathbf{D}) = \langle \ldots, w_i \in \mathbf{D}, \ldots \rangle \tag{6.1}$$

In summary, $\mathbf{B}(\mathbf{D})$ creates a vector of bits of all record types in $\mathbb{W}$ for each sample. The bit is set to true when the record type has been observed in the sample over the period w the bit is set to false when the record type did not occur. We denote this indicator function as shown in Equation 6.1. Samples of the training set also have a corresponding fault label $f \in \mathbb{F}$. Using the indicator function $\mathbf{B}$ the samples of the training set are converted into a set of bit-vectors that form the vector-space model. Each sample that is to be classified in the second phase is also converted into a bit-vector first.

6.2 Symptom Learning

During the learning phase we take vector-space model that was described in the previous section and train a classifier to predict fault labels of new samples. That vector-space classification can be used to train various types of classifiers. While probabilistic classifiers, such

Sample	**Record type**			**Fault**
	w_1	w_2	w_3	
$\mathbf{B(D_1)}$	X		X	f_1
$\mathbf{B(D_2)}$			X	f_2
$\mathbf{B(D_3)}$	X			f_3
$\mathbf{B(D_4)}$		X		f_4

Table 6.1: Example vector-space representation of fault sample

as Naïve Bayes, can be used to produce a ranking of likely faults, we focus our evaluation of our approach on decision-tree classifiers and use C4.5 [106] in particular (*see* Section 2.4.3).

As we described previously, decision-tree classifiers generate tests for individual attributes of the training set. In our case the attributes correspond to the presence or absence of particular record types over w. Because these attributes are binary the test will only consist of the presence or absence of a particular record type. These tests form the inner nodes of the learned decision tree and the leaf nodes correspond to the fault labels.

This representation is attractive because identification rules for particular faults can be relatively easy inferred from the tree. Each path from a leaf node to the root corresponds to a particular symptom. A symptom of a particular recurrent fault thus corresponds to a set of record-types that need to occur or be absent within the period w to identify the fault. Each path corresponds to a symptom for the fault label and maps to an element in the set of symptoms $\mathbb{O}$.

Let us illustrate the approach with an example. We show the vector-space model of a data set that consists of four samples, four faults and three different record types in Table 6.1.

Each of the samples of the input set has been converted to into its vector representation. For the record types an X denotes the containment of the record type within w for that particular sample. It is absent otherwise. The set of recurrent faults consists of four faults $\mathbb{F} = \{f_1, f_2, f_3, f_4\}$. A decision tree that describes the data is shown in Figure 6.3.

The fault labels form the leafs of the decision tree and the test for the presence of

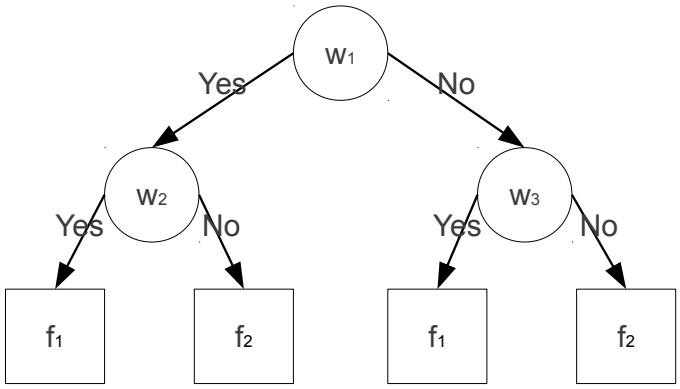


Figure 6.3: Example of a decision tree

particular attributes form the inner nodes. The path from each leaf to the root expresses a symptom for that particular fault. For example Fault f_1 is identified by the occurrence of w_3 and w_1 over the period w in a sample.

As we described in Section 2.4.4, C4.5 establishes the structure of decision trees recursively according to the information gain of particular tests. It also contains several heuristics, such as reduced-error pruning to minimize over-fitting. In our experience samples of discrete monitoring data are characterized by a larger number of record types. Figure 6.3 only serves as a conceptual illustration.

In addition to training a C4.5 decision-tree classifier that produces discrete symptoms, one can also train probabilistic classifiers such as the Naïve Bayes classifier [144]. The trained classifier assigns likelihood to each fault label for the sample. The label of maximum likelihood can be used to flag a recurrent fault in the sample. It has been shown [48] that Naïve Bayes and C4.5 have a similar classification performance. As such we do not consider Naïve Bayes as part of our evaluation.

6.3 Symptom Identification

Once a failure is validated by external error detection, we seek to identify recurrent faults that seem associated with the observed failure. If the fault that triggered a failure is recurrent, we may be able to fast-track resolution by retrieving information about past actions to restore the failed component.

Having the trained decision-tree classifier that we described in the previous section, a fault can be quickly identified from the vector-space representation of the new sample. We want to remind the reader that many popular log-reconciliation tools such as Logsurfer [136] and IBM Log and Trace Analyzer [145] allow for the incorporation of identification rules. For example, Logsurfer accepts rules consisting of Regular Expressions [100] to attribute potential problems in the log data. The rules that can be inferred from the decision tree can easily be back into these tools by formulating features (i.e., LCS representation that was described in Chapter 5) as Regular Expression.

Furthermore, if a probabilistic classifier such as Naïve Bayes is trained the recurrent faults can be assigned as ranking of likely faults. We generate a ranking based on the likelihood that the sample $\mathbf{D}$ belongs to a particular fault. When applying Naïve Bayes the set of recurrent faults $\mathbb{F}$ needs to be mapped to $\mathbb{C}$ and the vector-space model of the record-types needs to be mapped to $\mathbb{V}$ (*see* Section 2.4.4). We report the suspected faults in a list ordered by rank and include the likelihood of each suspected fault as an anomaly score. This enables a human investigator to quickly validate the diagnosis or find out if the sample was falsely attributed.

In the absence of probe and recovery action this model may provide useful information about the alleged fault. During prototype demonstration being able to rank likely recurrent faults in the data set was perceived positively by a professional audience.

6.4 Summary

In a practical problem-determination scenario a human investigator usually has access to log files from the system in a healthy state. Knowing which elements of the corpus are only present in the healthy state greatly improves the accuracy and sensitivity of machine learning approaches over the case where only features from the abnormal states are used. In this work we focus on failures that have a clear manifestation in monitoring samples rather than failures that do not generate records at all. In particular, failures that are performance-related can be detected and diagnosed by other means as we have shown in our prior work [91].

In this chapter we have described methods to learn fault symptoms from log files using semi-supervised learning. Specifically, we used a Naïve Bayes model to classify faults and we employed decision trees to learn discrete symptoms to identify faults. We have assumed that the log files are processed such that they only contain data for the time window wherein a suspected failure occurred. In practice, log files vary in size and time-span covered. The files may also contain manifestations of multiple independent faults. It is therefore imperative to limit the scope of the analysis. If service identification is possible in the monitoring data, the health of services can be obtained on an individual basis. Limiting the time window of the diagnosis limits the likelihood of multiple independent faults as well.

The methods presented in this chapter can be integrated into the tooling employed by support personnel to perform root-cause analysis. We do not attempt to fully automate root-cause analysis; some aspects of this process are still manual. For instance, a support operator still needs to decide upon a labelling scheme for the purpose of classification. It is possible to present the operator with an interactive rule-generation workbench, which suggests labelling schemes that will improve results. Such a feature would guide a human operator in choosing a scheme that minimizes ambiguity and yet aids diagnosis by assigning meaningful fault labels.

Chapter 7

Proactive Fault Diagnosis and Recovery

A challenge when diagnosing the cause of a failure is the lack of operational context. Diagnosing recurrent faults as described in Chapter 6 may cause perceptual aliasing (*see* Section 4.2). Different faults can have identical manifestations or faults may have no manifestation in log data at all.

The approach we discussed so far diagnoses the problem by evaluating log data. We did not use additional probes to evaluate the state of the system. In this chapter we extend the approach to mitigate perceptual aliasing. In addition to probes that should not change the system state, this model also allows self-recovery by including actions that change the system state. We construct our model on top of a Partially Observable Markov Decision Process (*see* Section 2.5). A POMDP allows decision making under uncertainty. In our case the diagnosis of a fault can be uncertain because its manifestation is perceptually aliased. In order to integrate probing, diagnosis and recovery, we need to expand the definition of a fault. In our model a fault is associated with a particular recovery action. In this context, we do not seek to provide a diagnosis for non-recoverable problems.

The approach discussed in this chapter should be seen as optional to the steps discussed so far. In order for automated recovery to be applicable, very limiting assumptions need to be made about the monitored system. While the approach to diagnosing symptoms of recurrent faults discussed up to now may suit many different application areas, the

extension discussed in this chapter may only be applicable to a sub set of these systems. In Section 7.1 we introduce extensions to our system model and provide an example of the approach in Section 7.2.

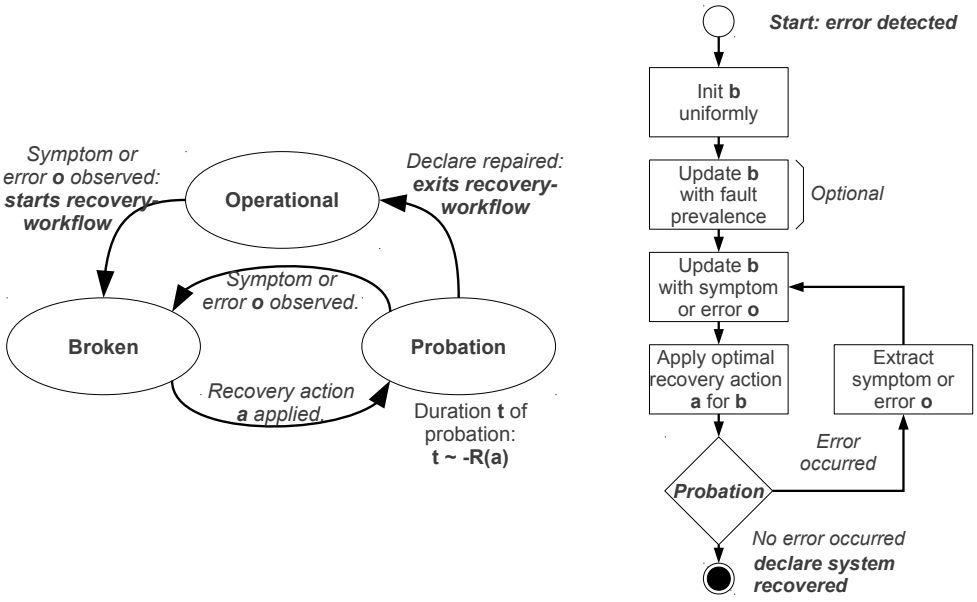
7.1 System Model

In Figure 7.1a we show the basic modes of our proposed controller. We model the system to be in three modes of operation. It can be operational, broken or in probation. A system is assumed to be running in the operational mode. If an error is detected while the system is operational, optional fault diagnosis mechanisms are invoked, the controller enters the broken mode and the recovery starts (*see* Figure 7.1b). The recovery spans the modes broken, and probation. In the broken mode we select a recovery action with respect to cost and, optionally, the observed manifestation of a failure (i.e., symptom). The recovery action is applied and the system enters the probation mode for an amount of time that is specified for individual recovery actions. The system remains in probation until it is declared recovered.

The set of controller modes bear similarity to Isard [55] and Goldszmidt [40]. Our proposal differs in the workflow (*see* Figure 7.1b) for choosing an optimal recovery action and the specification of recovery actions, probation, optional symptoms, and expected faults. In the following sections we describe how we specify these artefacts and map them to POMDP parameters. The mapping of model parameters to POMDP artefacts is shown in Table 7.1. In the next sections we describe these mappings in detail and, furthermore, describe the termination (i.e., how the system is declared recovered) for the controller in Section 7.1.6 and how optimal control is achieved with this mapping in Section 7.1.5.

7.1.1 Modelling Faults

In our model a fault is an unobservable system state that causes the system to fail. We assume the set of faults to be finite and fixed (but not necessarily known). We furthermore



(a) Controller modes (b) Recovery workflow

Figure 7.1: Controller overview

POMDP Model Artefact	**Model Parameter Without Symptoms**	**Model Parameter With Symptoms**
Initial Belief State b_0	Uniform distribution	Initialized from fault symptom and, optionally, fault prevalence
Set of hidden system states $\mathbb{S}$	One state per recovery action and fault-free state $f_\emptyset$	One state per known fault and fault-free state $f_\emptyset$
Set of observations $\mathbb{O}$	A state for error observed o_e and a state for no error observed $o_\emptyset$	A state for each possible symptom and a state for no error observed $o_\emptyset$
Set of actions $\mathbb{A}$	An action for each recovery and probe action	same
Transition function $\mathbf{T}$	Expected success rate of recovery action, assuming the system is broken in a way, such that, the particular recovery action is applicable	Expected success rate of recovery action for given faults
Observation function $\mathbf{\Omega}$	Deterministic observation if a recovery action succeeded $o_\emptyset$ or failed o_e	Observation probabilities for symptoms of known faults and successful recovery $o_\emptyset$

Table 7.1: Mapping of POMDP parameters to self-recovery parameters

assume that we have complete control over the states, such that we can always recover from any fault (*see* Chapter 4). This assumption can be implemented in an actual recovery controller by incorporating a super recovery action (for example calling a system administrator) that at least subsumes all faults that cannot be handled by other recovery actions of the model. Under these assumptions, we consider two modelling avenues: First, the set of potential faults that a system can experience is unknown. Second, the set of potential faults is known. The first avenue suits distributed ISSes, in which potential faults are not known. Since we assume that we have complete control, although the individual faults are unknown, we assign a fault state $f_i \in \mathbb{S}$ for each recovery action $a_i \in \mathbb{A}$. In other words we assume the existence of a potential fault, for which a specified recovery action is applicable. The second avenue suits systems where the set of faults is known. In that case the set of potential faults $f \in \mathbb{S}$ is directly included in the POMDP set of states. For example consider a multi-tier system, in which individual components can only crash. The proposed recovery actions are individual component reboots. Note that since our model can handle partial observability only a limited number of components need to be monitored to reason about which component to reboot. The recovery actions should consider potential component dependencies. As we described previously the POMDP approach needs an accurate system model (i.e., effects of actions) but can operate under partial observability (i.e., limited state coverage by monitoring). From our perspective production systems that suit this requirement are usually weak-coupled distributed systems comprising stateless components.

In Section 7.1.4 we describe how to incorporate symptoms of recurrent faults and fault prevalence in detail.

7.1.2 Modelling Recovery and Probe Actions

In our model actions are specified by a conditional success probability, a set of conditional observations and their cost. We distinguish probe and recovery actions.

Recovery actions directly influence the system state. We specify a conditional probability of success p_i for each hidden fault state $f_i \in \mathbb{F}$ for which the recovery action $a_i \in \mathbb{A}$ is

applicable. This probability is directly included in the transition function $\mathbf{T}(f_\emptyset|f_i, a_i) := p_i$. Assuming the recovery action has no side-effects (i.e., causing other faults), the probability of failure $(1 - p_i)$ is assigned for $\mathbf{T}(f_i|f_i, a_i) := (1 - p_i)$. If we presume independent recovery actions, for all other states $f \in (\mathbb{S} - \{f_i\})$ the action a_i has no effect, such that $\mathbf{T}(f|f, a_i) := 1$. In Section 7.1.3 we relax this assumption to model recovery actions that subsume other recovery actions and are applicable to multiple faults. After the recovery action has been committed we transition the system to probation mode and monitor the system for errors (in the case of unknown faults) or symptoms of particular faults. The conditional probabilities for the anticipated observations are directly modelled in the POMDP observation function. For now, we only model the observations error o_e, the empty observation $o_\emptyset$ (*see* Section 7.1.6). We model their occurrence directly in the POMDP observation function, such that:

$$\begin{aligned} \mathbf{\Omega}(o_\emptyset|f_i, a_i) &> \mathbf{\Omega}(o_e|f_i, a_i) \\ \mathbf{\Omega}(o_e|f, a_i) &= 1 \quad \forall f \in (\mathbb{S} - \{f_i\}) \end{aligned}$$

The specified cost (i.e., negative reward) of the recovery action is proportional to the anticipated duration of the probation period, such that the longer the duration of the anticipated probation period, the more expensive the action becomes.

A probe action p does not change the current system state but results in an observation that depends on the current system state, such that $\mathbf{T}(f|f, p) := 1 \quad \forall f \in \mathbb{S}$. We include probe actions in the set of POMDP actions $\mathbb{A}$. For each probe $p_i \in \mathbb{A}$ we define one observation, o_{if}, that represents the failure of probe $p_i \in \mathbb{A}$ and at least one observation, $o_{is1}, \ldots, o_{isn}$, that correspond to successful probe outcomes. These are the only observations to be made after a probe has been issued; such that:

$$\sum_{o \in \{o_{if}, o_{is1}, \ldots, o_{isn}\}} \mathbf{\Omega}(o|f, p_i) = 1 \quad \forall f \in \mathbb{S}$$

In other words after a probe has been issued, we only monitor the outcomes of the probe action in the probation mode. The cost is modelled similarly to recovery actions.

7.1.3 Subsumed Recovery Actions

In the previous section we only considered recovery actions that were applicable for one individual fault. In this section we consider the special case that recovery actions are applicable to multiple faults. Since the observations and the state transitions are expressed as conditional probabilities over the hidden states in POMDPs, the actions that apply to multiple faults can be easily modelled by simply specifying the conditional probabilities of state transitions for each fault and an action. Likewise, the expected observations can be specified. In general, to apply an action $a \in \mathbb{A}$ to a subset of fault states $f_1, f_2, \ldots, f_m \in \mathbb{S}$ the assumed success rates for each covered fault need to be specified as:

$$\mathbf{T}(f_\emptyset | f_i, a) \forall f_i \in \{f_1, f_2, \ldots, f_m\}$$

7.1.4 Incorporating Symptoms of Known Faults

Symptoms are obtained from the fault diagnosis that is performed after the initial error detection and before the recovery workflow executes. This fault diagnosis is executed again with error detection in the probation mode. In order to incorporate symptoms of recurrent faults, the model introduced in Section 7.1.2 needs to be extended. Instead of modelling an error observation, $o_e \in \mathbb{O}$, and an empty observation $o_\emptyset$, we model each symptom as an individual observation, $o \in \mathbb{O}$, in addition to the empty observation $o_\emptyset$, such that the set of observations for the underlying POMDP is $\mathbb{O} = \{o_1, o_2, \ldots, o_\emptyset\}$. In addition, the observation probabilities need to be adjusted to reflect the expected symptoms for individual faults in $\mathbf{\Omega}$. Such observation probabilities can be obtained by analyzing historic monitoring data, as we have shown in previous work [60, 112, 113], or by incorporating expert knowledge. The reader should note that each action only generates one observation at a time, but individual faults can be identified by multiple symptoms. Furthermore, one symptom can map to multiple faults. The association of a particular symptom o to a particular fault is done by setting $\mathbf{\Omega}(o|f)$ to a non-zero value.

The initial belief state $\mathbf{b}$ of a POMDP represents the distribution of the state the system is in. We can use the specification of $\mathbf{\Omega}$ directly to initialize the belief state of the POMDP

to select the optimal recovery action for the diagnosis $\mathbf{b}$, as follows:

$$\mathbf{b} = \{\mathbf{\Omega}(o|f_1), \mathbf{\Omega}(o|f_2), \ldots\}$$

The reader should note that this initialization also holds for symptoms of ambiguous faults. So if an extracted symptom identifies a set of faults this will directly map into the initialization of the belief state.

7.1.5 Achieving Optimal Control

A POMDP is a discrete-control paradigm. In each control step the POMDP selects an optimal recovery action based on the belief state, then applies the recovery action and waits for the observation emitted by controlled system. In our case, the observations are received by error detection or fault diagnosis that is performed in the probation mode. Once an observation is made the observation is used to update the belief state and to select the next recovery action. In order to select the optimal recovery action, a POMDP maximizes the expected rewards received (e.g., negative cost) over an infinite horizon, as shown by the term in Equation 7.1. The variables r_t describe the rewards received at each time step t that arise from the specification of $\mathbf{R}$ [69].

$$\mathrm{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right] \tag{7.1}$$

Although our recovery model is designed to terminate after a finite number of steps, using the POMDP model for the finite horizon problem is undesirable because the steps to recovery, k, are not known in advance. We therefore chose to maximize the rewards over an infinite horizon to satisfy Equation 7.1. In order to make the sum finite, POMDP models include a discount factor, $0 \leq \gamma < 1$. The discount factor can be interpreted as probability for termination; such that a decision to terminate the run is made at each step with probability $1 - \gamma$. The larger the value of γ the more importance is put on future rewards. A low value makes the agent opportunistic (it would only consider immediate

rewards) [69].

In the context of self-recovery the value of γ should be chosen with respect to the number of estimated recovery actions to be applied. If, for example, fault diagnosis is very accurate; such that particular observations occur with a high probability for particular faults, γ should be chosen closer to 0. If, on the other hand, the number of recovery actions is extensive and the observability of the system is very limited, γ should be chosen closer to 1. This is the case when the system has many failure modes that result in ambiguous symptoms [113] (i.e., one observation for many fault types).

Value Function and Policies

The behaviour of the agent is referred to as policy π. A policy is a mapping from a belief state to an action, $\pi : \mathbf{b} \rightarrow \mathbb{A}$. The expected total rewards received by applying a policy are referred to as value of the policy [69]. In order to achieve optimal control, we want to derive an optimal policy for every initial belief state of the controller. It was shown by Howard [47] that there exists such a policy that is optimal for every state. Because of that property, the policy does not need to be derived explicitly; instead, we can compute maximum expected reward of each belief state $\mathbf{b} = (\mathbf{b}(s_1), \mathbf{b}(s_2), \ldots)$ as shown in Equation 2.6.

Because the function is non-analytical, Kaelbing *et al.* [69] describe how this value function can be solved to select an optimal action for every belief state. Sondik [125, 126] has shown that this value is convex and piece-wise linear for the finite case. For the finite case there exists a geometric interpretation such that each action spans a hyper-plane over the coordinates of the belief space. The height of the hyper-plane at a given belief state corresponds to the expected maximal reward that can be achieved in all future states after committing the action associated with the hyper-plane. Optimal control is achieved by selecting the action that is associated with the highest hyper-plane for the current belief state. Cassandra *et al.* [21] propose an efficient algorithm to approximate the infinite-horizon value function as a set of hyper-planes from the POMDP model parameters and show how the highest hyper-plane is selected for a given belief state. The top-most region of a hyper-plane is referred to as witness region. We use their algorithm in our model to

approximate the value function over the infinite horizon and to select the optimal recovery action.

The computation of $\mathbf{V}$ depends on the choice of γ. For computing an infinite-horizon value function, the parameter γ should be chosen fairly close to 1, otherwise some anticipated actions may have empty witness regions and will never be invoked. This seems to be the case if corresponding recovery actions incur a high cost, but are necessary to transform the system into a state of higher value in the future. Although the general interpretation of γ reflects the probability of termination, in the infinite case choosing γ too low may yield unexpected effects on the control policy. At the time this document is written we are not aware of a model or heuristic approach to set a value for γ to ensure that required actions, such as "declare-recovered" are invoked. In a practical implementation of this controller we would specify γ and then adaptively increase it until all required recovery actions have non-empty witness regions.

Belief Updates

As we have described in Section 7.1, after a recovery or probe action has been committed an observation is given to the controller either through error detection or fault diagnosis. This observation gives the controller information about the current system state that it uses to update its belief about the current system state (i.e., probability of individual faults and the probability of the non-fault state). Using the POMDP parameters this belief update is performed as shown by Equations 2.5.

7.1.6 Terminating the Recovery Process

To terminate the recovery process we need to make a fault free system the optimal solution for the POMDP in a finite number of steps. We do this by adding a POMDP non-fault state $f_\emptyset \in \mathbb{S}$. All recovery actions attempt to transform the system into this non-fault state. Therefore, the controller also maintains another dimension for the probability of being in the non-fault state in its belief state $\mathbf{b}(f_\emptyset)$. This dimension is equivalent to the confidence

of the controller being in the non-fault state and a threshold thereof can be leveraged as the termination condition, such that the controller declares the system recovered as soon as $\mathbf{b}(f_\emptyset) \geq 1 - \alpha$ with α being the significance level of being recovered.

We also include an empty observation in our model that is most likely to be emitted in the non-fault state for any action committed, as follows:

$$o_\emptyset = \text{argmax}_{o \in \mathbb{O}} \mathbf{\Omega}(o | f_\emptyset, a_i) \forall a_i \in \mathbb{A}$$

This assures that as soon as the controlled system reaches a non-fault state and does no longer emit any errors, the confidence of the controller about the system being recovered is strictly monotonically increasing. Furthermore, as described in Section 7.1.2, we impose restrictions to make the reward for successful recovery actions maximal, such that the derived optimal policy always tries to reach the non-fault state over an infinite number of steps (*see* Section 7.1.5).

Termination under External Influences

The termination criteria and process described in the previous section only hold if we strictly assume that external influences do not have an effect on the current system state. For example, termination is not guaranteed if the internal system state changes due to a non-agent influence. In practice an ongoing recovery-process can be disturbed by an independent transient fault or a recovery action causes another fault that was not accounted for in the specification of $\mathbf{T}$. A solution to enforce termination is to limit the total number k of recovery attempts. If the system has not recovered within k attempts, apply a super-recovery action, for example calling a system administrator. Another option to account for external influences that cause the system state to change within the set of known system states $\mathbb{S}$ is to smooth the specified probability distributions of $\mathbf{T}$ and $\mathbf{\Omega}$ to prevent the specification of impossible observations and state transitions for recovery actions. In any case state changes caused by external influences are outside our model.

7.2 Motivational Example

In order to highlight the benefits of our model we simulate a hypothetical system to demonstrate the ease of use of the model. In particular, we want to show how the controller parameters are expressed as POMDP parameters. Furthermore, we want to show how this mapping is used to compute the value function and, thus, implicitly the optimal policy. Third, we want to analyze the effect of uncertain error detectors and false positives in the error detection on the time to recovery.

7.2.1 Simulated System Parameters

We have chosen to simulate an instance of an image of a virtual machine. The virtual machine is running on a data-centre node that provides a cloud service. Our controller acts as a data-centre infrastructure service and has to fix broken instances of virtual machines. We treat the instance as black-box and only monitor the performance metrics that are exposed to the hypervisor.

Assume that a Service-Level-Objective (SLO) (*see* Sturm *et al.* [132]) has been defined against the metrics to perform error detection. An SLO violation indicates a broken system with a confidence of $c_d = 0.75$ (i.e., the fraction of SLO violations that actually identify a broken system).

In order to validate the unreliable error detection, we include a probe action that can be performed by the controller. If the probe fails, the system is most likely broken. If the probe succeeds, the initial error detection is most likely a false positive. We model the estimated confidence in the probe as $c_p = 0.99$. The approximate duration of a successful probe cycle is $t_p = 10s$. The reader should note that it is assumed that a probe action does not influence the probed system.

In order to recover the broken instance, we reboot the broken instance. The confidence in a successful reboot for a broken system is expressed as $c_r = 0.95$ and the time until the instance is assumed to have booted is expressed as $t_r = 120s$.

We also introduce a do-nothing action $a_\emptyset$ that has the duration of error detection $t_\emptyset = 2s$ and does not change the system state.

The reader should note that we only use the system states broken and operational to consider one particular fault; therefore, the belief state of the controller has only two dependent components and can be easily visualized.

7.2.2 Mapping Model Parameters to POMDP Parameters

First we allocate the states, recovery actions and observations. Since we only consider error detection with probing and not fault diagnosis the POMDP states are as shown in Equation 7.2. The set of POMDP actions is equivalent to the set of controller actions of the previous section and shown in Equation 7.3. The set of observations is shown in Equation 7.4. As described in Section 7.1.2, for each probe action we introduce a success and a failure observation.

$$\mathbb{S} = \{\text{operational}, \text{broken}\} \tag{7.2}$$

$$\mathbb{A} = \{\text{nothing}, \text{probe}, \text{reboot}\} \tag{7.3}$$

$$\mathbb{O} = \{\text{no error}, \text{error}, \text{probe fail}, \text{probe success}\} \tag{7.4}$$

We show the allocation of $\mathbf{T}$ in Table 7.2a. As described before, probe actions do not influence the system state, nor does $a_\emptyset$. As such their transitions are initialized to 1.0 for remaining in the state. In the specification we implicitly assume that a reboot has no side effects, such that a reboot from an operational state always yields an operational state. If the system is broken we use the c_r to specify the transition probabilities. The allocation of $\mathbf{\Omega}$ is shown in Table 7.2b. For any action, except probes, error detection is performed. As such the observation probabilities are initialized to the confidence of the error detection. For probes only the outcomes of the probe are relevant to reason about the current system state.

The reader is reminded that that $\mathbf{\Omega}$ has to provide probabilities for the total enumeration of possible observations for a given action and a given state. The transitionfunction

$a =$ **nothing**	$s' =$ **operational**	$s' =$ **broken**	$\sum$
$s =$ **operational**	1.0	0.0	1.0
$s =$ **broken**	0.0	1.0	1.0
$a =$ **probe**	$s' =$ **operational**	$s' =$ **broken**	
$s =$ **operational**	1.0	0.0	1.0
$s =$ **broken**	0.0	1.0	1.0
$a =$ **reboot**	$s' =$ **operational**	$s' =$ **broken**	
$s =$ **operational**	1.0	0.0	1.0
$s =$ **broken**	$c_r = 0.95$	$1.0 - c_r = 0.05$	1.0

(a) Transition function $\mathbf{T}$; s current state; s' target state

$a =$ **nothing**	$o =$ **no error**	$o =$ **error**	$o =$ **probe_ok**	$o =$ **probe_fail**	$\sum$
$s =$ **operational**	$c_d = 0.75$	$1.0 - c_d = 0.25$	0.0	0.0	1.0
$s =$ **broken**	$1.0 - c_d = 0.25$	$c_d = 0.75$	0.0	0.0	1.0
$a =$ **probe**	$o =$ **no error**	$o =$ **error**	$o =$ **probe_ok**	$o =$ **probe_fail**	
$s =$ **operational**	0.0	0.0	$c_p = 0.99$	$1.0 - c_p = 0.01$	1.0
$s =$ **broken**	0.0	0.0	$1.0 - c_p = 0.01$	$c_p = 0.99$	1.0
$a =$ **reboot**	$o =$ **no error**	$o =$ **error**	$o =$ **probe_ok**	$o =$ **probe_fail**	
$s =$ **operational**	$c_d = 0.75$	$1.0 - c_d = 0.25$	0.0	0.0	1.0
$s =$ **broken**	$1.0 - c_d = 0.25$	$c_d = 0.75$	0.0	0.0	1.0

(b) Observation function $\mathbf{\Omega}$; s state; o anticipated observation

Action	**Reward operational**	**Reward broken**
nothing	$-t_\emptyset = -2$	$-\min\{t_\emptyset, t_p, t_r\}$
probe	$-t_p = -10$	$-t_p$
reboot	$-t_r = -120$	$-t_r$

(c) Reward function $\mathbf{R}$

Table 7.2: Overview of selected POMDP parameters

T has to provide probabilities for the total enumeration of target states for a given action and a given system state. We highlight these consistency constraints by showing the row sums in the respected tables in Table 7.2.

The POMDP function is a complement of the costs of the actions. In order to express the times as rewards we invert them. Furthermore, the POMDP reward function does not accept units; therefore, we normalize the parameters by one second. In general all rewards should be expressed on the same scale of measurement. Furthermore, to bias the do-nothing action for the broken state, we initialize the reward to the maximum cost that is modelled. The allocation of **R** is shown in Table 7.2c.

The initial belief state can be initialized uniformly. However, since we assume the same error detection is used to start the recovery process we can use the confidence c_d as follows: We assume we have a 25% chance of a false positive. Therefore, the initial belief dimension for being broken is initialized to $\mathbf{b}(\mathbf{broken}) = 0.75$ and the belief dimension for actually being operational to the false positive rate, $\mathbf{b}(\mathbf{operational}) = 0.25$. This allocation is an analogy to initializing the belief state with a prevalence of known faults.

7.2.3 Computing the Value Function

To compute the value function we have chosen $\gamma = 0.95$. We approximate the value function over an infinite horizon to accommodate a Bellman error (*see* Cassandra *et al.* [21]) of $e_b = 0.05$. The hyper-planes of the derived value function are shown in Figure 7.2. With these parameters the action *reboot* has a witness region for a belief of being broken of about $P(\text{broken}) = (0.95 - 1.0]$. The action *probe* has a witness region of $P(\text{broken}) = [0.07 - 0.95]$. The action *do nothing* has a witness region of $P(\text{broken}) = [0.0 - 0.07)$.

7.2.4 Simulating Faults

The main focus of this chapter is the proposed recovery model; therefore, we restrict our evaluation to the recovery workflow. For that reason the only simulation artefact is the

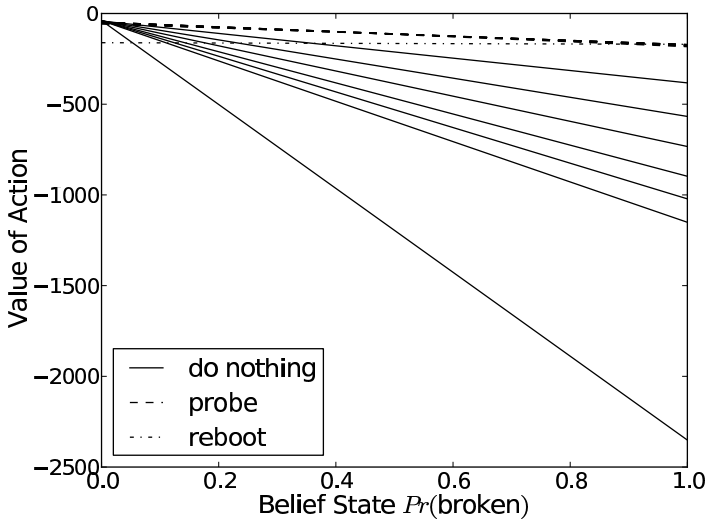


Figure 7.2: Hyper-planes of the value function

True Positives (error detection)	7557	False Positives (error detection)	2443
False Negatives (termination)	172	True Negatives (termination)	9827

Table 7.3: Distribution of initial and final recovery outcomes

simulation of false-positive error detection. We use Bernoulli trials with a probability of $P_b = 0.75$ to simulate a broken system. We model the effects of actions on the simulated system as a Markov chain using the same parameters as for the controller, such that the observations and state transitions follow the same distribution. We simulate 10000 runs of the recovery controller. Our objective is to analyze the behaviour of the controller in general and for false positives in particular (i.e., the recovery was started on an operational system). We observe the distribution of initial situations and false negatives (i.e., the recovery workflow was terminated prematurely on a broken system) that is shown in Table 7.3.

The distribution of total steps to recovery is shown in Figure 7.3a. In the majority of cases the controller terminates after just two steps. This fraction is mostly represented by

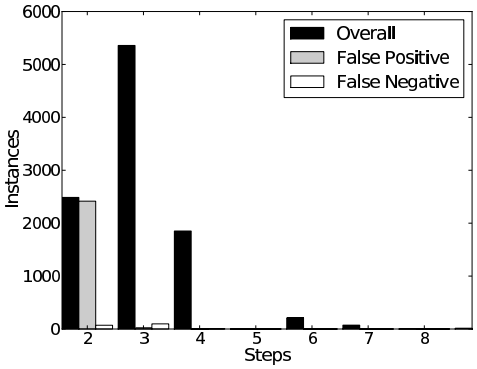
true-positive invocations of the recovery workflow. Since the initial belief state lies within the witness region of the probe action a probe is committed to validate the system state. For a true positive, initialization most likely resulted in a failed probe. The resulting belief update then put the controller in a state where the reboot was applicable. If the confidence of the controller in the recovery was high enough, the controller was terminated. Putting this observation into perspective with the false negative rate, one can see that the number of false negatives almost directly matches the confidence threshold on the recovery.

The distribution of chosen actions is shown in Figure 7.3b. Since the specification of the recovery action results in a value function where probe has the largest witness region (*see* Figure 7.2) and the initial belief state falls into this witness region it is the most dominantly applied action. Since the witness region of "do nothing" is so close to the recovery confidence, it is never invoked.

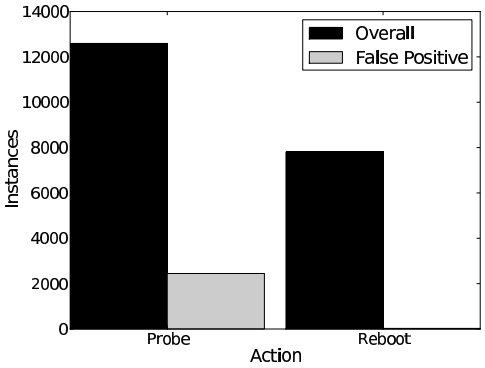
The overall distribution of recovery cost is shown in Figure 7.3c. The distribution of recovery costs is sparse. Most failures are recovered within 120-130 seconds, which usually maps to the application of one reboot cycle and a few probes. In false positive cases the system state was validated with a few probes and no reboot was applied. This is shown by the far left peak close to zero. In some cases the reboot failed then system had to be rebooted multiple times, which is shown by outliers on the right side of the spectrum.

7.3 Summary

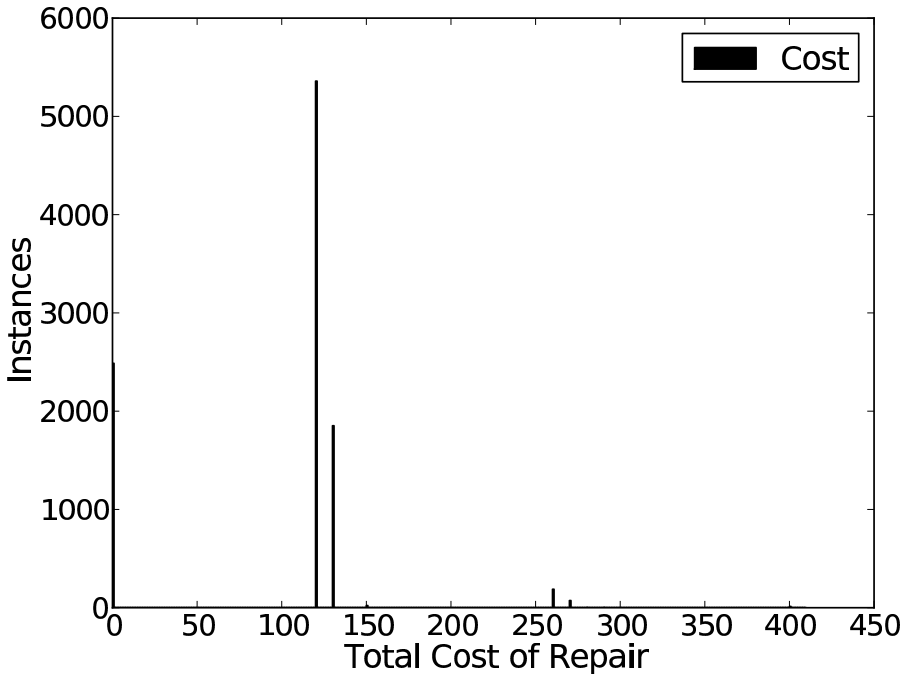
The methods for self-recovery surveyed in Chapter 3.4 do not include a comprehensive approach that integrates probing, diagnosis and recovery. In this chapter, we propose an approach to modelling proactive fault diagnosis and recovery. This model allows the specification of recovery and probe actions, which specifications integrate with fault diagnosis. Even in the context of uncertain fault diagnosis, controllers based on our model are able to recover from failure. Our model allows a very flexible specification of recovery and probe actions. Based on those specifications our model automatically computes an optimal policy for the information that can be obtained from the monitoring data of the system.



(a) Distribution of steps to termination



(b) Distribution of actions



(c) Distribution of total recovery cost

Figure 7.3: Evaluation results

Our proposal is the first to integrate diagnosis of recurrent faults, fault prevalence and the specification of probe actions.

We show through the use of simulation how an optimal policy can be inferred from a specification of recovery actions.

Although we believe that POMDP is a very powerful model to automate the operation of software systems, the paradigm may be simplified [110]. Many of the POMDP parameters are intuitively given (i.e., $\mathbb{A}$ and $\mathbb{S}$) or can be estimated from historic data, such as fault prevalence (i.e., $\mathbf{b}_0$) and the mapping of symptoms to faults (i.e., $\mathbf{\Omega}$). However, data to accurately specify $\mathbf{T}$ as well as $\mathbb{R}$ is hard to obtain. An accurate specification of these parameters strongly relies on expert knowledge. We [110] have evaluated preliminary reinforcement learning strategies for $\mathbf{T}$ and $\mathbb{R}$ that are based on delayed-cost observations. The assumption of total controllability is another limiting factor for the applicability of this model. In practice, faults may be transient or the system state may change without controller intervention. This model should only be applied to systems where this assumption with respect to the actions is reasonable and, as such, be seen as an addition to the approach that was discussed in the previous chapters that suits a broader range of applications.

Chapter 8

Evaluation

In this section we evaluate our approach to modelling log files, extracting symptoms of recurring faults and proactive diagnosis and recovery. Our evaluation methodology comprises controlled fault injection and real data acquisition experiments to obtain monitoring data from faults and simulation to evaluate self-recovery approaches.

We evaluate proactive recovery approaches by combining the diagnosis from the realistic fault injection experiments with simulated recovery. This simulation environment provides us with direct control of the effects and side-effects of the recovery and probe actions, enabling reproducibility of the experiments.

Furthermore, because the fault-injection experiments may appear synthetic we have also chosen to evaluate our log model and symptom extraction using a large data set that was obtained from a supercomputer.

We first describe our experiment set up and then evaluate all aspects of our approach in detail. For each aspect we describe the detailed evaluation objectives and the methodology to attain and quantify each objective. We evaluate our approach with respect to its diagnostic performance and scalability with other state of the art proposals in this area. We compare our proposal against related work with respect to the following objectives:

1. Scalability with respect to input data: How scalable is the approach with respect to the size of the raw monitoring data that is modelled?

2. Quality of features for symptom extraction:
   (a) How rich are the generated features of the raw input data?
   (b) How descriptive are the generated features with respect to the attributed faults?
   (c) Is the set minimal to identify recurrent faults?

We describe our experimental setup for the fault injection in Section 8.1. We describe the evaluation objectives and outcomes for each aspect of our approach in the individual sections. In Section 8.2 we evaluate our model of monitoring data against existing work [34, 64, 140]. In Section 8.3 we evaluate our approach to learning symptoms of recurrent faults. In Section 8.4 we evaluate proactive approaches to fault diagnosis. Finally in Section 8.5 we summarize the evaluation of our approach.

8.1 Experiment Set-Up

Our approach to experimentation is two-fold.

First, we evaluate our approach using a multi-tier test-bed that emulates a stock-trading system. We reused Munawar's test-bed [91] that injects a variety of faults into the system that we activate at defined points in time during the experiment. The principal advantage of this approach is the full controllability of experiment parameters that interest us. We control directly the types of faults injected, the duration of an active fault, observation when an error is present, and ground truth of the injected fault. This also includes injecting faults that are hard to diagnose or cannot be diagnosed at all from the monitored interfaces of the system. The details of this set-up are explained in Section 8.1.1.

Second, in order to ensure that our fault injection experiments are valid we also evaluate our system against real log data that was obtained from the BlueGene/L supercomputer that is deployed at the Lawrence Livermore National Laboratory (LLNL). This data spans approximately six months. Individual log records that are classified as alerts (i.e., they demand the attention of a system administrator) are tagged with pre-defined alert types. We use these alert annotations to model recurrent faults and extract the relevant monitoring data. We describe the related experiment set-up in Section 8.1.2.

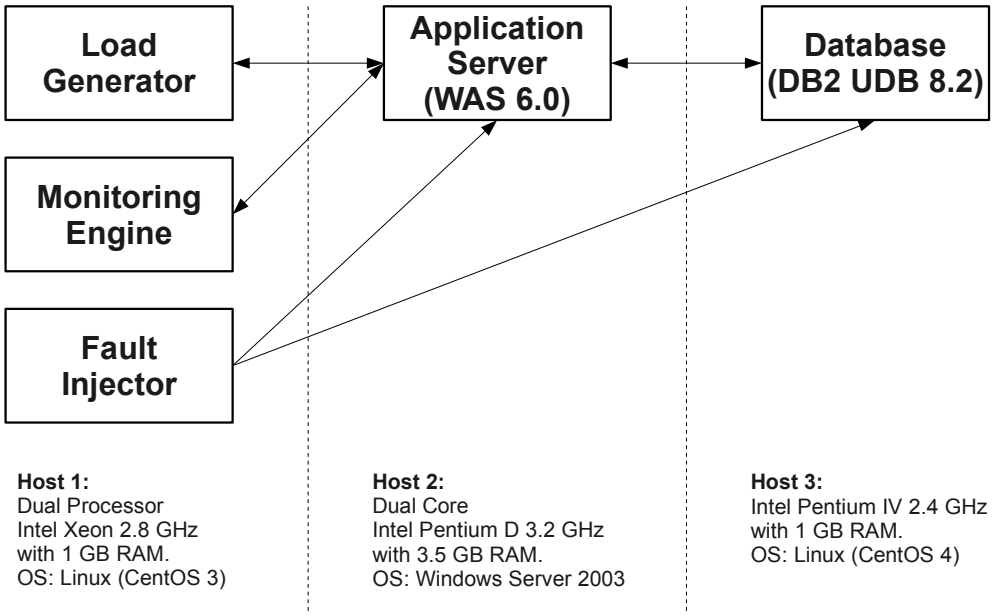


Figure 8.1: Evaluation testbed

8.1.1 Set-Up of Fault-Injection Experiments

The testbed used for fault-injection experiments is shown in Figure 8.1. It consists of a DB2 UDB 8.2 database server [50], a WebSphere 6 Application Server [52], a workload generator, and a fault-injection module. All components are connected using a gigabit Ethernet switch. The experiments are purposefully designed such that the network does not become a bottleneck. We execute the Trade6 benchmark [26] that emulates a stock-trading system on the application server. The stock data and accounting information is stored in the DB2 instance. We expose Trade6 to a steady workload, inject one fault per experiment into the system, and collect the WebSphere Application server trace-log file at the end of each experiment.

Workload Generation

Our objective is to emulate the system in operation; therefore, we expose the Trade6 application that is running on WebSphere to a random workload. Our custom workload generator implements a closed-loop workload to avoid overload of the system. We execute the workload generator on a separate machine specifically dedicated to that purpose. The random workloads are implemented by Trade6. Trade6 exposes a scenario interface that generates a random usage pattern of buy, sell, and user authentication client transactions when invoked. Our closed-loop workload generator invokes that interface to emulate the system in operation.

Data Acquisition and Transformation

After each experiment completion we move the WebSphere trace-log file [53] from the WebSphere machine off-site for further analysis. Each experiment, therefore, starts off with an empty trace-log file, and the trace-log file only spans the duration of each experiment.

The trace-log file is a plain-text log-file that monitors the individual transactions of the applications running on WebSphere. Each log record consists of a timestamp, thread-id, and various attributes that enable the identification of the component that issued the record. The reader should note, however, that this scheme is not consistently applied. From our observation, stack traces, exceptions, and data logged by other sources of recovery code are often logged verbatim into the file. Being able to process such data is crucial to modelling log data for diagnosing faults.

Because our objective is to evaluate our approach for modelling unstructured log data, we remove the timestamp and thread-ID attributes from the trace-log file and treat all other attributes as part of the plain-text message. For evaluating the fault injection data we consider the experiment duration equivalent to the window of interest (*see* Section 6.1).

Because several steps of our approach as well as related work rely on the tokenization of each log record, we split the plain-text message of each record by all non-word characters (i.e., regular expression `\W`). Furthermore, we filter out all tokens that entirely consist of

Window of Interest	60 mins
Delimiters	non-word characters `\W`
Ignored tokens	non-word characters `(\W+|[0-9a-fA-F]+)`
Removed attributes	timestamp thread-ID

Table 8.1: Data acquisition and transformation parameters for fault-injection experiments

non-word characters (i.e., regular expression `\W+`) and numeric tokens (i.e., regular expression `[0-9a-fA-F]+`).

All data transformations are our own implementations using the standard library of the Python programming language [104]. We summarize the data acquisition and transformation parameters in Table 8.1.

Experiment Timing and Injected Faults

Our experiment duration is 60 minutes. After a warm-up-period of 45 minutes, in which the closed-loop workload generation is active, we inject one particular fault of the fault types shown in Table 8.2 into a component of Trade6, the application server, and the database. We repeat each experiment 5 times and consider the collected log data as individual samples.

The fault types *busy*, *exception*, *null*, *remjsp*, and *sleep* are injected into various components. The injection of one fault into one particular component corresponds to one experiment. These faults emulate reoccurring software defects and operator errors. The faults *auth* and *threadpool* emulate operator error. The fault *dblock* emulates conflicting access or an inappropriately sized database instance.

The reader should note that the components were chosen arbitrarily. Specifically we inject faults also into components that do not log any events in the trace-log. Because of the limited system coverage of log files, a lot of faults occurring in real systems do not have

clear manifestations in the log files or are confused with other faults [98].

8.1.2 Set-Up of BlueGene/L Experiments

In this section we describe the set-up of our BlueGene/L experiments. In order to evaluate our approach, we have obtained system logs from the IBM BlueGene/L installation at Lawrence Livermore National Labs (LLNL); these logs were previously annotated by Oliner *et al.* [98]. The logs contain all monitoring records that have been generated by the system over a period of 215 days. Oliner *et al.* normalized the data and annotated records that would be of interest to system administrators as alerts.

BlueGene/L Architecture

The BlueGene/L architecture comprises up to 65,536 processors. In Figure 8.2 we show an overview of the packaging of the architecture. The processors of the architecture are configured as a $64 \times 32 \times 32$ torus. Up to 16 node cards can be installed on a node board. Up to 32 node boards are stored in a rack. A BlueGene/L installation can scale up to 64 racks [2]. For each rack there are two service nodes that perform the monitoring. The ASICs on each node card store errors locally until they are polled by the service nodes using a polling frequency of about one millisecond. The service nodes then relay the monitoring records to a centralized DB2 database. This database includes hardware and software errors from all monitored nodes of the installation [2]. The set that we obtained consists of a single file that includes the records of all nodes that occurred over the period of six months.

Data-Set Description

The data we are using is the same as used by Oliner *et al.* [98] who obtained the records from the centralized DB2 database, and tagged and normalized them. Examples of the log records are shown in Figure 8.3. Since new alert types have been discovered after its [98]

Fault name	Emulated Fault Type	Behaviour
auth	Operator	WebSphere JDBC authentication failed
busy	Performance	Randomly trigger busy loops in Trade6
connpool	Operator	Set database connection pool size to 1
exception	Software Bug	Randomly trigger exceptions in Trade6
null	Software Bug	Randomly return NULL from selected methods in Trade6
remjsp	Operator	Remove components from Trade6, including Java Server pages
sleep	Performance	Randomly trigger injected `Thread.sleep` in selected methods of Trade6.
threadpool	Operator	Set WAS threadpool size to 1.
dblock	Performance	Randomly trigger table locks in DB2

Table 8.2: Injected fault types

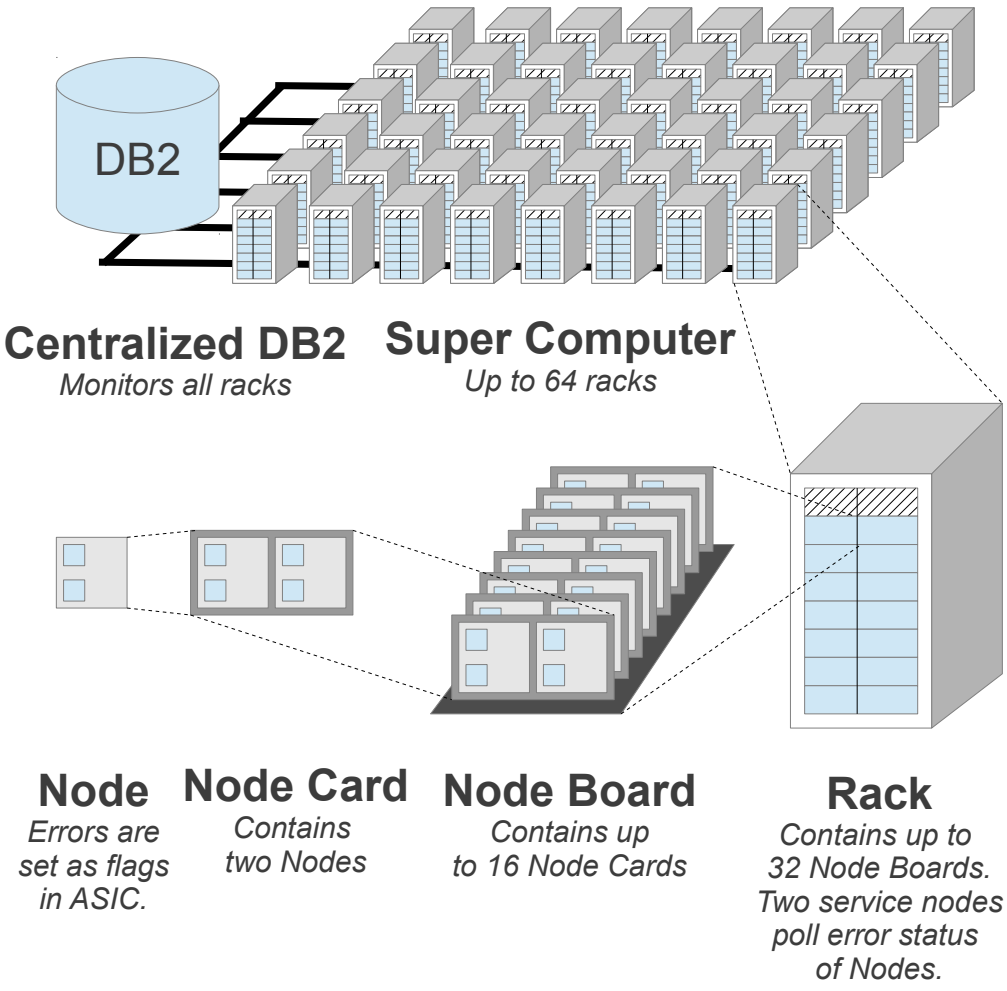


Figure 8.2: Conceptual overview of BlueGene/L architecture

```
- 1117838570 2005.06.03 R02-M1-N0-C:J12-U11 2005-06-03-15.42.50
     R02-M1-N0-C:J12-U11 RAS KERNEL INFO
     instruction cache parity error corrected;

R_DDR_EXC 1117846777 2005.06.03 R16-M1-N2-C:J17-U01 2005-06-03-17.59.37
     R16-M1-N2-C:J17-U01 RAS KERNEL INFO
     ddr: excessive soft failures, consider replacing the card
```

Figure 8.3: Example of normalized BlueGene/L log records

publication we updated the tags using the Tagger tool [127]. We have used the Tagger tool published on that website to annotate the original data set. Our evaluation is based on the updated data set. Each record has the following order of attributes.

- Alert type: A tag that identifies the type of alert, − indicating no alert.
- Normalized timestamp: The raw log timestamp converted to an UNIX timestamp (i.e., number of seconds since the epoch).
- Normalized source: The component or subsystem that emitted the log record.
- Raw timestamp: The timestamp of the record in the native format.
- Raw component: The component identifier in the native format.
- Message: The message associated with the record. The message may include severity or other specific attributes.

An overview of the properties of the data set is shown in Table 8.3. Compared to other supercomputer logs the BlueGene/L data set has a relatively small logging rate [98]. The number of different components that appear in the data set is larger than the maximum number of processors. This is due to other components, for example, the service nodes that also appear in the data set.

The number of alerts is relatively high, if each individual alert actually requires the attention of the system administrator. It is our understanding that many of the alerts are not independent, and possibly map to the same fault. For example, an error on a compute node may cause several flags to be enabled on an ASIC simultaneously. Such flags then generate multiple monitoring records (*see* Adiga *et al.* [2]). Unfortunately,

Property	Value
Size of data set:	708 MB
Duration:	215 days
Number of records:	4747963
Number of alerts:	348698 (7.34 %)
Number of filtered alerts:	1915
Number of alert types:	43
Number of components:	69251
Average size of record:	156 bytes
Logging rate :	40.0 bytes/second

Table 8.3: Properties of BlueGene/L data set

Oliner *et al.* [98] do not provide any direct comments if such errors were independent alert types or not; instead, they propose a filtering algorithm to filter out possibly dependent alerts. We describe the aspects of the filtering in detail in the next section. It should be noted, however, that this filtering reduces the number of alerts substantially, down to 1915 filtered alerts (on average 8.90 filtered alerts per day). The reader should note that our initial assessment of the data set differs in terms of total size, size-related metrics, and number of alert types from Oliner *et al.* because we updated it using the Tagger tool.

Approximating Failures, Recurrent Faults, and the Window of Interest from Alerts

Our objective is the classification of recurrent faults. Unfortunately, the data provided by Oliner *et al.* does not provide direct indications of failures or faults; therefore, we have to provide a mapping from alerts to failures and to recurrent faults.

To highlight the importance of this mapping, let us briefly review our definitions from Section 2.1. A fault is the cause of a system failure. A system has failed when it can no longer provide adequate service to its users. If two or more system failures are caused by the same fault then this fault is said to be recurrent. According to Oliner *et al.* [98] these alerts are monitoring records that merit the attention of a system administrator. Despite

Input: Sequence of alerts $\mathbf{A} = ((a_1, t_1), \ldots, (a_i, t_i), \ldots, (a_N, t_N))$
Output: Sequence of filtered alerts $\mathbf{A}' = ((a'_1, t'_1), \ldots, (a'_i, t'_i), \ldots)$
Set $l \leftarrow 0$
for $i \leftarrow 1 \ldots N$ **do**
    if $t_i - l > T$ **then**
        clear $(\mathbf{X})$
    end
    $l \leftarrow t_i$
    if $a_i \in \mathbf{X} \wedge t_i - \mathbf{X}[a_i] < T$ **then**
        $\mathbf{X}[a_i] \leftarrow t_i$
    else
        $\mathbf{X}[a_i] \leftarrow t_i$
        output $((a_i, t_i))$
    end
end

Algorithm 1: Alert filtering algorithm

having 43 different alert types, many of the alerts appear to be correlated with other alerts in the system and do not indicate independent failures.

In order to reduce the burden on system administrators, Oliner *et al.* proposed a temporal-spatial filtering algorithm to reduce the number of alerts generated by the logs; see Algorithm 1 for details. This algorithm takes the raw sequence of alerts and outputs a sequence of filtered alerts. Each alert is identified by an alert type a_i and a timestamp t_i. The vector $\mathbf{X}$ maintains the timestamps of the alert types last seen by a particular component within the time period T. The variable l is updated to the timestamp of the last alert seen. We followed the suggestion by Oliner *et al.* to set the period T to five seconds. An alert message is considered redundant if any source reported the alert type within the last T seconds.

We implemented this algorithm in Python based on the description of Oliner *et al.* Applying this algorithm reduces the number of alerts by 99.45%, to 1915 alerts (*see* Table 8.3). We assume that these filtered alerts approximate the *failures* that occurred during the lifetime of the data set. We use the alert type of the filtered alert to attribute the *fault* that caused the failure.

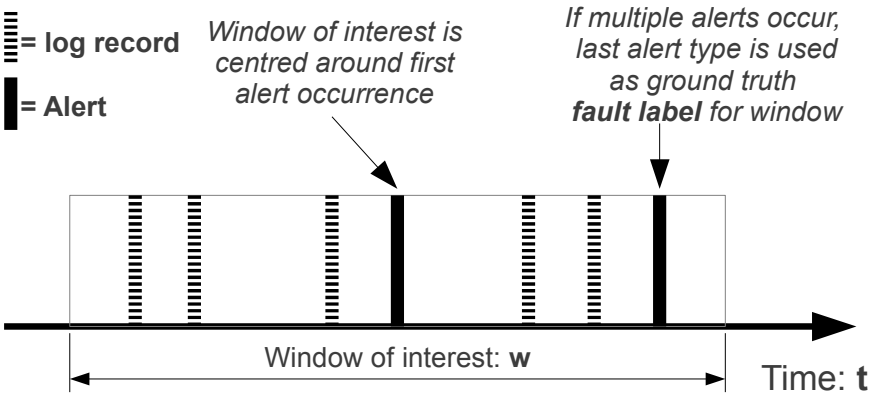


Figure 8.4: Window of interest and attributing fault type

Furthermore, these filtered alerts only indicate a point in time when the system has failed and do not indicate a duration of the failure. In order to select log records that need to be analyzed we define a *window of interest* of duration w around the filtered alert. All log records in that window are taken into account to model the features of the corresponding fault. The log records inside this window include alerts as well as non-alerts. We vary w between 600 and 3600 seconds. If in that window we encounter multiple filtered alerts we use the last alert in that window as the fault label. Therefore each window approximates an independent system failure. The label is used to model the ground-truth fault label for the symptom learning and classification. We illustrate that windowing concept in Figure 8.4. Table 8.4 show the types of alerts and their occurrence in the data set. We also show the regular expression used by Tagger that was used to attribute the alert type. From the table it can be seen that the number of possibly independent occurrences is reduced substantially for very frequent and software-related alerts. The consistency between the filtered and non-filtered alerts of hardware defects may be due to such events following stable long-run frequencies.

In Figure 8.5 we show the total number of approximated independent failures with respect to the window size w. It can be seen that the total number of failures gradually decreases with an increased window size. The heuristic filtering algorithm provided by

Alert Type	Number of alerts		Regular expression
	No filtering	Filtering	
KERNDTLB	152734	101	data TLB error interrupt
KERNSTOR	63491	8	FATAL data storage interrupt
APPSEV	49651	232	ciod:.*Link has been severed
KERNMNTF	31531	255	Lustre mount FAILED :
KERNTERM	23338	99	rts: kernel terminated for reason
KERNREC	6145	9	Error receiving packet on tree network
APPREAD	5983	17	ciod: failed to read message prefix on control stream
KERNRTSP	3983	293	rts panic! - stopping execution
APPRES	2370	18	Connection reset by peer
APPUNAV	2048	3	Resource temporarily unavailable
APPTO	1991	6	Connection timed out
KERNMICRO	1503	8	FATAL.*Microloader Assertion
APPOUT	816	4	failed: Input/output error
KERNMNT	720	9	Error: unable to mount filesystem
APPBUSY	512	1	Device or resource busy
KERNMC	342	326	KERNEL FATAL.*machine check (interrupt|register:)
APPCHILD	320	3	No child processes
KERNSOCK	209	39	KERNEL FATAL.*socket closed
R_DDR_STR	197	196	ddr: Unable to steer.*consider replacing the card
KERNPOW	192	12	FATAL Power deactivated:
LINKIAP	166	83	MidplaneSwitchController.*iap failed:
APPALLOC	144	2	Cannot allocate memory
KERNSERV	94	6	A service action may be required
R_DDR_EXC	41	41	ddr: excessive soft failures, consider replacing the card
MASABNORM	37	37	exited abnormally due to signal: (Aborted|Segmentation fault)
LINKDISC	24	12	MidplaneSwitchController.*port disconnected:
KERNPAN	18	15	KERNEL FATAL kernel panic
KERNCON	16	16	TIMEOUT connection lost to
LINKPAP	14	7	LINKCARD FATAL MidplaneSwitchController.*pap failed
KERNNOETH	14	13	KERNEL FATAL no ethernet link
MONPOW	12	12	FAILURE.*power module
MASNORM	10	10	BGLMASTER.*exited normally with exit code 1
APPTORUS	10	5	uncorrectable torus error
KERNPROG	5	5	KERNEL FATAL program interrupt$
KERNFLOAT	3	2	KERNEL FATAL floating point unavailable interrupt
KERNRTSA	3	2	KERNEL FATAL rts assertion failed
MMCS	3	2	MMCS FATAL L3 major internal error
MONNULL	2	2	MONITOR FAILURE While inserting monitor
LINKBLL	2	1	LINKCARD FATAL MidplaneSwitchController.*bll_clear_port failed
MONILL	1	1	MONITOR FAILURE monitor caught java.lang.IllegalStateException: while executing CONTROL Operation caught java.io.EOFException and is stopping
KERNTLBE	1	1	KERNEL FATAL instruction TLB error interrupt
KERNEXT	1	1	KERNEL FATAL external input interrupt.*tree header with no target waiting
KERNBIT	1	1	KERNEL FATAL ddr: redundant bit steering failed

Table 8.4: Alert types and the corresponding expressions for identification

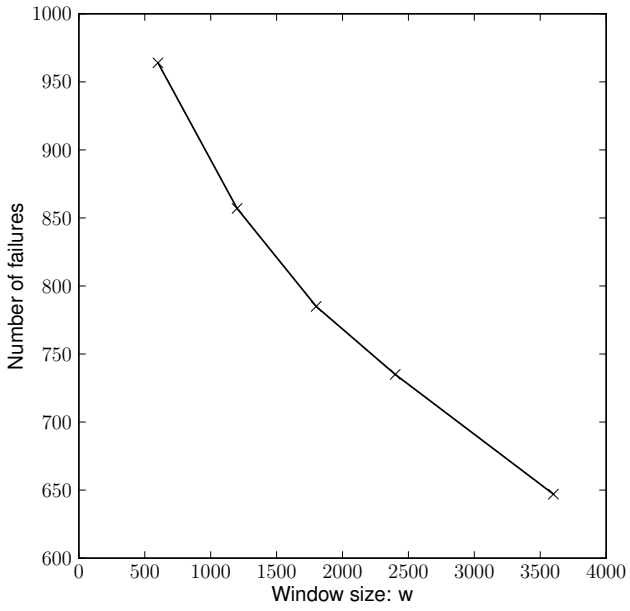


Figure 8.5: Relationship between failures and window size

Oliner *et al.* [98] may not filter out all dependent alerts. Given that the average inter-arrival time between filtered alerts is 4850 seconds, we may possibly mask independent failures for large values of w. Since the window size w has a significant impact on the number of independent failures for the BlueGene/L data set we will recognize it and evaluate it as a tunable parameter in the following sections.

Data Transformations

We apply similar data transformations to the BlueGene/L monitoring data that we apply to the data of our fault-injection experiments. These transformations are summarized in Table 8.5.

To reduce the size of the input data and exclusively focus on modelling unstructured monitoring data we remove the alert, normalized timestamp, normalized source, raw timestamp, and raw component attributes from the monitoring data and consider the message

Window of Interest w	600s - 3600s
Delimiters	non-word characters `\W`
Ignored tokens	non-word characters `\W+`
Removed attributes	alert type normalized timestamp normalized source raw timestamp raw component

Table 8.5: Data transformation parameters for BlueGene/L experiments

attribute exclusively. For further processing, we limit our evaluation to log messages that comprise the windows surrounding the filtered alerts using the filtering approach we described in the previous section.

For tokenization of each record, we split the plain-text message of each record by all non-word characters, filter out all tokens that entirely consist of non-word characters and mark the positions accordingly. This step is consistent with corresponding transformation set-up step used for the fault-injection experiments.

8.1.3 Experiment Set-Up Summary

We have described the experiment set-up to acquire samples of monitoring data. Further, we have described how to attribute ground truth labels that are used for symptom learning and classification.

It should be noted that the number of components, the topology, and the usage pattern of these two systems are vastly different. We purposefully selected such different set-ups to highlight the widespread applicability of our approach. As a result individual experiment parameters are not directly comparable across the fault-injection experiments and the BlueGene/L experiments. As we will summarise in Section 8.5, however, the conclusions drawn are consistent regardless of the application area.

The reader should note that that we evaluate the proactive diagnosis and recovery approach using simulation based on the outcomes of the log model evaluation of the fault-injection experiments. The reader is referred to Section 8.4 for details of the experiment description.

8.2 Log Model

In this section we evaluate our log model against other proposals in the field. We want to emphasise again that our focus is on modelling log files only from the log data itself, we only allow for minimal dependencies on expert knowledge about the system. Hence we chose to compare our approach only against approaches that make the same assumption, such as Fu *et al.* [34], Jiang *et al.* [64], and Vaarandi [140].

Our primary objective is to assess the scalability of our approach and compare it with other proposals in the field. In this section we focus our evaluation on the BlueGene/L data set (*see* Section 8.1.2). The reader should note that the findings from the BlueGene/L data set discussed in this section are consistent with what we observe from the fault injection experiments. We include a brief evaluation of the fault injection data in Section 8.2.6, which supports this claim. We will compare the BlueGene/L data set and the fault injection experiments in more detail in the sections that address symptom learning and identification.

As quantitative measures we have chosen to evaluate our proposal with respect to the following metrics:

- The time it takes to process a selection of records,
- The total number of features generated for windows of interest, and
- The number of features generated from a single sample of a defined size.

Because some of the approaches conceptually use the similar components to our proposal we will also evaluate the steps of these approaches in detail. We show the conceptual steps of each approach in our comparison again in Figure 8.6. The reader is referred to Chapter 3 for the detailed comparison of approaches. All approaches conceptually comprise of four types of forward processing steps. First the raw log data is filtered in an

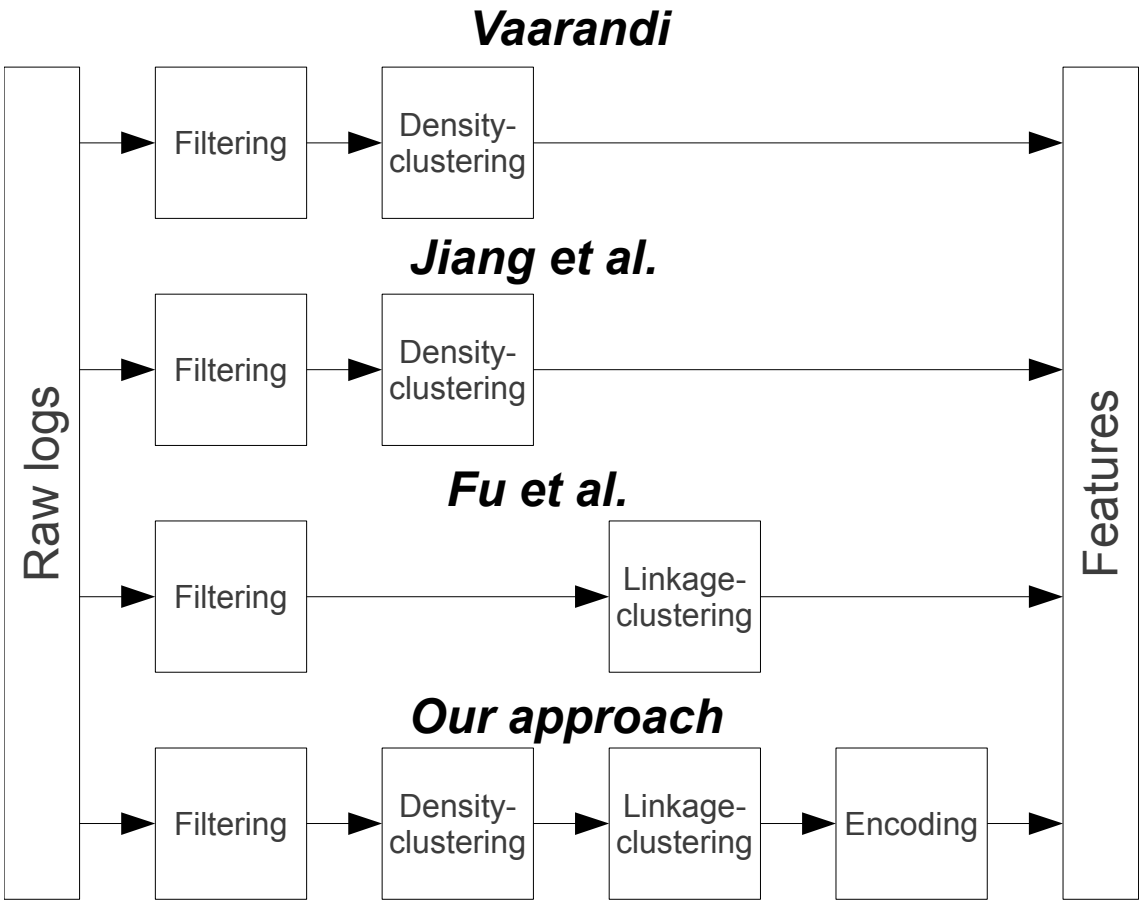


Figure 8.6: Conceptual comparison of unstructured log modelling approaches

attempt to remove volatile elements such as dynamic parameters from the individual log records. In our evaluation we apply the same filtering step to all approaches (*see* Tables 8.1 and 8.5). The outcomes of this step are discussed in Section 8.2.2. Our approach and the approaches proposed by Vaarandi [140] and Jiang *et al.* [64] contain a density-based clustering step that merges similar records into clusters. Many algorithms of this class expose linear complexity with respect to the number of input features (*see* Chapter 3). We discuss the results for the applicable approaches in Section 8.2.3. Because the nature of the density-based clustering approaches is very coarse with respect to producing meaningful features the approach by Jiang [64] uses a linkage-based clustering algorithm. In our approach we chose to combine the benefits of density-based clustering and linkage clustering. The outcomes of this step are discussed in Section 8.2.4. Finally because the outcomes of any clustering approach are clusters of multiple elements that need to be matched against the input data, we chose to encode our clusters for more efficient matching of features. We evaluate this step in Section 8.2.5. In Section 8.2.7 we summarize the results of the individual steps and provide an end-to-end comparison of our and related approaches.

8.2.1 Record Selection

In this section we describe the selection of records relevant to the performance evaluation. We use two approaches to select data for evaluation. The rationale behind this is to analyse selected aspects of our log model in more detail. We visualize the different objectives in Figure 8.7. First, we select all windows that are associated with failures using the approach illustrated in Figure 8.4 for varying window sizes. This selection is used to evaluate the overall performance of our approach. We use the same sample in Section 8.3 to evaluate the symptom extraction. The objective of this sample is to evaluate our data across parallel tasks. We filter each sample independently then cluster the filtered records and apply the remaining steps of our approach.

Figure 8.8 shows the number of failure-relevant records and the size of failure-relevant records with respect to the window size. It can be seen that the number of records as well as the size of the relevant records scales approximately linear with an increased window

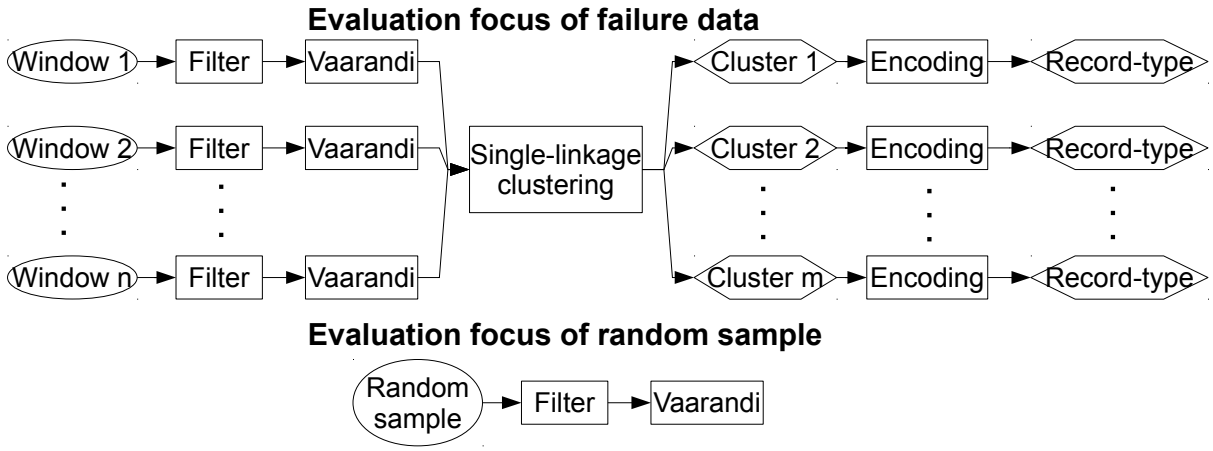


Figure 8.7: Objectives of record selection

size. Given the approximated failure rate (*see* Section 8.1.2), window sizes larger than an hour could potentially mask independent failures.

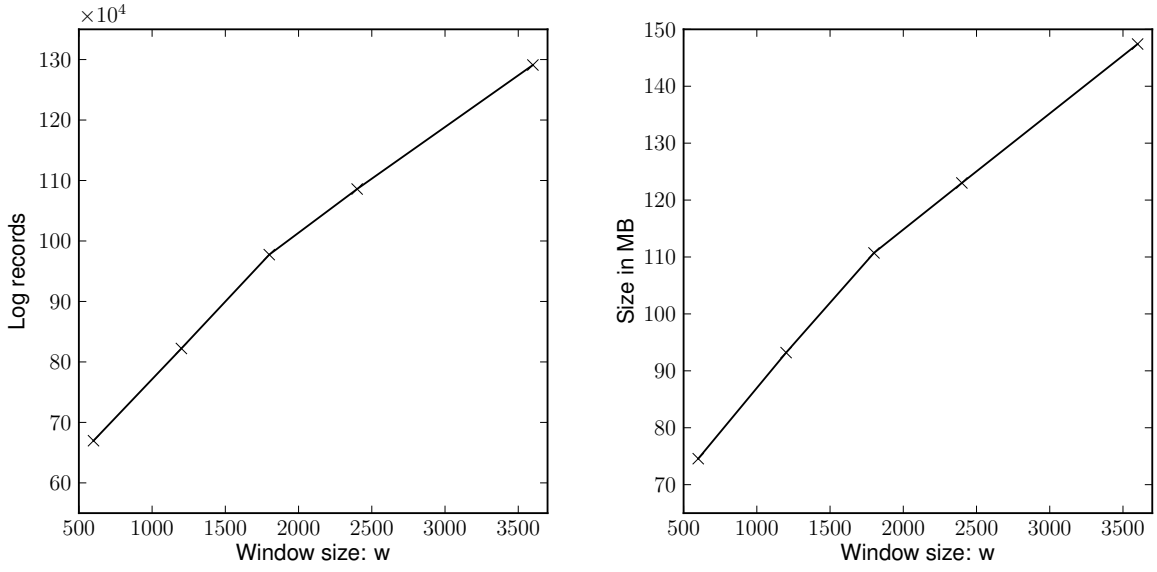
Second, we sample randomly from the BlueGene/L data set. This sample is used to evaluate the filtering and clustering steps in more detail and compare those with the performance of related work [34, 64] that do not expose parallelism.

The sample has a total of 4,747,963 records that consume 708 MB of storage. These records include log records that constitute to normal behaviour and alerts. We select between $250,000$ and $2,000,000$ randomly selected records with displacement. For each number of selected records we generate 10 samples. Figure 8.9 shows the average sizes and their 95 %-confidence interval for each selection. It can be seen that the size of the samples is fairly consistent with very little variation.

In the following sections we evaluate both data sets side-by side.

8.2.2 Filtering of Raw Logs

Given the samples described in the previous section, we analyze the impact of preliminary filtering in this section. This filtering step removes noise from the log-records that are



(a) Number of records

(b) Total size of records

Figure 8.8: Failure relevant log records with respect to window size

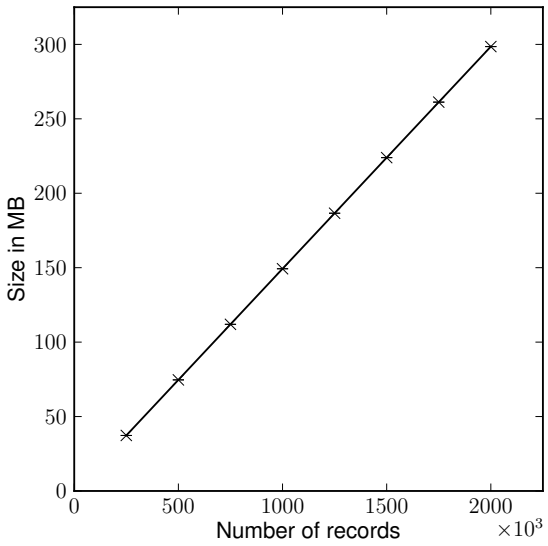


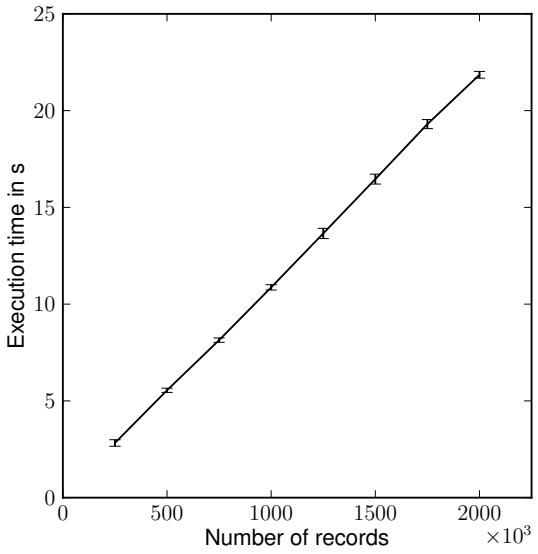
Figure 8.9: Size of random record selection

introduced by volatile parameter values. In order to allow a fair comparison of our work to the related approaches we apply the same filtering step to all approaches in comparison.

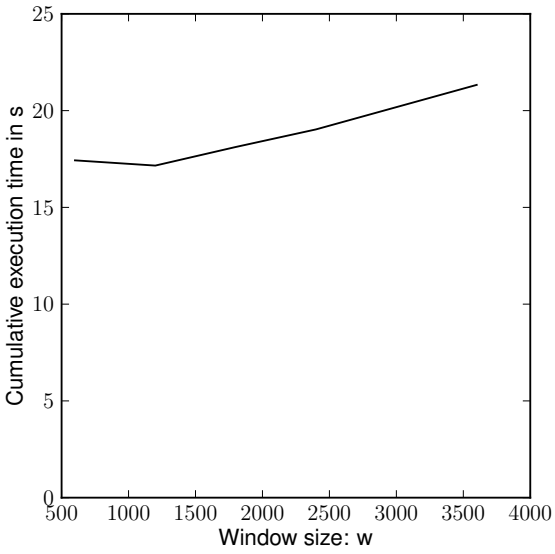
In this step we tokenize each log record, remove selected attributes and tokens as shown in Table 8.5 and Table 8.1. We show the running time, the number of generated filtered records for the random record selection in Figure 8.10a and for the failure data in Figure 8.10b. The reader should note that we display the standard deviation for the random samples because the data allows for variation (i.e., 10 samples per number of records). As these results show the execution time only has little variation for this step of the process. Because the failure data does not allow for variation of the input data, we do add variation by repeating the filtering process 10 times for each window size. The reader should note that the latter figure is indexed by the window size w and not by the number of records. This allows us to carry the execution times forward and summarize the outcomes with respect to our model parameter w. To obtain the mapping from window size w to the number of records, the reader is referred to Figure 8.10.

Figure 8.10b displays the cumulative execution time of all filtering tasks of the failure data. In Figure 8.10c we show the distribution of the execution times of the individual filtering tasks. As we have described in Section 5.3 the filtering is stateless and can be done independently for each window. The median execution time is 0.01s, which is the minimum resolution of the UNIX time command and may indicate that most of the samples are processed faster than indicated. This filtering step yields a set of unique patterns that describe the raw records of the input data as sequences of string tokens. The number of these unique data elements is shown in Figure 8.11. It can be seen that this filtering step identifies a relatively small number of unique features with respect to the input data. It may appear surprising to the reader, but in our experience system logs are only described by relatively small number of base-strings that vary in the parameters. In addition to dumps, corrupted log records have been removed from the BlueGene/L data set by the publishers [98] to allow processing. We do not show a confidence interval in Figure 8.11b because the input data does not allow for variation.

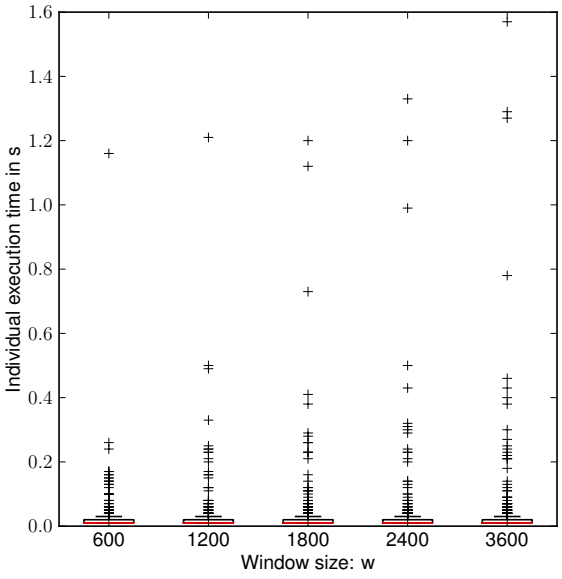
Of further interest to us is what the characteristics of the filtered log messages are. In order to show how the complexity of the features evolves over the processing steps, we show



(a) Random samples

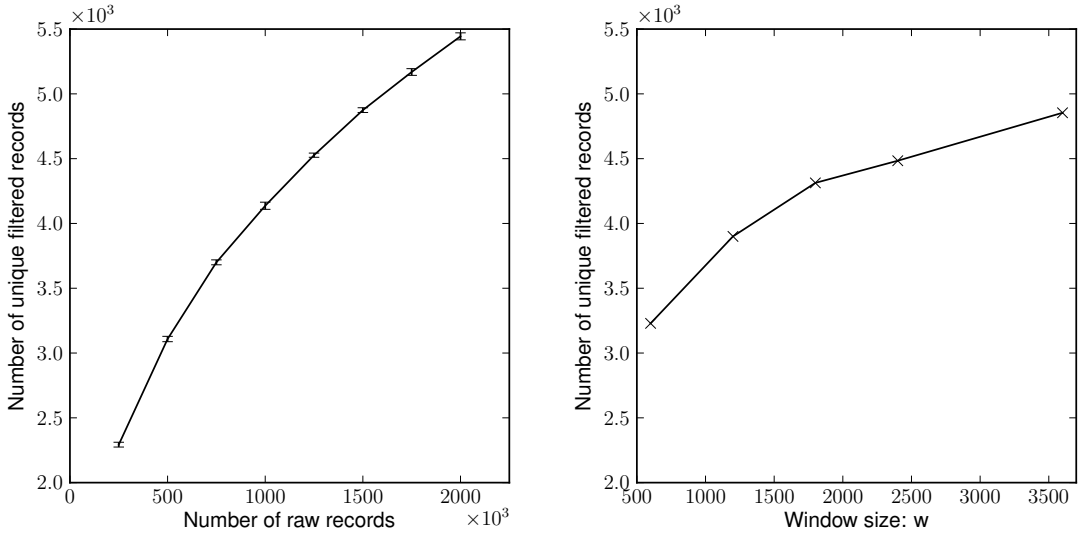


(b) Failure data cumulative



(c) Failure data individual

Figure 8.10: Execution times for filtering raw logs

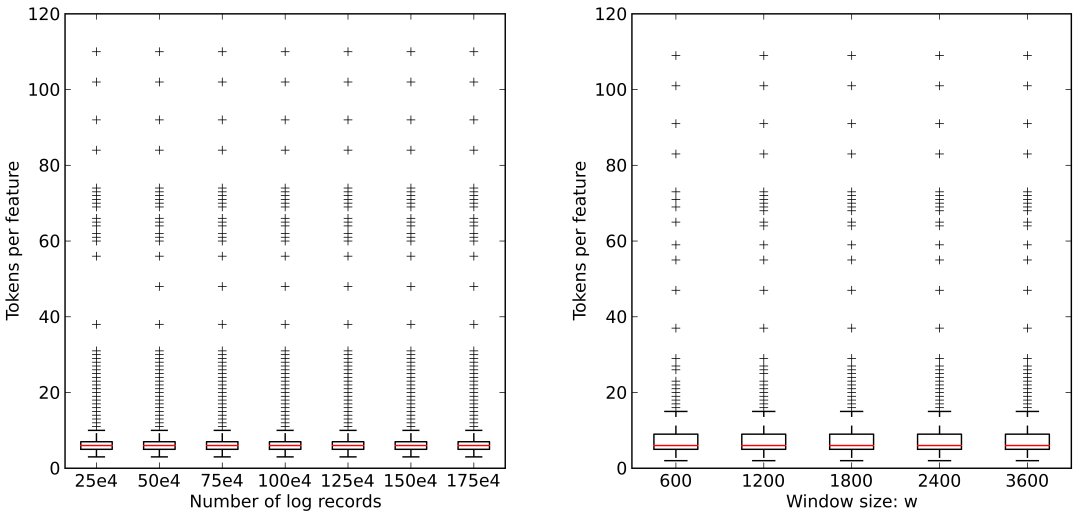


(a) Random samples (b) Failure data

Figure 8.11: Number of unique filtered records

the number of tokens per log record in Figure 8.12. As expected varying the scope of the log records in consideration has little impact on the distribution. It is interesting to note that individual log records can have up to 120 tokens per line and that the failure sample has slightly larger number of tokens per record. This may be due to log records that are emitted from recovery code tend to be more verbose than status information emitted under normal operation. The reader should note that the number of randomly selected samples and the total number of records covered by different window sizes is of the same order of magnitude. However, in order to highlight the scalability of our proposal we use use only a single filtering task to process random samples, while the all individual windows of interest from the failure data are processed in parallel. As such the input size to the filtering task is much larger for the random samples than the input size for the windows of the failure data.

The reader is also reminded of Figure 8.8 that shows the relationship of the window size and the size. The larger number of tokens in the failure data appears to be an artefact



(a) Random samples

(b) Failure data

Figure 8.12: Complexity of filtered records

that is related to the sample size.

Furthermore, we determine the minimum value of the sensitivity parameter s of Vaarandi's algorithm for our evaluation according to the heuristic proposed in Equation 5.3. We show the distribution of the filtered records for each window size of the failure sample in Figure 8.13. The medians are 71, 105, 134, 167, and 254 for the window sizes 600s, 1200s, 1800s, 2400s, and 3600s respectively. For example, for the 3600s window the minimum sensitivity for Vaarandi should be chosen at $s = 0.004$. That value of s would extract most records from samples as independent regions that have an equal amount or less filtered records than the median. For the failure data as well as the random selection of records we have chosen to evaluate $s \in [0.0078, 0.2500]$.

The reader should note that our implementation is pipeline based. The raw logs are read from a gzip compressed stream using `gzip -d -c` and then piped into our filtering program implemented in Python [104]. The outputs (i.e., the filtered records) are piped into a file that resides on disk. Given the relatively small size (50-300MB) of the input

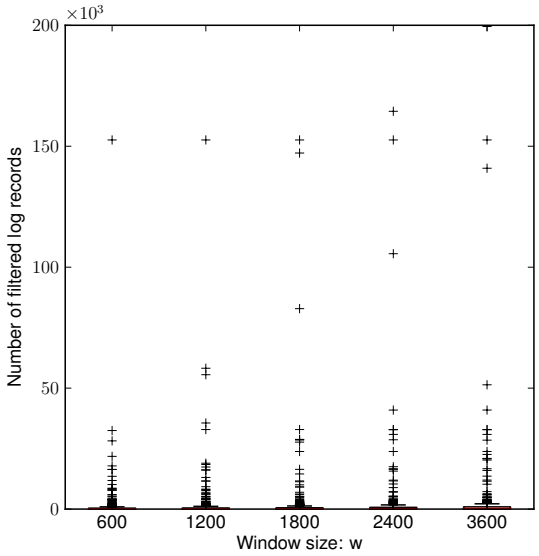


Figure 8.13: Failure data: Length of filtered samples

data, we do not expect disk-IO to be a bottleneck. The measurements were obtained by running the implementation on an Intel Core i7-2600K (i.e., has 4 cores) system clocked at 3.40 GHz that is running Ubuntu Linux 11.10 and has 16 GB RAM. The measurements were obtained using the real run-time as obtained by the BASH time command. The system was only running our single-threaded filtering program and was otherwise idle.

8.2.3 Density-Based Clustering

After having described preliminary filtering, we compare the running times and number of generated clusters for our and all the related approaches that contain a density based-clustering step. Specifically Vaarandi [140] and Jiang *et al.* [64] use density-based clustering techniques to describe the clusters. The reader should note that the clusters that are generated at this stage generate the final feature-set for Vaarandi [140] and Jiang *et al.* [64]. Our approach leverages Vaarandi [140] too but refines the created features in later stages in an effort to have a smaller feature-set of features that can be efficiently matched; therefore,

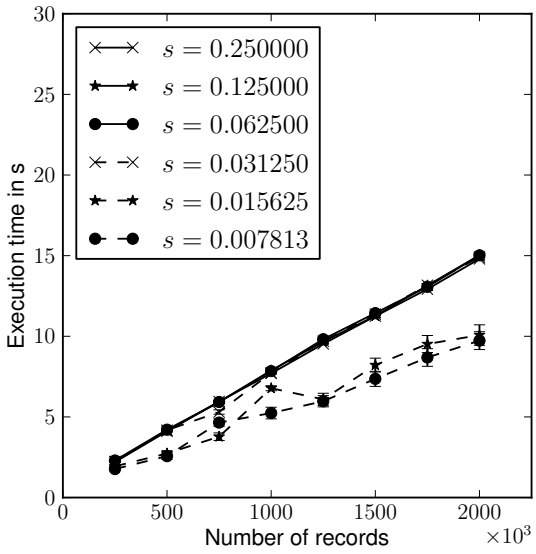
we limit our comparison here to Jiang *et al.* and Vaarandi in this step. Our approach reuses the data generated by Vaarandi in this step in the later stages.

We implemented the approaches of Vaarandi [140] and Jiang *et al.* [64] based on the authors' descriptions in Python [104]. We modified Vaarandi's proposal such that we can express the sensitivity of the algorithm with respect to the input size. Vaarandi's proposal stated the number of log records needed to form a region as absolute parameter while we express this measure with respect to the input size (*see* Chapter 5.3.1). In our observation Vaarandi's algorithm is very sensitive to the size of the input data. Being able to express the parameters with respect to the input data makes the approach more robust.For evaluating the proposal by Jiang *et al.* [64] we chose to use the parameter values stated in their paper.

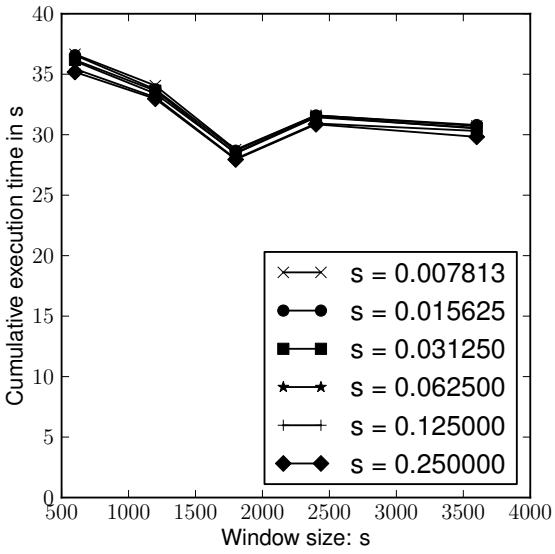
We apply this clustering step on the data to the filtered samples generated in the previous step. Figure 8.14 shows the execution times of Vaarandi and Figure 8.15 shows the execution times of Jiang *et al.* The number of features is shown in Figure 8.16 for Vaarandi and in Figure 8.17 for Jiang *et al.* The complexity of the features, as in the number of tokens per feature, is shown in Figures 8.18, 8.19 and 8.20.

As described in Section 3.1, Vaarandi's algorithm has a linear complexity. This feature is consistent with the execution time of the random sample shown in Figure 8.14a. The same applies to the algorithm by Jiang *et al.* For the evaluation of our further steps we set the sensitivity of Vaarandi's algorithm at $s = 0.0078$. We included that figure for comparison in Figure 8.15a. The execution times of Jiang *et al.* are of the same order of magnitude as Vaarandi; however, Jiang *et al.* appears to have a steeper slope than Vaarandi. This may be due to additional indexing overhead that is required for Jiang *et al.*, because this approach retains the features as clusters, while Vaarandi's features can be expressed as single region. Interestingly the cumulative execution time of Vaarandi's algorithm decreases with an increased window size. The choice of s not only controls the sensitivity with respect to the records, but also the sensitivity with respect to the tokens. Closer analysis revealed that this appears to be an artefact of our implementation of the heuristic filter (*see* Equation 5.2) that indexes tokens faster for smaller values of s.

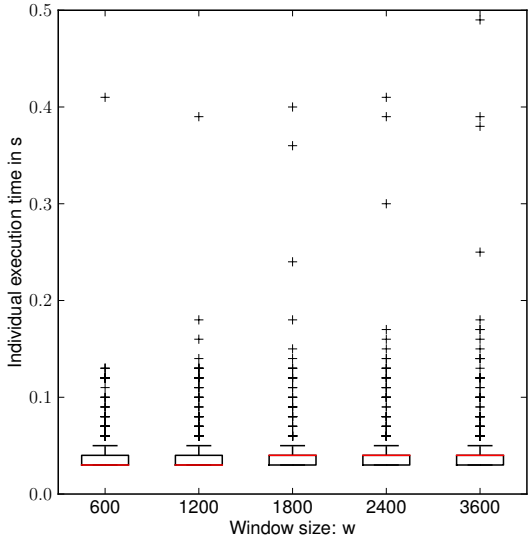
The number of features for Jiang *et al.* seems to be fairly consistent at about 130



(a) Random samples

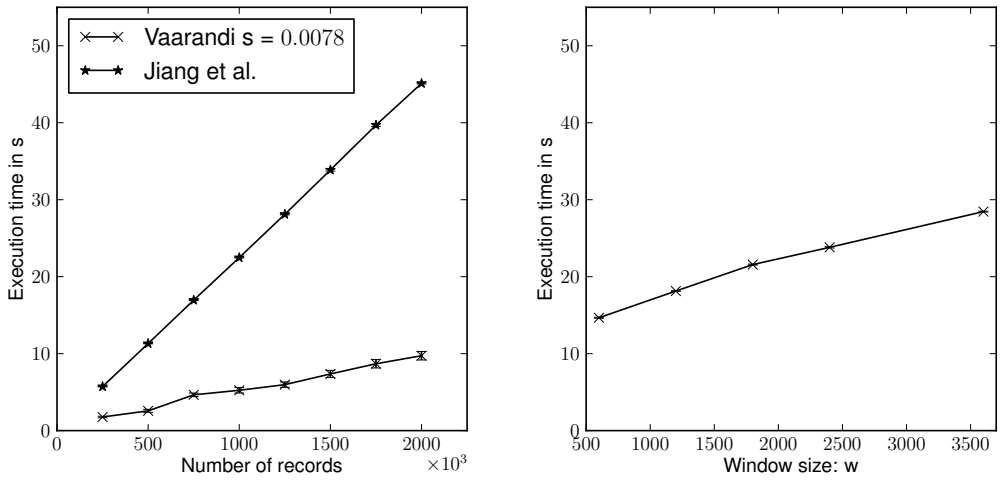


(b) Failure data cumulative



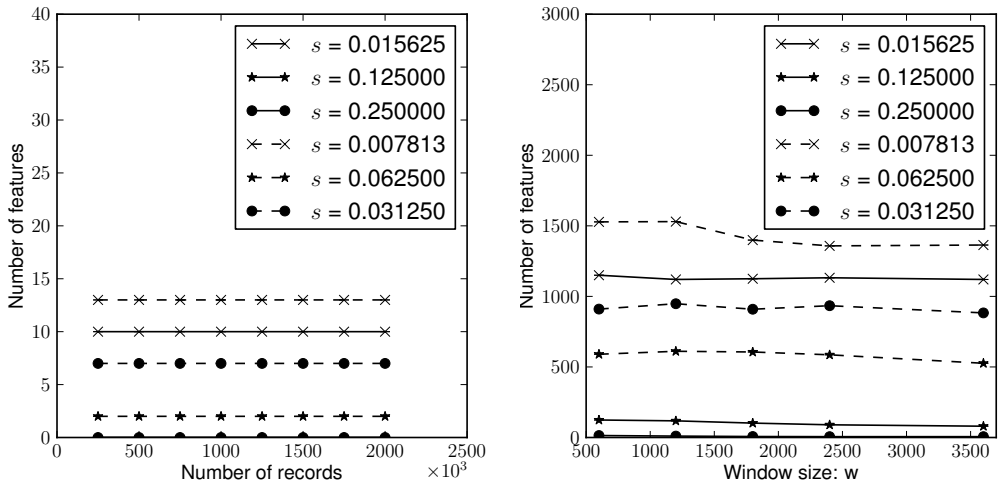
(c) Failure data individual ($s = 0.0078$)

Figure 8.14: Execution times for different sensitivities of Vaarandi's algorithm



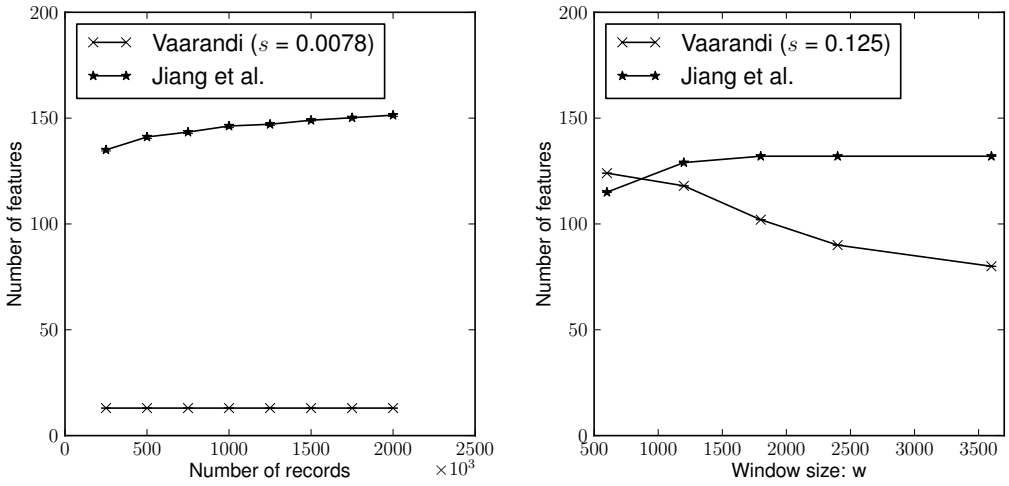
(a) Random samples (b) Failure data

Figure 8.15: Execution times of Jiang *et al.* and Vaarandi



(a) Random samples (b) Failure data

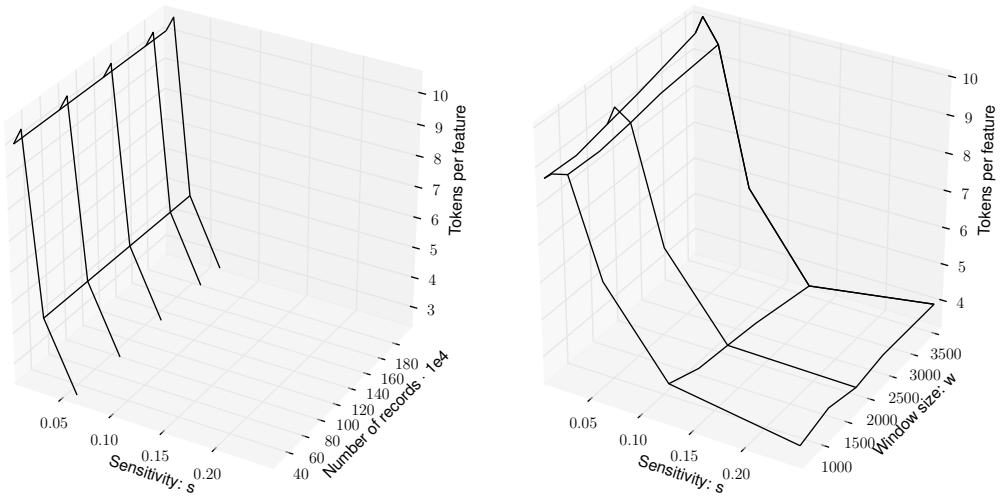
Figure 8.16: Number of features for different sensitivities of Vaarandi's algorithm



(a) Random samples (b) Failure data

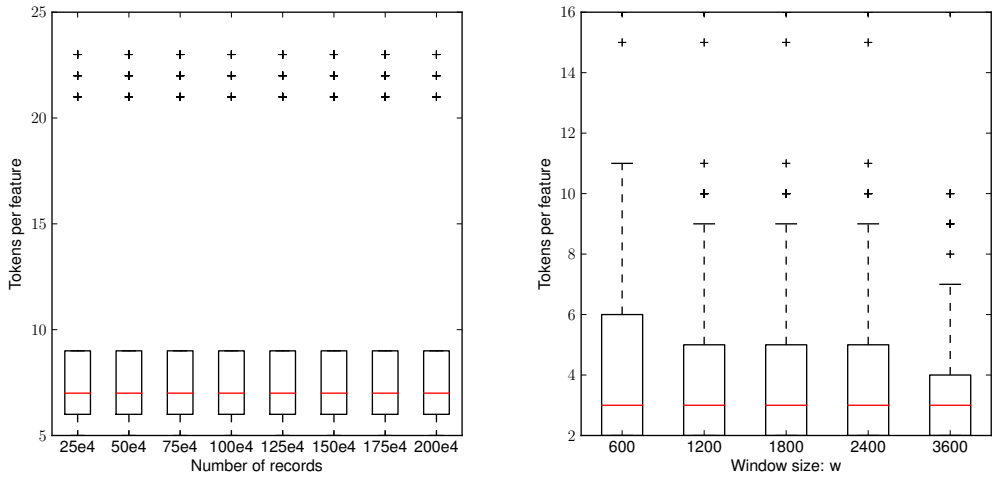
Figure 8.17: Number of features of Jiang *et al.* and Vaarandi

features. It can be seen that the approach by Vaarandi is very sensitive to the choice of s. A coarse sensitivity of about $\frac{1}{4}$ only yields one or no features. These are features that are present in almost all log records. In the case of the BlueGene/L data set these features are `KERNEL.*FATAL*` for the failure data set and `RAS.*KERNEL.*` that are very frequent in the respective data sets. A small value of s also increases the risk of over-fitting because then almost every log-record is allowed as part of the features. We included various values of Vaarandi in the comparison. It should be noted that Vaarandi is very sensitive to the size of the log file. Because we set the lower limit of s according to the failure data, it only extracts a small number of features when used against a large selection of random records. Due to assumption that the behaviour of a system can be characterized by the record types over a fixed time period, our modelling approach extracts coarser features from windows that have a large number of records. As shown [45, 98], error messages tend to occur in bursts. In order not to over-fit individual records of bursts we normalize the sensitivity of the algorithm to the size of the sample. Since in this case the records were chosen entirely at random and did not contain any significant bursts.



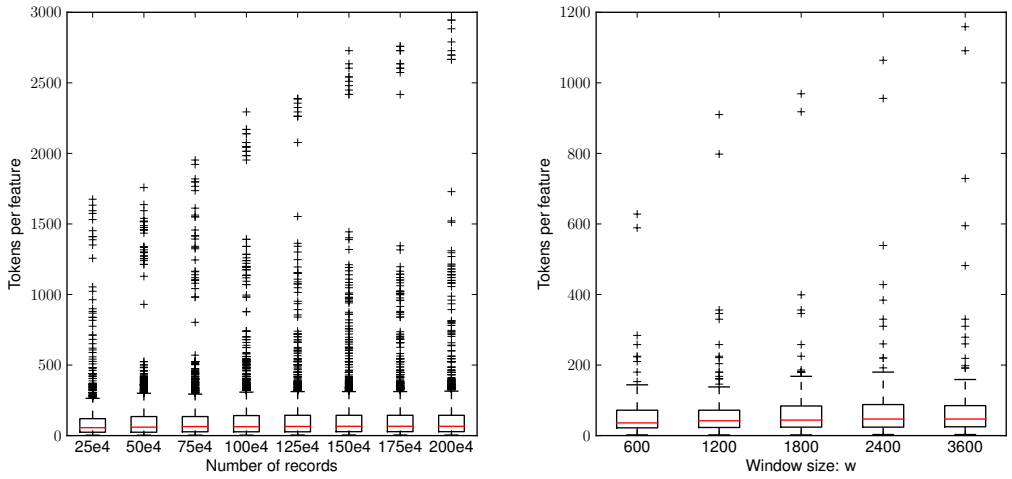
(a) Random samples (b) Failure data

Figure 8.18: Average trend of the complexity of features of Vaarandi



(a) Random samples ($s = 0.0078$) (b) Failure data ($s = 0.125$)

Figure 8.19: Complexity of features of Vaarandi



(a) Random samples

(b) Failure data

Figure 8.20: Complexity of features of Jiang *et al.*

Although the number of features of Jiang *et al.* and Vaarandi are of about the same order of magnitude for our parameter selection for the failure data, the complexity of features greatly varies. Since Jiang *et al.* retain the individual features as clusters of patterns, the clusters are comprised of many more tokens than Vaarandi, which can express any cluster as finite sequence of tokens as regions of density. The high complexity of the features of Jiang *et al.* may be an indication that the approach is prone to over-fitting. The corresponding complexity figures are shown in Figure 8.19 for Vaarandi and Figure 8.20 for Jiang *et al.* The average trend over the entire parameter space is shown for Vaarandi in Figure 8.18. It can be seen that the complexity of Vaarandi remains relatively stationary for a fixed choice of s and is not directly impacted by the size of the input set. This insight was our primary motivation to modify Vaarandi's proposal to use a sensitivity measure that can be expressed in relation to the input size. The parameter s should be chosen such that the number of features generated as well as the complexity of the features is sufficient. Given the low complexity and hence low overhead of the algorithm that choice can be easily made by an expert using a full-parameter sweep across the input data and

evaluating figures such as Figure 8.18 and Figure 8.16. In the case of the random sample it can be seen that a threshold $s > 0.0625$ does not extract any regions and, as such, the complexity figures are missing too. When choosing s too large there is a risk that large samples of monitoring data that do not contain repeated error messages or bursts of messages will be missed by the feature extraction. The running time of this step is relatively short, such that, that it is possible to evaluate a possible range of parameter for s larger than the minimum proposed in Equation 5.3. We will show in Section 8.3 that the cost of performing a cross-validation of a classifier can also be done in acceptable running time to evaluate the impact of s on the classification performance.

8.2.4 Linkage Clustering

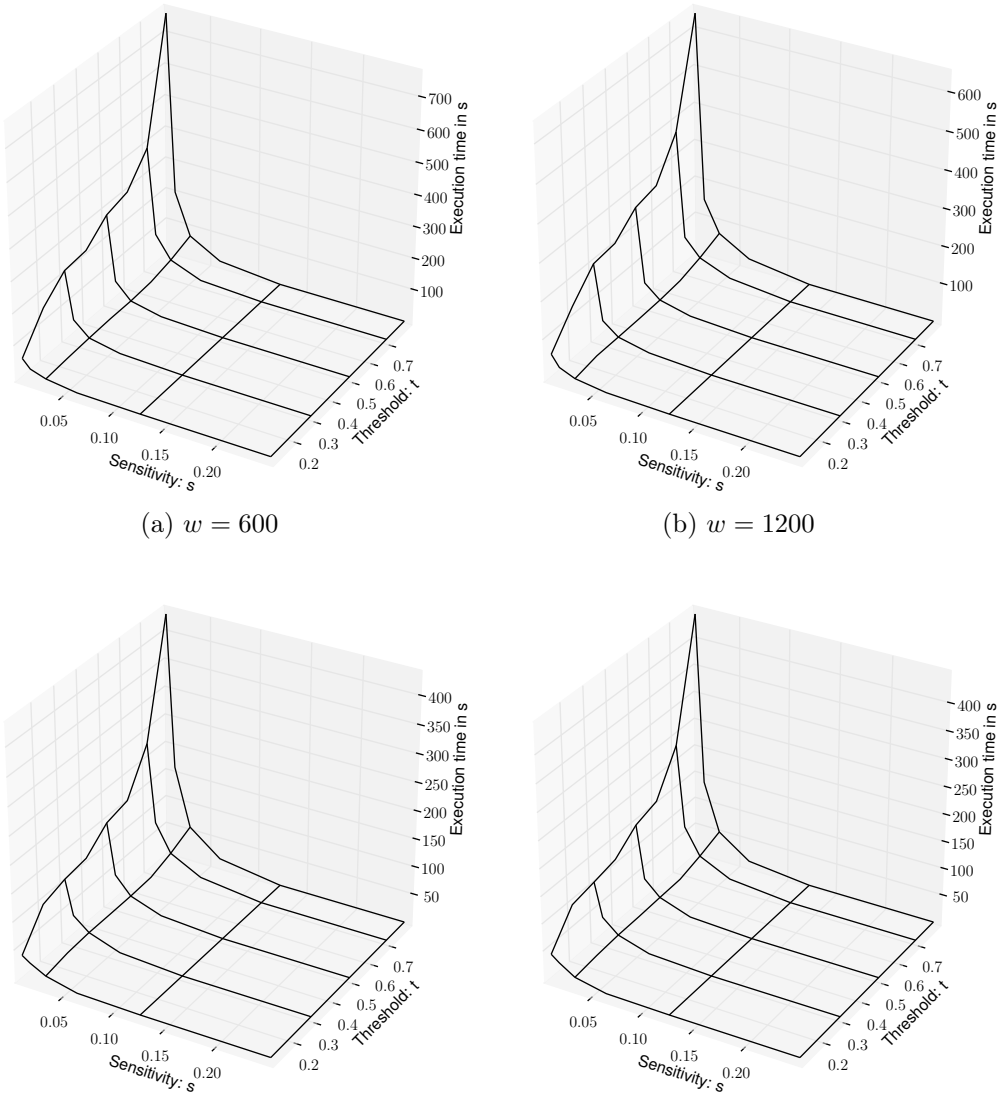
After having described how density-based clustering is used to reduce the feature-space of the filtered records for the respective approaches, we compare related work to our approach that uses linkage-clustering. The primary conceptual difference between density clustering approaches and linkage clustering is that linkage-clustering approaches rely on a one-to-one comparison of the individual elements. Density-based approaches select a seed item and grow these seed items into clusters with more input data being compared to that seed; therefore, density-based approaches expose a lower complexity than linkage-clustering approaches as we described in Chapter 2.4. Unfortunately, simplifications need to be made to represent and compare these seeds against the input data. A one-to-one comparison of all elements of the input data allows for more detail during the comparison at the cost of a higher complexity.

In this section we compare our approach to Fu *et al.* [34]. Being aware of the increased complexity of linkage-clustering approaches, we perform this step after performing a first round of density-based clustering using Vaarandi's algorithm [140]. Fu *et al.* use linkage clustering as only measure to aggregate the input data into features. As we show in this section this decision puts a significant burden on the scalability of their approach. Our approach and the approach by Fu *et al.* need to establish a threshold t for the clustering of the input-data. Our approach as well as their proposal use derivatives of the Levenshtein

distance [72] as distance function. Fu *et al.* propose a normalized variant that aims to follow a sigmoid pattern [34]. In our approach we simply compute the Levenshtein distance between two elements and normalize it by the length of the largest element. The reader should note that we do not compare the input data character by character, but instead compare the input data token by token using the Levenshtein distance.

Furthermore, because the computation of the Levenshtein distance has quadratic complexity with respect to input data we heuristically optimise our implementation. Assuming that the distance matrix of the input data is sparse and not many elements can be merged into clusters, it is prudent to minimize the computation overhead for items that will not be merged. Because we normalize the Levenshtein distance by the length of the two elements that are compared, a simple comparison of the lengths is needed to decide whether the Levenshtein distance needs to be computed in the first place. If the lengths vary by more than the threshold t we label the elements are too far apart, and the distance is not computed. This heuristic optimization substantially reduces the execution of our approach. Unfortunately, Fu *et al.* do not offer any optimizations to their distance metric. Even our close evaluation of their work did not offer us any hints how to heuristically predict the outcome of their distance metric efficiently and optimize their approach. It is our impression that the authors did not consider the input size as problem of their approach. Because Python [104] is an interpreted language we decided to implement the distance metric by Fu *et al.* in C and import the function as a native code module in the Python interpreter. In our implementation all string tokens are converted to their integer hash value, and then the distance computation is carried out on the array of hashes of the tokens. While these implementation details cannot optimize the complexity of the Levenshtein algorithm, they substantially reduce the execution time of the implementation.

In Figure 8.21 we show the execution times of our approach for the failure sample and selected window sizes. The execution times decrease with an increased window size and an increased sensitivity s because the input size was reduced in such a way in the previous step. Furthermore, it can be seen that the execution time increases with increases in t. Increased values of t cause more elements to be merged into the individual cluster, which increases the running time of the clustering algorithm. These outcomes also highlight the



(a) $w = 600$ (b) $w = 1200$

(c) $w = 2400$ (d) $w = 3600$

Figure 8.21: Failure data: Execution times of single-linkage clustering (our approach)

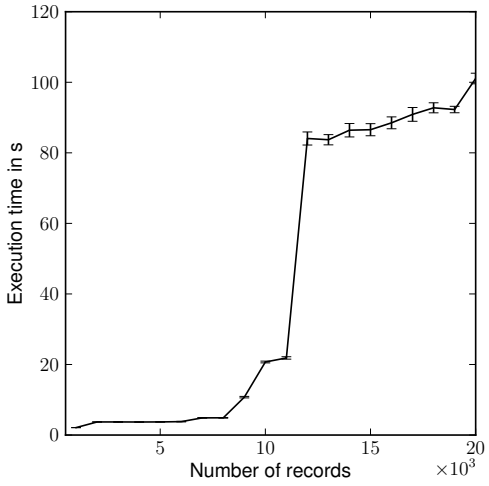


Figure 8.22: Execution times of Fu *et al.* using reduced input size

importance of the heuristic to limit s to reduce the burden of this step of our approach.

The inputs to this clustering step are the outcomes of Vaarandi's algorithm. We attempted to run Fu *et al.* against the failure set as well as the randomly selected records. Unfortunately, our implementation did not complete in time. In order to highlight the overhead of the algorithm, we ran the algorithm against 1000 to 10000 randomly selected records from the filtered data set. The outcomes are shown in Figure 8.22. For this artificial workload we repeat each execution 10 times to add variation. Besides being sensitive to the number of input elements, the algorithm also appears to be sensitive to similarity properties of the input data. The more clusters that need to be merged the more overhead is created. A symptom of this property is seen in the steps of the execution time for an increasing number of records. It should be noted that this random selection used for Fu *et al.* is equivalent to only approximately 0.05 % to 0.4 % of the filtered records that are processed by our approach. It also may seem unfair to the reader to compare the two approaches with different input data sets, but the reader is reminded that the proposal by Fu *et al.* by design operates against the filtered samples. The reader is referred to Section 8.2.7 for an end-to-end comparison of the approaches.

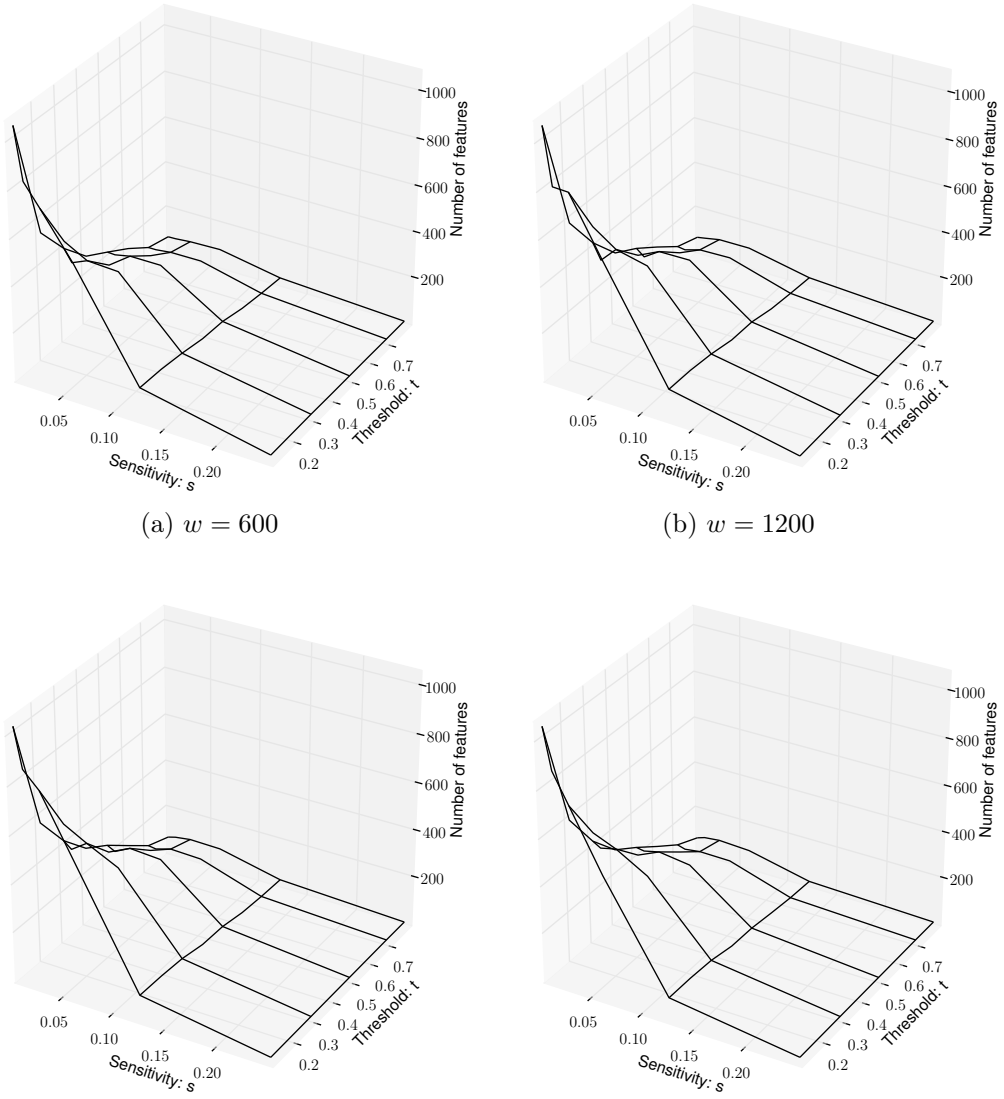
With respect to our proposal it can be seen that although the complexity of linkage clustering is high, by effectively reducing the input size by applying density clustering and filtering first, the overhead of this step is very small. As described in Chapter 3.1, our approach heuristically estimates the threshold t for the linkage-clustering. The 95 % confidence interval for that parameter is $t = 0.43 \pm 0.01$ for the failure data set for all selected window sizes. As follows we use $t = 0.4$ unless stated otherwise.

Because Fu *et al.* did not return adequate results for the complete set of randomly selected log records and the failure data set, we limit the evaluation of the derived features to our proposal. The number of features is shown in Figure 8.23 for selected window sizes. The complexity of the Features for a selection is shown in Figure 8.24.

The number of features extracted does not vary a lot with respect to the window size. We believe that this step merges similar regions that are created by Vaarandi's algorithm for various window sizes. While Vaarandi is more sensitive to the choice of w the clustering step evens out outliers created in the previous step.

The spread of the complexity of the features tends to decline consistently when more data is taken into account. This is a side-effect of our implementation of Vaarandi that was discussed in the previous section. Because the sensitivity of our implementation is adjusted for the input-size the features become more coarse when more data is taken into account.

In summary, linkage-clustering incurs substantial overhead. Therefore our approach aims to reduce the input size of the features to this step as much as possible. We have shown that after applying the filtering steps the overhead generated by this step is minimal. The related work by Fu *et al.* that applies this step directly to the filtered log records (*see* Section 8.2.2) suffers from prohibitively high overhead. We have shown with a very small sample set how overhead affects the scalability of this approach.



(a) $w = 600$ (b) $w = 1200$

(c) $w = 2400$ (d) $w = 3600$

Figure 8.23: Failure data: Number of features of single-linkage clustering (our approach)

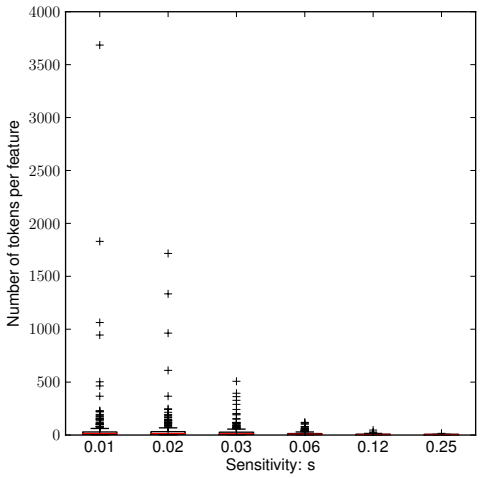


Figure 8.24: Failure data: Complexity of single-linkage clustering (our approach, $w = 1200$, $t = 0.4$)

8.2.5 Encoding of Clusters

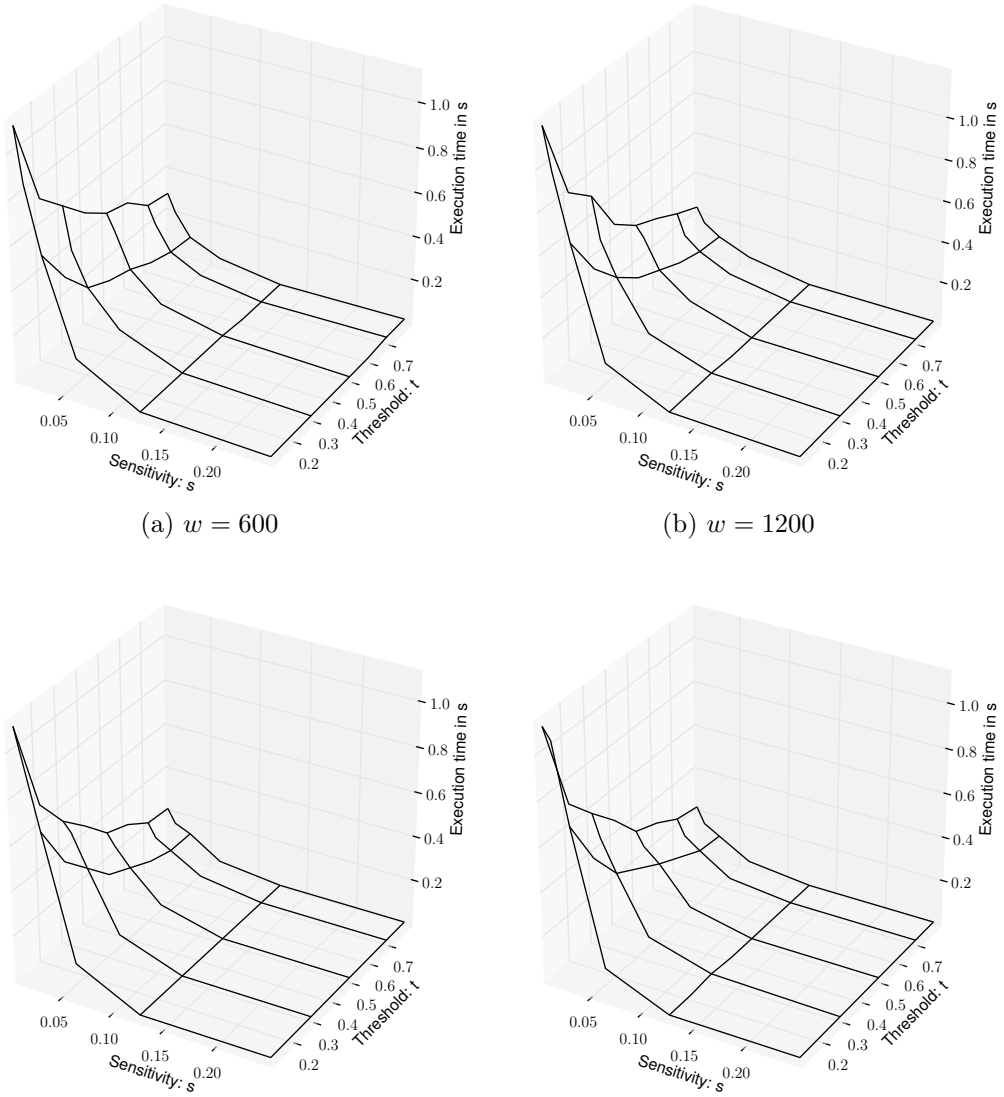
After having described how the individual features are derived from the unstructured log records, we describe in this section how the features are encoded. When comparing Figure 8.24 and Figure 8.23, it can be seen that the derived number of features is relatively small but the features still expose a high complexity. After having clustered the outcomes of Vaarandi's algorithm, the features of the single-linkage clusters are individual similar regions.

From a user perspective these features are hard to understand and matching the clusters against raw log records may be expensive, because every element of the cluster needs to be evaluated. In order to reduce the complexity of the clusters we encode each cluster using its longest-common subsequence. Because the elements of the clusters are very similar the cluster can be approximated in place by its longest-common subsequences (LCS). As discussed in Chapter 2.4.3, the LCS problem is in general NP-complete with respect to the size of the clusters. Hence we moved this step to the end of our processing pipeline. Given the small number of clusters that remain from the previous step the overhead is actually

expected to be small, which is confirmed by the results shown in Figure 8.25. Encoding more but smaller clusters (i.e., a low value of t) surprisingly has a bigger impact than encoding a small number of large clusters. We believe the assumptions made with respect to the properties of our LCS approach are confirmed, such that the similar nature of the elements inside the cluster reduces the processing time substantially, compared to naïve approaches [142] that do not exploit properties, such as similarity from the input data. The reader should note that the encoding is performed sequentially for the individual clusters in our implementation. We believe a parallel implementation as proposed in Section 5.2 may reduce the execution time even further. We regard that aspect as part of future work.

The number of encoded clusters, shown in Figure 8.26 is consistent with the number of clusters shown in Figure 8.23 as we expected. The reader should note that our implementation of the LCS includes heuristic optimizations. We do not compute the LCS for clusters in which all elements have less than or equal to two tokens. Such clusters are split up and each cluster element is regarded as a separate sequence. We also ignore any remaining cluster that has an LCS of less than two elements. This optimization is geared towards avoiding side-effects of the Levenshtein distance used in the previous step. Because our implementation normalizes the absolute value of the Levenshtein distance by the longest string in comparison, a short length can cause major distortions because for small numbers the steps of the quantization become larger.

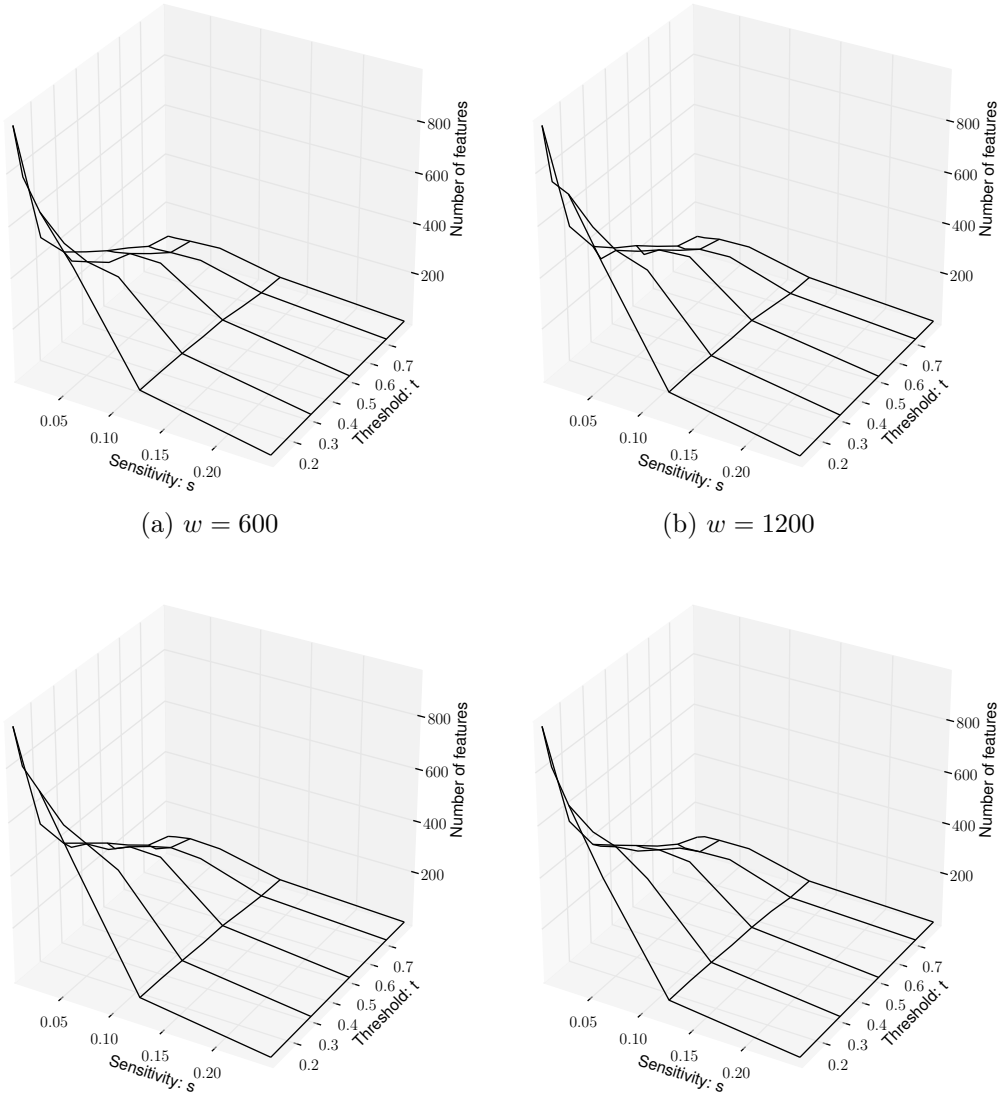
The complexity figures reflect that our objective to reduce the complexity of the clusters is fulfilled. In comparison with Figure 8.24, the complexity of the encoded clusters in Figure 8.27 shows that the number of tokens per feature have been reduced substantially. This encoding step makes the features coarser than related work or the outcomes of previous steps, but also reduces the potential that our approach may be prone to over-fitting. We will evaluate in the following sections how the features derived during these steps perform for symptom learning and symptom matching.



(a) $w = 600$ (b) $w = 1200$

(c) $w = 2400$ (d) $w = 3600$

Figure 8.25: Failure data: Execution times of encoding clusters



(a) $w = 600$ (b) $w = 1200$

(c) $w = 2400$ (d) $w = 3600$

Figure 8.26: Failure data: Number of encoded clusters

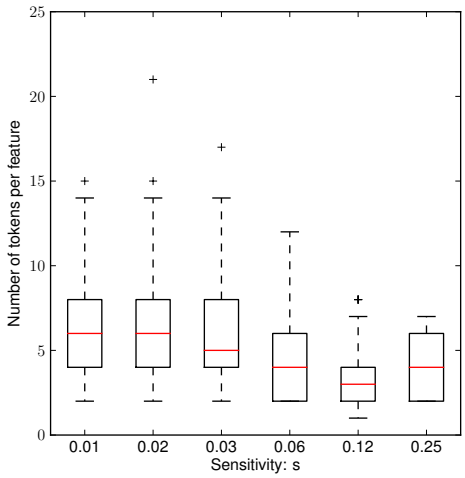


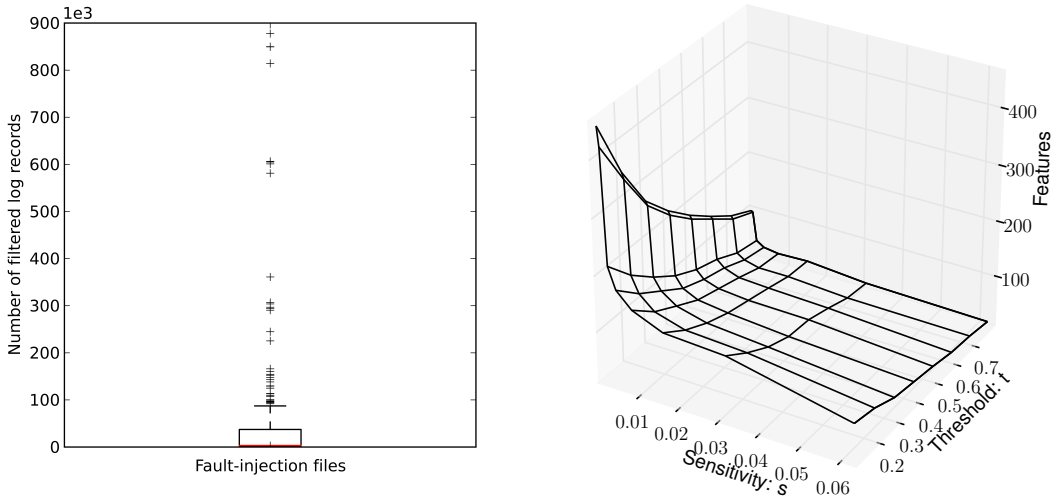
Figure 8.27: Failure data: Complexity of encoded clusters ($w = 1200$, $t = 0.4$)

8.2.6 Fault-Injection Data-Set

In this section we describe the results from the fault-injection experiments. The data of this sample consist of 315 log files, each of which is associated with a single fault label out of nine possible recurrent faults. The reader should note that one objective for the fault injection experiments is to show how our approach performs in the context of perceptual aliasing. Therefore, we injected faults into the test-bed that are known to have no manifestation in the log files collected. Furthermore, the information that is recorded by WebSphere is much more verbose as such the data set is larger in size, approximately 40GB, than the BlueGene/L data set and contains a richer set of features. Preliminary filtering returned 55858 distinct records and took 519s in total.

The distribution of the number of filtered log records per file is shown in Figure 8.28a and the number of features with respect to the threshold on the Levenshtein distance t and the sensitivity of Vaarandi's algorithm s is shown in Figure 8.28b. Each of these files spans 60 minutes, which is equivalent to the chosen window size $w = 3600s$.

The median of the file length is 3182 lines. Using the heuristic, shown in equation 5.3, the parameter s should be limited to be at minimum $s = 0.0003$. That value would extract



(a) Length of samples

(b) Number of encoded features

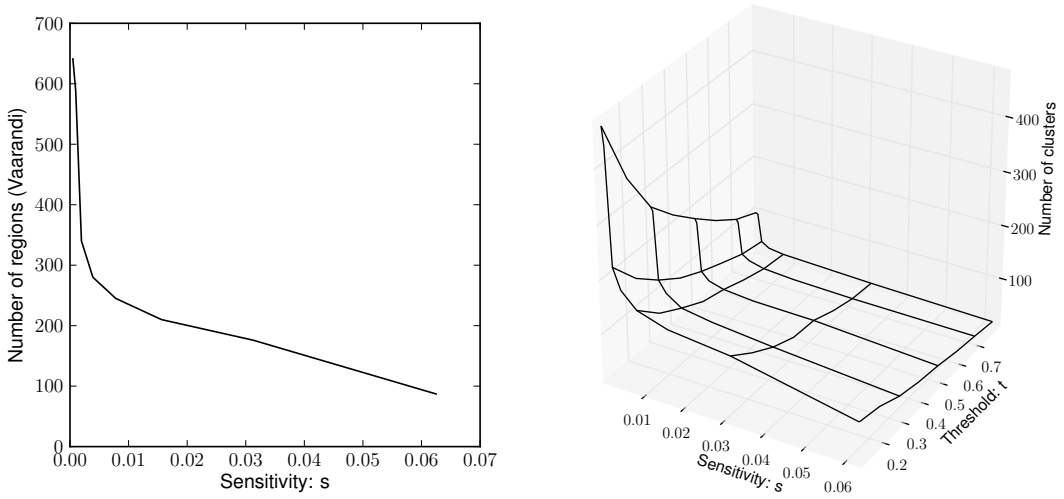
Figure 8.28: Fault-injection: The distribution of file lengths and the number of features

most records from files that have an equal amount or less filtered records than the median as independent regions of Vaarandi's algorithm. For the remainder of the evaluation, we have chosen to evaluate $s \in [0.0005, 0.0625]$.

The number of regions obtained using Vaarandi's algorithm and the number of clusters obtained using the single-linkage approach are shown in Figure 8.29.

It can be seen that the number of elements to process gradually decreases in each step. So the objective of reducing the input-size of the increasingly more expensive processing stages is attained. It can also be seen that the steepest decline in the number of features is near the self-imposed limit on s that we introduced in Equation 5.3, as shown in Figure 8.29a. For the fault injection sample the threshold for the single-linkage clustering results in $t = 0.450 \pm 0.003$ with a confidence of 0.95.

The execution times for the individual steps are shown in Figure 8.30. The reader should note that execution times for the filtering and Vaarandi's algorithms are sums of the execution times for applying the respective transformation to the individual files. The execution is stateless and could occur in parallel. For example, in a distributed monitoring

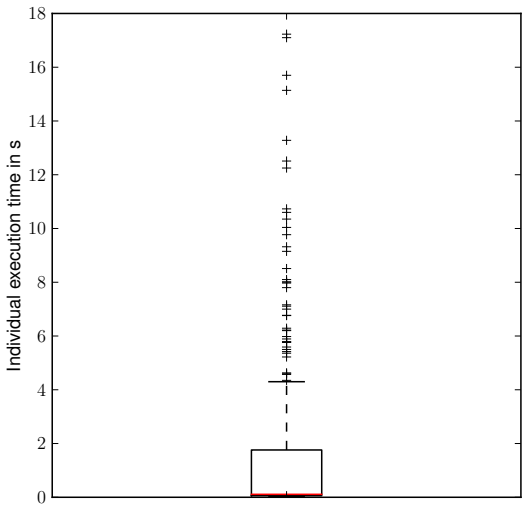


(a) Number of regions (b) Number of clusters

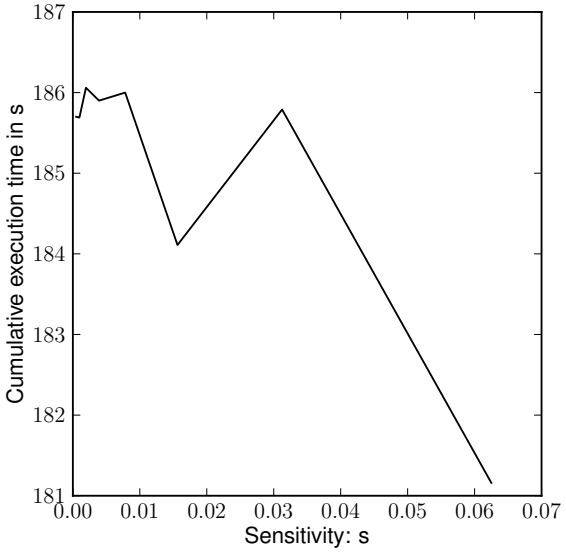
Figure 8.29: Fault-injection: The distribution of file lengths and the number of features

solution the filtering and Vaarandi's algorithm could be applied directly on the monitored node on regular intervals (i.e., of the size w). As shown this would substantially reduce the amount of monitoring data to be transmitted for further fault-diagnosis. The median execution time for the filtering is 0.11s and the cumulative execution time was 519.14s. The median execution time for Vaarandi's algorithm is 0.08s for all evaluated variations of s.

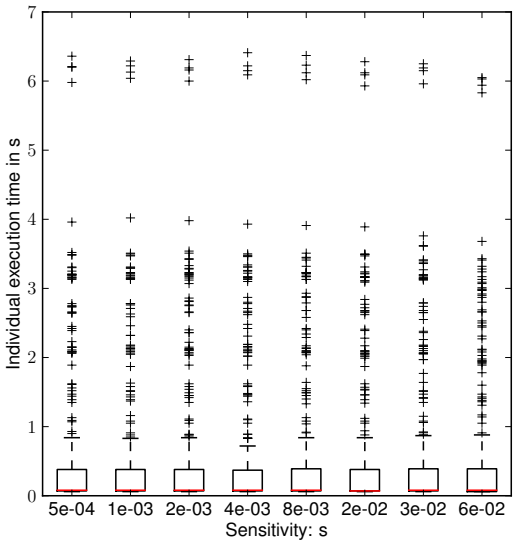
In Figure 8.31 we show the execution times of the clustering and the encoding step. Selecting the unique outcomes of the Vaarandi step is part of the execution time of the clustering step. The overhead of the clustering increases substantially closer to smaller values of s because more clusters are created. This should highlight the importance of the heuristic limit proposed in Equation 5.3. The influence of s is similar in the encoding step as well. Higher values of s result in more clusters that need to be encoded. As the threshold t increases the size of the individual clusters increases, which increases the overhead on the clustering algorithm. However, the increase in t reduces the number of clusters that need to be encoded reducing the execution time of the encoding.



(a) Filtering individual execution time

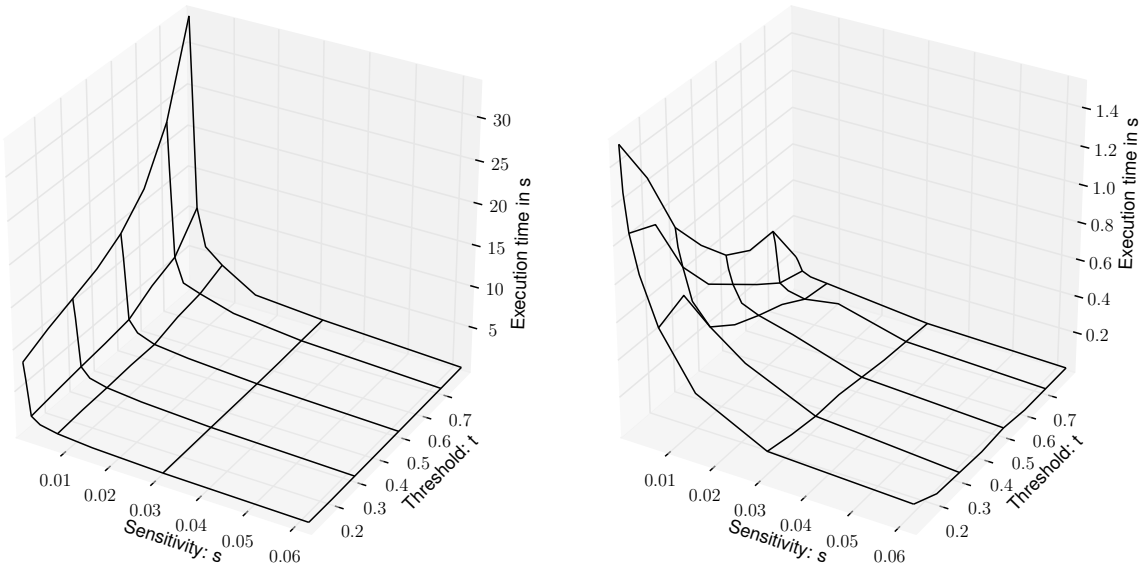


(b) Vaarandi cumulative execution time



(c) Vaarandi individual execution time

Figure 8.30: Fault-injection: Execution times of filtering and Vaarandi's algorithm



(a) Clustering execution time

(b) Encoding execution time

Figure 8.31: Fault-injection: Execution times of clustering and encoding

	Fu *et al.*	Jiang *et al.*	Vaarandi ($s = 0.125$) & Our Appr.
Filtered records	4854 (t = 22s)		
Density clusters	N/A	132 (t = 28s)	80 (t = 29s)
Linkage-clusters	$\emptyset$	N/A	54 (t = 0.09s)
Encoded features	N/A	N/A	53 (t = 0.01s)
Total features	$\emptyset$	132 (t = 50s)	53 (t = 51.1s)

Table 8.6: BlueGene/L: End-to-end comparison of log modelling approaches

The number of features that have been extracted for Jiang *et al.* using this data set was 117. The execution time of their approach was 480s. The reader should note that this approach executes against all log files of the sample at once and does not offer the option of parallel execution. We have used the same filtered samples as input to Jiang *et al.* that we used for Vaarandi's algorithm.

8.2.7 Discussion and End-to-End Comparison of Log Modelling

In the previous sections we analyzed the properties of our approach to modelling log files. We have shown that our approach scales sufficiently well with respect to the number of raw log data to model. In particular we have shown that despite having individual steps that have a high algorithmic complexity, we can process large amounts of data because we pipelined the steps with increasing complexity.

We also compared the steps of our processing pipeline conceptually to other approaches. In Table 8.6 we summarize the outcomes again for the failure sample of $w = 3600$ for the individual approaches. In particular the comparison with Fu *et al.* [34] was disappointing. Although the authors developed a model that may derive meaningful features for logs, the approach incurs prohibitive overhead when faced with a large input-sample, like the data that we obtained from the BlueGene/L data set. We have exemplified in Figure 8.22 how their approach scales for a very small selection of log-records from our sample.

The reader should note that the execution times displayed in Table 8.6 refers to the aggregate execution time of the individual steps and provides a pessimistic estimate. The

reader should note that in practice that the filtering and the density clustering can be executed in a parallel pipeline. Only the linkage-clustering depends on having the complete set of density clusters. Furthermore, the encoding step can be parallelised as well. The encoding only depends on individual clusters and does not need the entire set of clusters to be present at the same time.

The observations made for the BlueGene/L data set are consistent with the observation we made for the fault-injection data set. We have decided to use real data of the BlueGene/L data set to evaluate the properties of our approach in detail rather than limiting the evaluation to injected faults. The reader should note that the fault-injection data set is larger, approx. $40GB$ (approximately 57 times bigger than the BlueGene data set) and the preliminary filtering yields 55858 distinct data points. That is due to stack traces and other forms of recovery code log into the trace-log file. The reader should note that the size varies from experiment to experiment substantially. Some injected faults did not trigger any recovery code that logs into the file, while others create substantial bursts of logging activity.

In following sections we will refer to the outcomes of the log modelling evaluation and use them to learn symptoms of recurrent faults.

8.3 Learning Symptoms of Recurrent Faults

In this section we describe our evaluation of our model with respect to symptoms of recurrent faults. After having evaluated the extraction of features from the log files in Section 8.2, we evaluate the learning of symptoms of recurrent faults in this section.

Our primary objective is the evaluation of the quality of the features with respect to the classification of recurrent faults classification. We furthermore want to analyze, if our or related approaches are prone to over-fitting. Our evaluation methodology is to use 10-fold cross-validation. For classification we use Weka C4.5 classifier [139] using the default parameters. In order to measure the classification performance we have chosen the following metrics for our evaluation:

1. The classification error across all classes,
2. The redistribution error across all classes,
3. The precision,
4. The recall, and
5. The f-measure.

We will first describe the evaluation methodology of this in detail and then present actual measures in the following sections.

8.3.1 Methodology

Our methodology is to evaluate the classification performance using cross-validation. Cross-validation is a form of holdout classification that divides the initial sample is divided into k-folds that contain approximately n/k samples that are randomly selected with displacement of the original n samples. See Figure 8.32 for illustration. Once the folds are established, $k-1$ folds are used as the training set and 1 fold as the holdout to assess the performance of the classifier. The process is repeated over k steps. In each step a different fold is selected as the holdout. After k steps, the union of individual holdouts comprises the original sample.

A classifier is trained with the $k-1$ folds of the training set, then this classifier is used to classify the holdout samples. The metrics for the classification performance are measured from the holdout and the training set. After having described the evaluation methodology, we describe the quantitative performance measures and their interpretation. First we begin by introducing basic classification measures.

True positive (TP), true negative (FN), false positives (FP), and false negatives (FN), are the four different possible outcomes of a single prediction. In a single-class prediction these four outcomes are the only outcomes as shown in Table 8.7.

In the next section we take this illustration and define meaningful classification performance metrics.

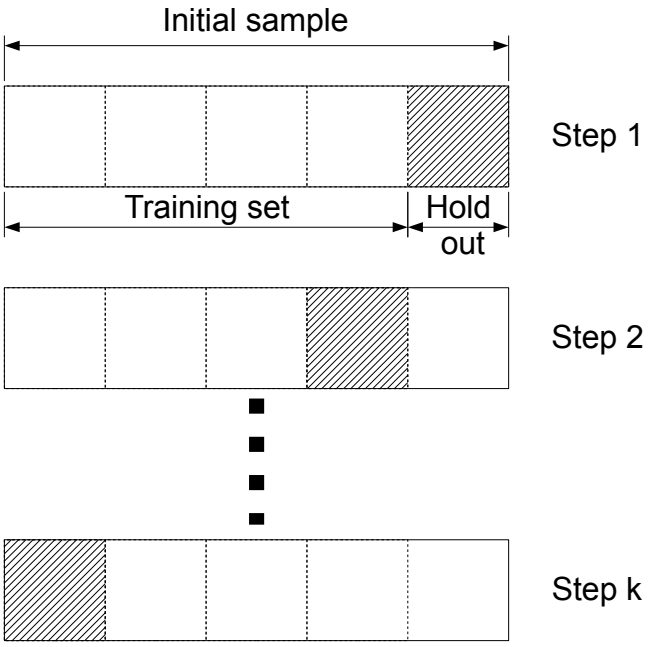


Figure 8.32: The concept of cross-validation

		Predicted class	
		Yes	No
Actual class	Yes	TP	FN
	No	FP	TN

Table 8.7: Illustration of outcomes: true positive, false negative, false positive, and true negative

		Predicted class		
		C_1	...	C_n
Actual	C_1	TP(C_1)	...	FN(C_1)
class	...	⋮	⋱	⋮
	C_n	FP(C_1)	...	TN(C_1)

Table 8.8: Confusion matrix with respect to outcomes

Confusion Matrix and Related Performance Measures

In a multi-class scenario (i.e., more than two predicted outcomes) the concept illustrated in Table 8.7 is generalized to represent aggregates of outcomes of a classification. These outcomes can be represented as a matrix, with one dimension representing the ground-truth class labels of outcomes and another dimension representing the outcome attributed by the classifier. This concept is illustrated in Table 8.8. Each matrix element at position (i, j) represents the number of outcomes of all classified samples that have the true class C_j that have been classified as the class C_i. If a total number of l classes exist and the classified outcomes are recorded as the confusion matrix $\mathbf{M}$, the true positives, false negatives, false positives, and true negatives can be computed as shown in Equations 8.2 for each class C_i. Table 8.8 illustrates the computation for C_1.

$$\begin{aligned} \mathrm{TP}(C_i) &= \mathbf{M}(i,i) \qquad (8.1) \\ \mathrm{FN}(C_i) &= \sum_{k=0 \wedge k \neq i}^{l} \mathbf{M}(k,i) \\ \mathrm{FP}(C_i) &= \sum_{k=0 \wedge k \neq i}^{l} \mathbf{M}(i,k) \\ \mathrm{TN}(C_i) &= \sum_{k=0 \wedge k \neq i}^{l} \sum_{j=0 \wedge j \neq i}^{l} \mathbf{M}(j,k) \qquad (8.2) \end{aligned}$$

These basic counters form the key building blocks for classification performance metrics we are using in our evaluation. We use precision ($\mathrm{Prec}(C_i)$), recall ($\mathrm{Rec}(C_i)$), and F-

measure (FM(C_i)) for each individual class C_i. The metrics are derived in Equations 8.3.

$$\begin{aligned} \text{Prec}(C_i) &= \frac{\text{TP}(C_i)}{\text{TP}(C_i) + \text{FP}(C_i)} \\ \text{Rec}(C_i) &= \frac{\text{TP}(C_i)}{\text{TP}(C_i) + \text{FN}(C_i)} \\ \text{FM}(C_i) &= 2 \cdot \frac{\text{Prec}(C_i) \cdot \text{Rec}(C_i)}{\text{Prec}(C_i) + \text{Rec}(C_i)} \end{aligned} \tag{8.3}$$

$$\tag{8.4}$$

Because all of the above metrics only yield meaningful results with respect to individual classes we need to derive metrics to measure the performance of a multi-class model as a whole. It is common practice to produce weighted-averages of the class-specific measures previously stated. We illustrate this for recall as follows. In essence average recall (Rec) is the sum of the recall metrics of the individual classes that are weighted by the number of true occurrences in that class.

$$\begin{aligned} \text{Rec} &= \sum_{i=1}^{l} \frac{\#(C_i)}{l} \text{Rec}(C_i) \\ \#(C_i) &= \sum_{k=0}^{l} \mathbf{M}(k, i) = \text{TP}(C_i) + \text{FN}(C_i) \end{aligned}$$

Furthermore, a generic metric for the performance of a classification performance is the classification error CE.

The classification error is the ratio of all outcomes outside the central diagonal of the confusion matrix to all outcomes. Since the elements on the central diagonal of the confusion matrix represent the true positives of the individual classes this metric can be computed by inversion as shown in Equation 8.5.

$$\text{CE} = 1 - \frac{\sum_{i=0}^{l} TP(C_i)}{\sum_{i=0}^{l} \sum_{j=0}^{l} \mathbf{M}(i, j)} \tag{8.5}$$

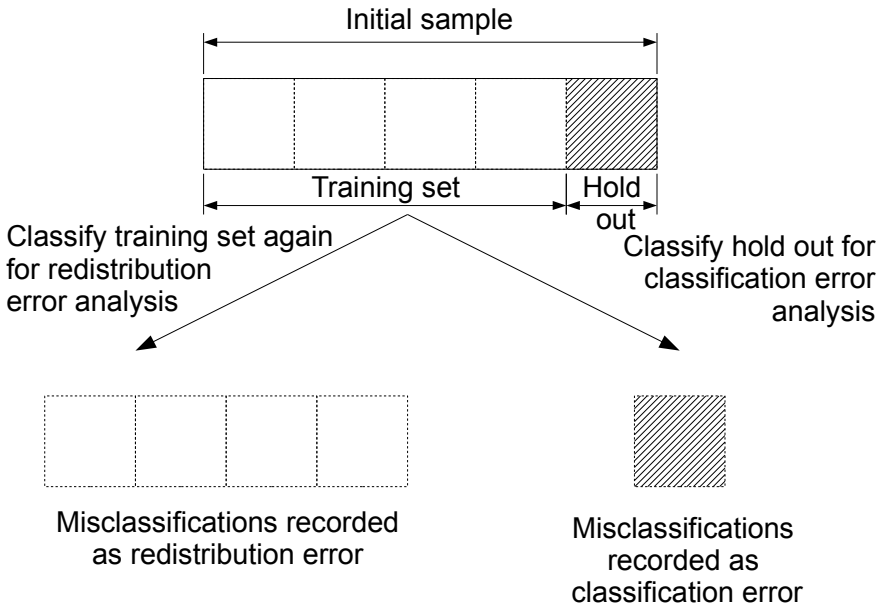


Figure 8.33: Redistribution- and classification error assessment

Classification and Redistribution

Now after having introduced performance metrics for the classification we describe how the classifier is trained and how outcomes are selected. Only deriving performance metrics from the holdout classification limits the evaluation to the trained classifier with respect to new data. In order to identify and highlight issues that may be associated with the classifier itself we also classify the training set and retain a separate set of performance metrics. In particular, when comparing how the trained classifier classifies the training set to how the classifier classifies the holdout one can assess if the trained classifier tends to over fit the data. A heuristic that captures this behaviour is the comparison of redistribution and classification error, shown in Figure 8.33. The reader should note that classification and redistribution refers to the same metric, shown in Equation 8.5, but the metric is derived from different data sets. If the redistribution error is significantly lower than the classification error, which means that the trained classifier is better at classifying

known samples, it is a symptom of over-fitting. The reader should note that assessing an over-fitting model is not absolute; instead, it is necessary to compare a model to other models [44]. In our case our model and related work use the same type of classifier but tend vary in the types of features used. While Hawkins was primarily concerned with the complexity of the classifier (as in the number of tunable parameters of the classifier) our evaluation varies in the number if input features available. We expect our approach to be less prone to over-fitting than related work because we can cope with a relatively small set of simple features.

Summary

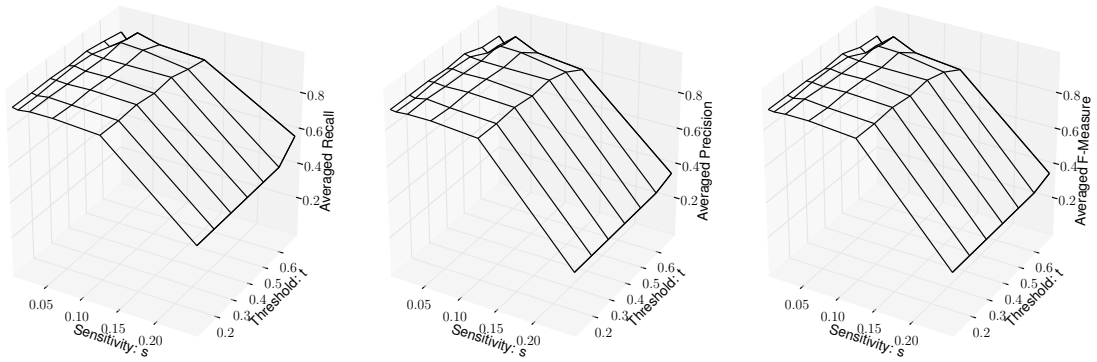
So far we have described our evaluation methodology and related performance metrics. Instead of just assessing the predictive performance using standard measures such as recall, precision, and F-measure, we also compare our approach with respect to assessing the impact of over-fitting. In the next two sections we show how these metrics are derived from the BlueGene/L sample, our fault injection experiments and how our approach compares to related proposals.

8.3.2 Classification Performance

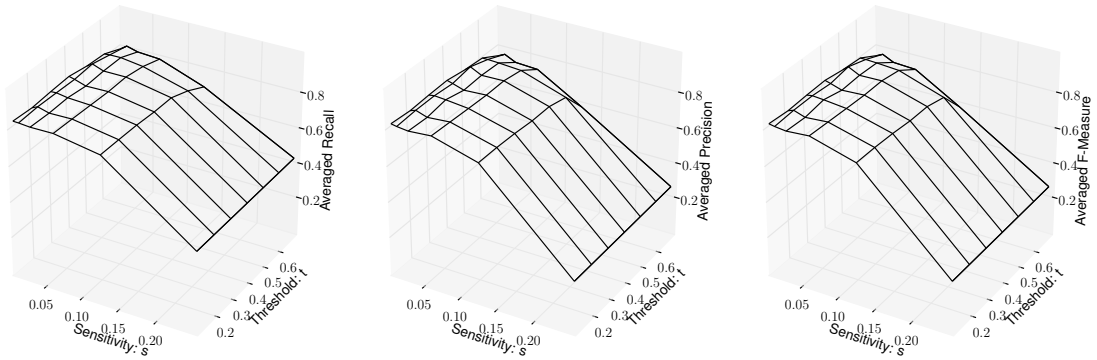
We now compare the performance of our approach to the proposals by Vaarandi [140], Fu [34], and Jiang [64]. We first present our results obtained from the BlueGene/L data and then show the performance of the fault injection experiments.

BlueGene/L Data-Set

The data used for this step is the data evaluated in Section 8.2.5 for our approach with variable window sizes, the data evaluated in Section 8.2.3 for Jiang *et al.* [64] and Vaarandi [140]. We have established to use $s = 0.002$ for the density-based clustering and $t = 0.4$ for the threshold of the linkage-clustering. In this section we evaluate the approaches across a



(a) Recall $w = 1200$ (b) Precision $w = 1200$ (c) F-measure $w = 1200$



(d) Recall $w = 3600$ (e) Precision $w = 3600$ (f) F-measure $w = 3600$

Figure 8.34: BlueGene/L: Aggregate classification performance (our approach)

range of these parameters and show that the heuristics we described previously are adequate. The Figures 8.34 shows the weighted average recall, precision, and F-measure of the classification for selected window sizes using our approach.

The precision and recall appear to be relatively insensitive to s for small values of t. It shows that the classification performance remains high, despite a reduction in the complexity and the number of features (*see* Section 8.2). While the classification performance is relatively stable across the parameter space the reader is reminded of the evaluation of the log model in Section 8.2.5, in which we show for the same data set that increased values for t and s substantially reduce the number of features.

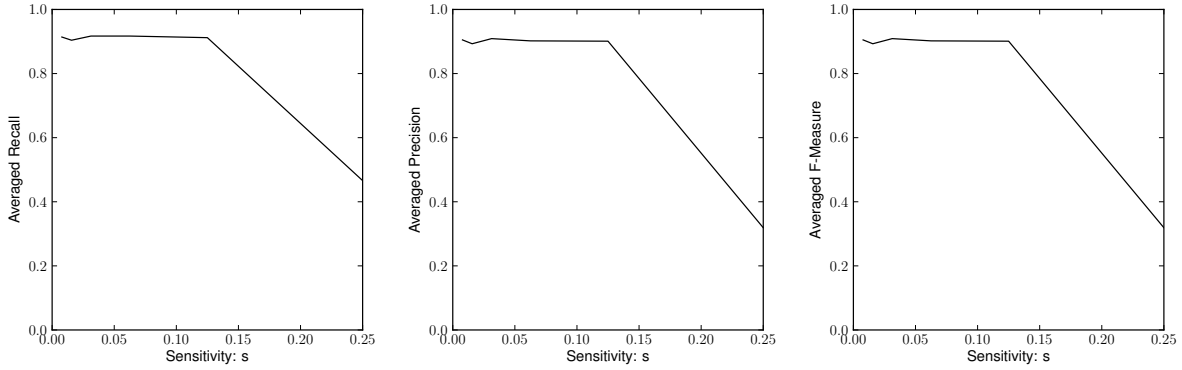
The automatically determined value of $t = 0.43 \pm 0.01$ (*see* Section 8.2.4) lies within an acceptable range of recall and precision for $s \leq 0.125$, confirming that our heuristic returns adequate values. Because the data model can be used to set up cross-validation experiments quickly due to its low overhead, we believe a feedback loop from the performance metrics of the cross-validation to the data model can be established to determine s heuristically as well. We regard that problem as part of our future work.

Now we compare the outcomes of our approach to the proposals by Jiang *et al.* [64] and Vaarandi [140]. We have show the results for Jiang *et al.* for various window sizes in Table 8.9. Compared to our approach the proposal by Jiang *et al.* has a similar classification performance. For $s = 0.125$ and $t = 0.4$ we can achieve approximately the same recall and precision for $w >= 1200$s using less features. For $w = 600$s our approach uses 85 features and has a recall of 0.921, a precision of 0.909 and an f-measure of 0.913, clearly outperforming the proposal by Jiang *et al.* It should further be noted that our features are much smaller and less complex than those of their proposal (*see* Section 8.2.3).

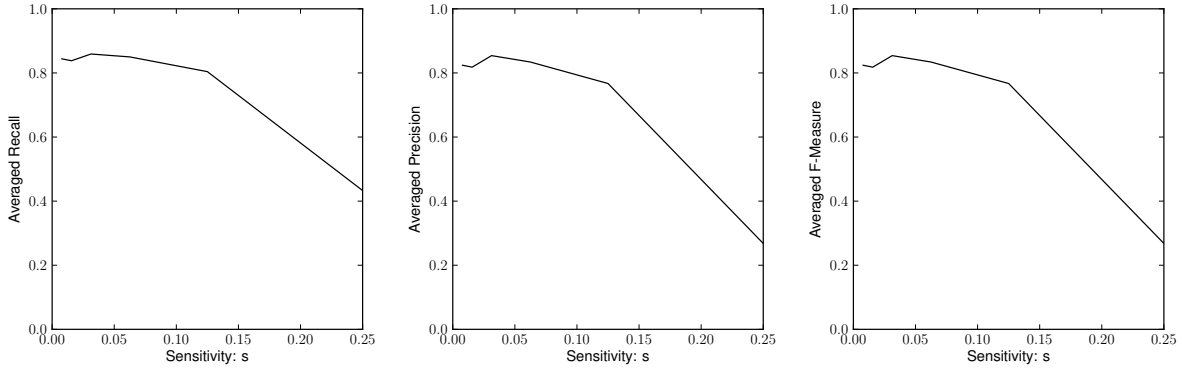
Window size in s	600	1200	1800	2400	3600
Features	32	129	132	132	132
Precision	0.586	0.848	0.83	0.813	0.75
Recall	0.661	0.868	0.852	0.834	0.787
F-measure	0.598	0.855	0.838	0.821	0.762
Classification error	0.33	0.13	0.15	0.16	0.21
Redistribution error	0.30	0.07	0.08	0.09	0.11

Table 8.9: Classification performance (Jiang *et al.*)

The results for Vaarandi's algorithm are shown in Figure 8.35. It should be noted by the reader that these are interim results of our approach because we made Vaarandi's algorithm part of our feature extraction. The results are in-line with those seen in Figure 8.34 for small values of t. However, our approach requires fewer features to attain the same classification performance.



(a) Recall ($w = 1200$) (b) Precision ($w = 1200$) (c) F-measure ($w = 1200$)



(d) Recall ($w = 3600$) (e) Precision ($w = 3600$) (f) F-measure ($w = 3600$)

Figure 8.35: BlueGene/L: Aggregate classification performance (Vaarandi's algorithm)

Fault Injection

In this section we evaluate our approach using the data analyzed in Section 8.2.6. The samples of the data set are individually labelled using the labels shown in Table 8.2. We compare our approach against the proposals by Jiang *et al.* and Vaarandi. Figure 8.36 shows the weighted average recall, precision, and F-measure of the classification of our approach.

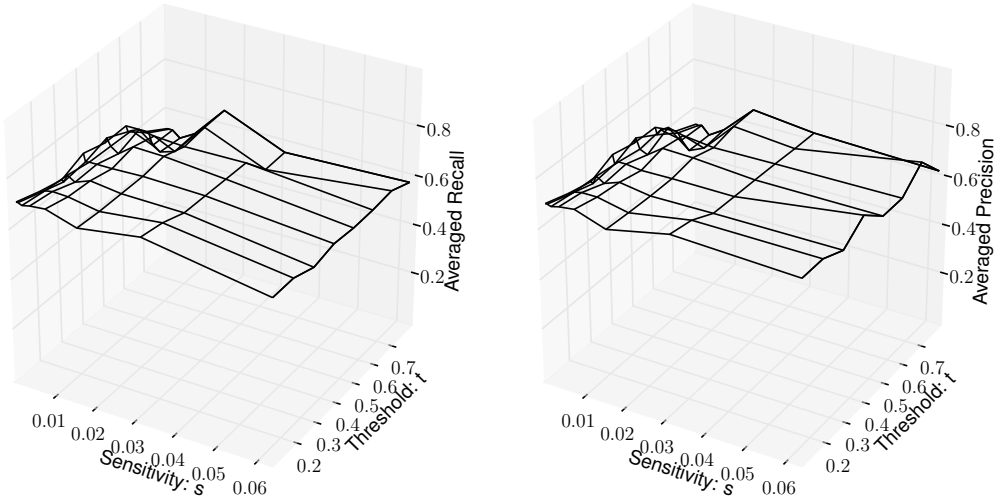
Throughout the chosen parameter space (i.e., s and t) our approach maintains a high classification performance. Precision, recall and, consequently, the f-measure are at about 0.70. For extreme values of t the measures start to drop. As shown in the chosen parameter space results in a large number of features in relation to the number of class labels. We expect that our approach might be over-fitting the training-set for most of those configurations. We analyze this problem in detail in Section 8.3.3.

Based on the features that have been extracted by Jiang *et al.*, we obtained an aggregate precision of 0.685, a recall of 0.594, and an f-measure of 0.596. Their model extracted a total of 117 features. By comparison with our results, by choosing $s = 0.0625$ and the heuristic estimate $t = 0.43$ we can achieve 0.727 precision, 0.616 recall, and 0.61 f-measure with just 45 features that have a substantially smaller complexity than those extracted by Jiang *et al.*

The results for Vaarandi's algorithm are shown in Figure 8.37. It should be noted by the reader that these are interim results of our approach because we made Vaarandi's algorithm part of our feature extraction. The observations made in the figures are therefore almost identical to those seen in Figure 8.36 for small values of t. Although Vaarandi's algorithm expectantly has a similar classification performance as our approach it depends on a larger set of features to attain that performance.

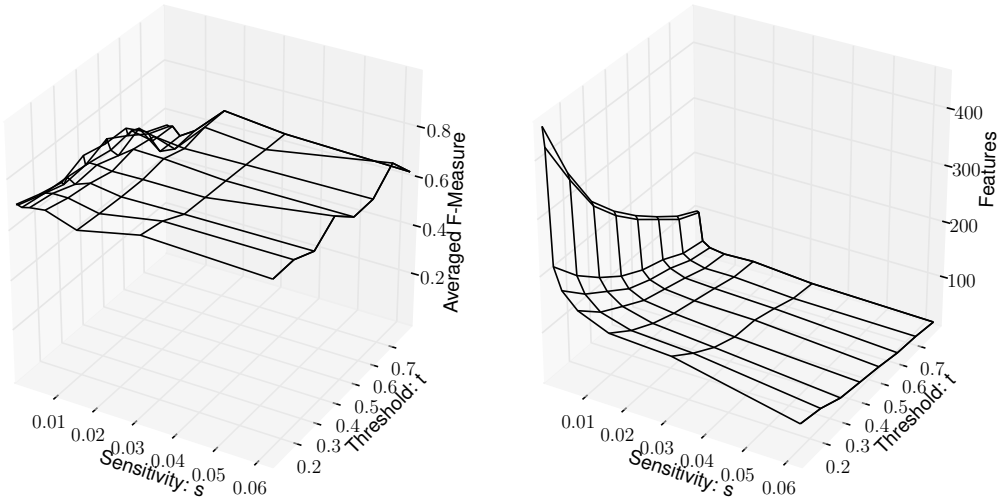
8.3.3 Over-Fitting

After having described the classification performance for the holdout classification, we show our results with respect to the possibility of over-fitting.



(a) Recall

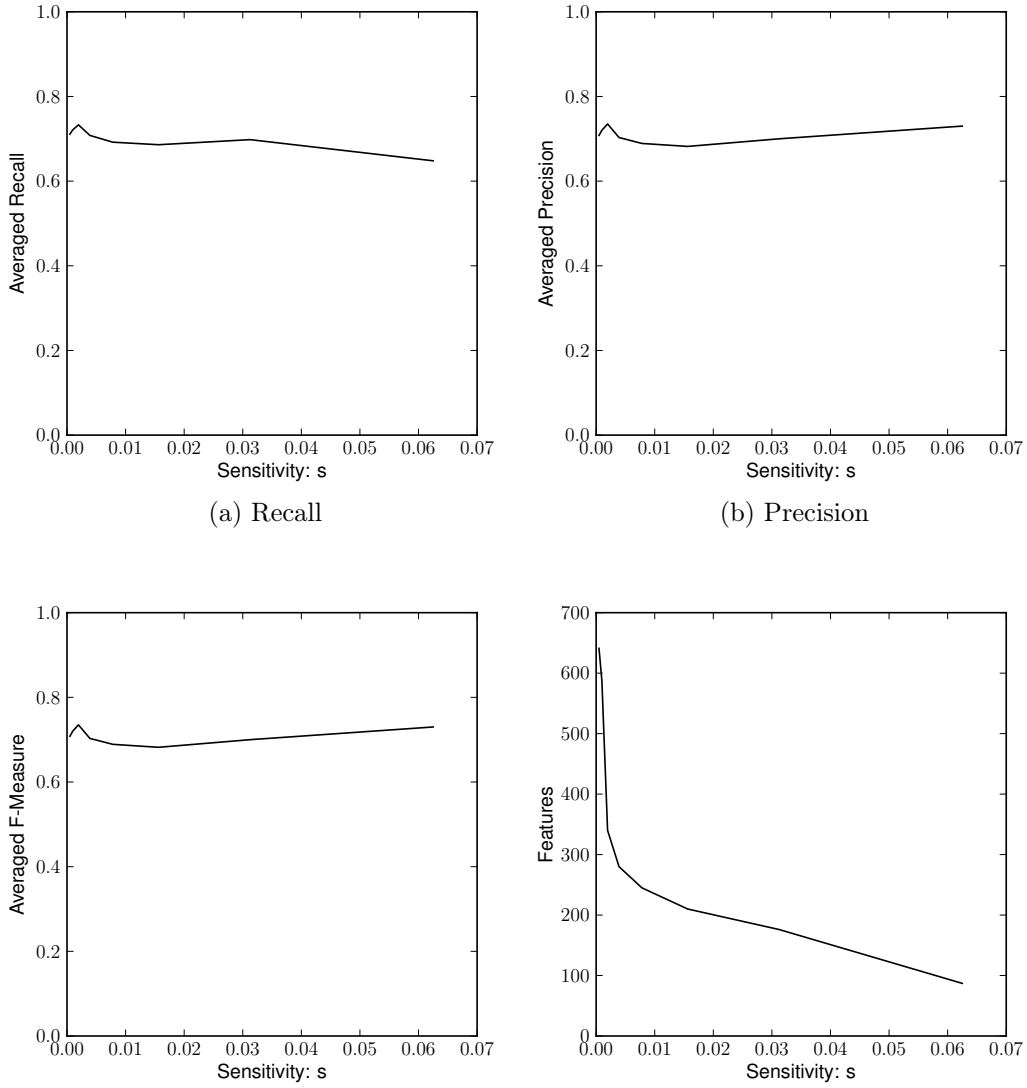
(b) Precision



(c) F-measure

(d) Number of Features

Figure 8.36: Fault-injection: Aggregate classification performance (our approach)



(a) Recall

(b) Precision

(c) F-measure

(d) Number of Features

Figure 8.37: Fault-injection: Aggregate classification performance (Vaarandi's algorithm)

BlueGene/L Data-Set

In Figure 8.38 we show the classification and redistribution error for our approach for selected window sizes. It can be seen that both measures increases with larger values of s and t, which results in fewer features being available for the classification. However, since there is little deviation among those two errors there is little evidence that the approach is over-fitting [44].

In Table 8.9 we already included the results of Jiang *et al.* for different window sizes. Although there is a small divergence between the classification and redistribution error, it is from our perspective not enough to attribute significant over-fitting as well. It should be noted that the error metrics of their approach tend to be higher than ours for $s \leq 0.125$.

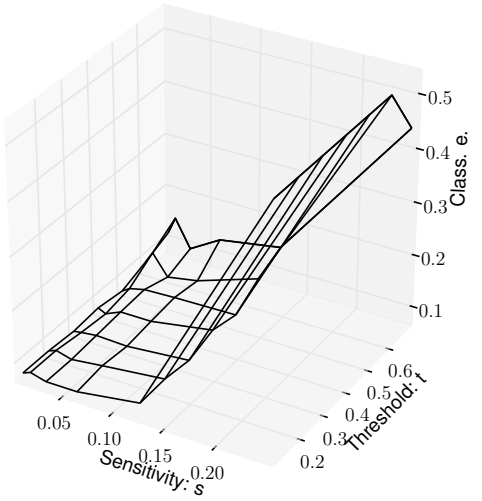
In Figure 8.39 we show the classification and redistribution error of Vaarandi for selected window sizes. For the BlueGene/L data set the approach shows no symptoms of over-fitting. The distribution and classification error are almost identical. We attribute it to the data set that consists of relatively small (i.e., number of monitoring records) individual samples.

Fault Injection

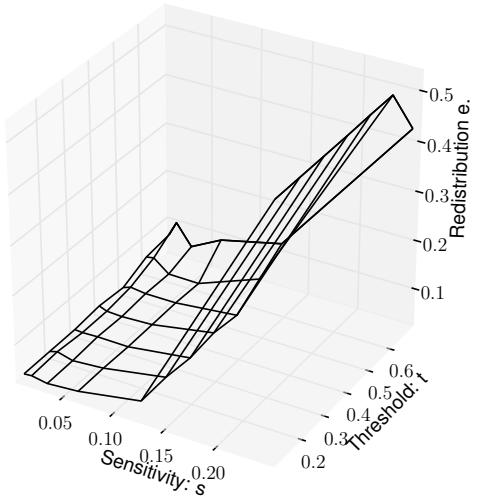
As we described in Section 8.3.2 our approach may be prone to over-fitting compared to related approaches. For small values of t and s we extract a large number of features that are used by the classification. We show the classification and redistribution error of our approach in Figure 8.40.

The reader should note that the classification error is not normalized by the number of instances as the performance measures described in the last section. It can be seen that the classification error increases substantially as the number of features decreases, however the classification error is in-line with the redistribution error. As a result our approach does not exhibit severe symptoms of over-fitting [44].

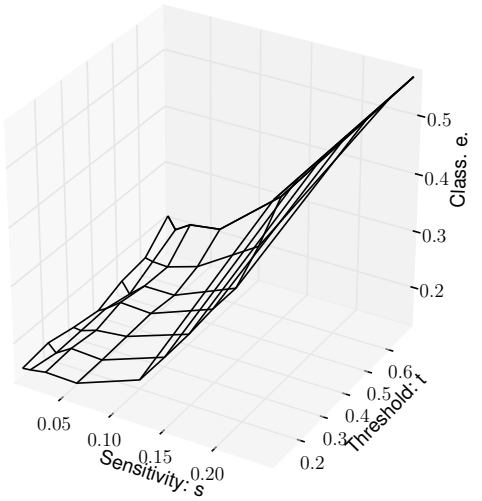
Based on the features that have been extracted by Jiang *et al.*, we obtained a classification error of 0.40 and a redistribution error of 0.35. Compared to our previous selection



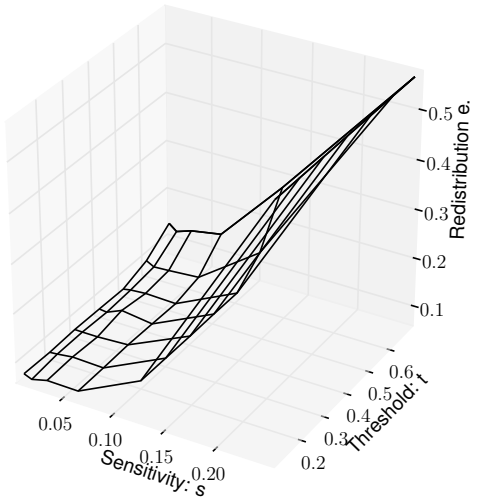
(a) Classification error ($w = 1200$)



(b) Redistribution error ($w = 1200$)

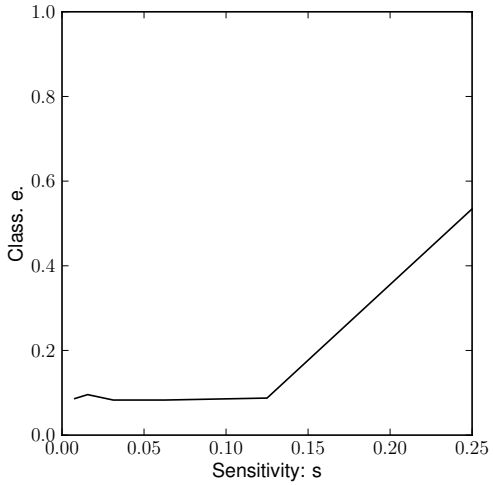


(c) Classification error ($w = 3600$)

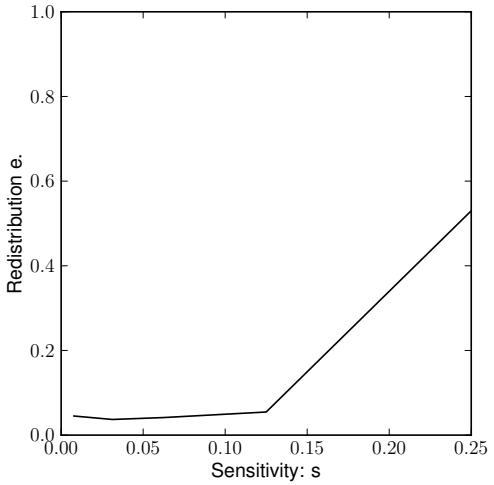


(d) Redistribution error ($w = 3600$)

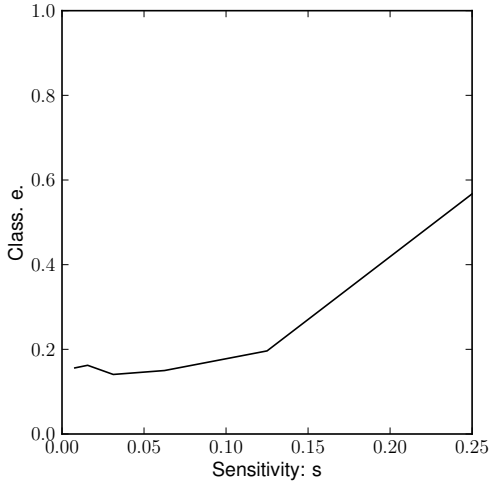
Figure 8.38: BlueGene/L: Classification and redistribution error (our approach)



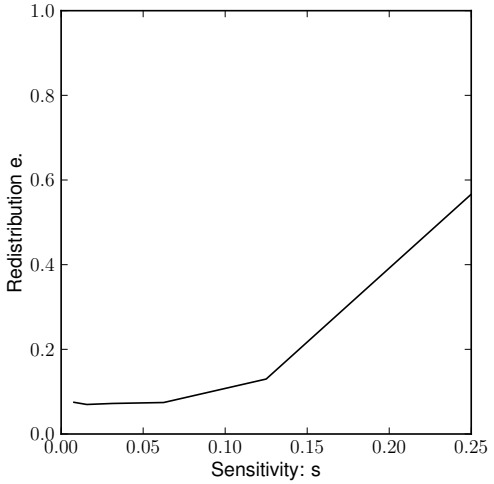
(a) Classification error ($w = 1200$)



(b) Redistribution error ($w = 1200$)

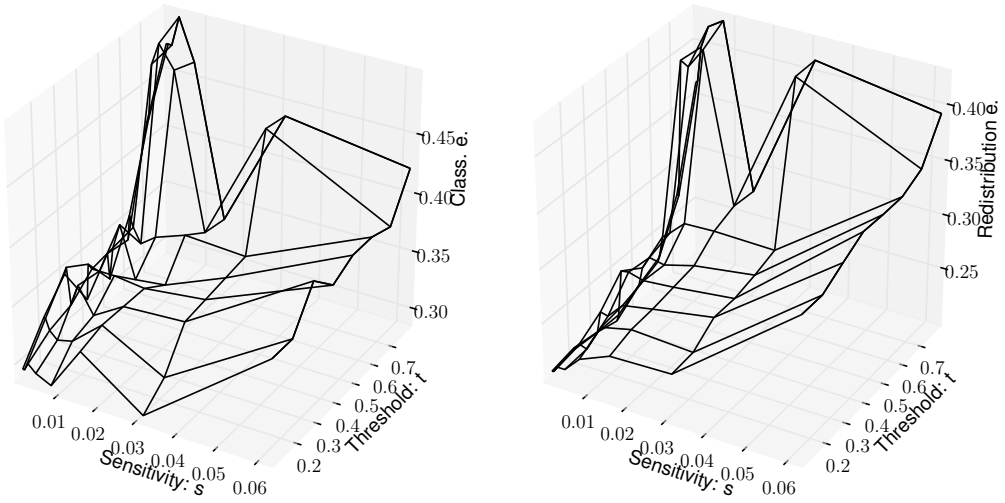


(c) Classification error ($w = 3600$)



(d) Redistribution error ($w = 3600$)

Figure 8.39: BlueGene/L: Classification and redistribution error (Vaarandi's algorithm)



(a) Classification error

(b) Redistribution error

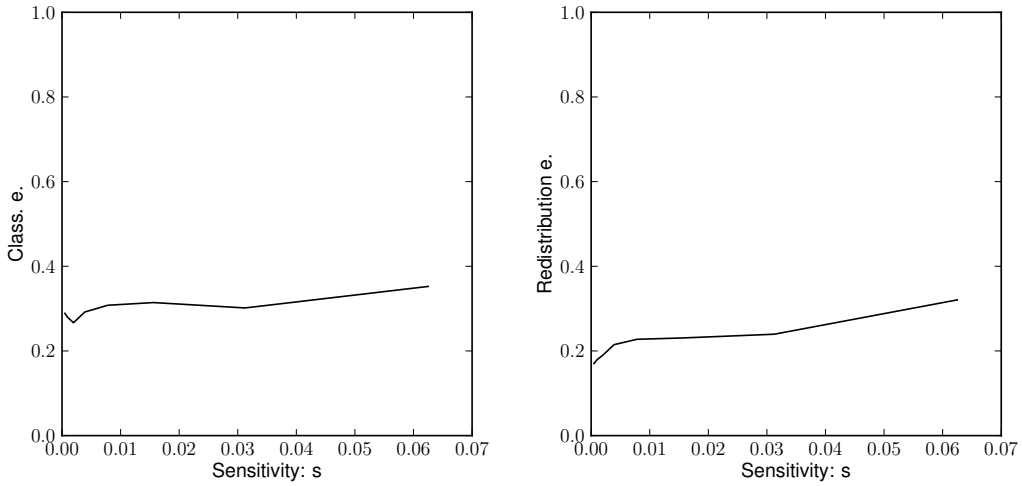
Figure 8.40: Fault-injection: Classification and redistribution error (our approach)

of $s = 0.0625$ and $t = 0.43$ we obtain a classification error of 0.38 and a redistribution error of 0.35. While our approach shows a smaller ratio of redistribution and classification error than the approach by Jiang *et al.*, we believe it is not significant to regard our approach as less over-fitting in general.

The results for Vaarandi's algorithm are shown in Figure 8.41. The ratio of classification and redistribution error does not indicate a significant degree of over-fitting. Compared to our approach Vaarandi's algorithm has a larger gap between those two measures than our proposal because of the very large feature set. That was a motivation to refine his proposal and develop the subsequent stages of our approach. As a result our approach can achieve the same classification performance with a much smaller feature set.

8.3.4 Summary

In this section we analyzed the classification performance of our log model with respect to extracting symptom signatures using the C4.5 classifier and matching them against folds



(a) Classification error

(b) Redistribution error

Figure 8.41: Fault-injection: Classification and redistribution error (Vaarandi's algorithm)

from a cross-validation.

As expected the classification performance is very sensitive to the parameters of the log. We have demonstrated that the heuristic approximation of t results in acceptable classification performance metrics. We consider it as part of our future work to develop a reliable estimate of s that may be based in-part on the classification performance metrics as well.

Since our and related approaches extract a relatively large feature set to model the individual samples we also analyzed our approach and related work with respect to over-fitting. Using traditional means [44] we cannot establish that our approach is significantly over-fitting or performing better in that regard than related work. To our surprise the metrics derived from the related approaches are of about the same order of magnitude as the over-fitting metrics derived from our approach. This is a surprise because the data model we propose extracts fewer and less complicated features than related work as we have shown in the previous section.

In the next section we show how a confusion matrix derived from the evaluation in

Section 8.3.2 can be used to create a recovery controller. For the fault-injection experiments we have deliberately chosen to inject faults that cannot be accurately detected using log files. For the BlueGene/2 data set the attribution of failure is inherently tied to the monitoring data, as such we think that the high classification metrics may only apply when we assume full visibility of state of the supercomputer from the log files.

8.4 Proactive Fault Diagnosis and Recovery

Having evaluated the performance of fault classification, we want to evaluate proactive approaches that are built on-top of the classification in this section. As the focus of this work is the evaluation of the discrete controller, we favour an evaluation environment, in which we have full control over all system parameters. Therefore, we evaluate the performance of the controller using discrete event simulation. As described in Chapter 7 this approach mey become unstable if these assumptions are not met.

8.4.1 Simulated System

In order to simulate a realistic environment we will obtain the simulation parameters, namely fault prevalence and the characteristics of symptoms of failure from fault-injection experiments conducted for unstructured data (*see* Section 8.3.2). The confusion matrix is shown in Table 8.10 and forms the basis for specifying the system parameters $\mathbf{\Omega}$ and $\mathbb{O}$.

Since our previous evaluation did not consider recovery we seed a simulation with this confusion matrix. We assume an error detection confidence of $p_e = 0.99$. We mock up recovery actions with their anticipated cost and success rate. We consider the recovery actions shown in Table 8.11.

8.4.2 Impact of Wrong System Specification

Our first objective is the analysis of the impact of system drift. We compare the drift scenarios against a simulation of optimal system specification. The differentiating measure

a	b	c	d	e	f	g	h	i	classified as
5	0	0	0	0	0	0	0	0	a = auth
0	53	0	0	1	0	6	0	0	b = busy
1	0	4	0	0	0	0	0	0	c = connpool
0	0	0	54	0	5	0	1	0	d = exception
0	3	0	8	38	0	11	0	0	e = null
0	5	0	0	0	30	0	0	0	f = remjsp
1	10	0	0	1	0	48	0	0	g = sleep
0	5	0	0	0	0	0	0	0	h = threadpool
0	17	0	0	0	0	0	0	8	i = undetectable/ dblock

Table 8.10: Example confusion matrix

is the total time to recovery (i.e., the sum of the time of the individual recovery actions). We consider four scenarios of a wrong system specification:

1. Ignoring the fault prevalence (avg. TTR = 129s),
2. Overestimating the confidence of recovery actions (avg. TTR = 146s),
3. Assuming deterministic observations (avg. TTR = 121s), and
4. The combination of the above (avg. TTR = 155s).

For each case we compute the POMDP and compute parameters based on the incorrect specification (*see* Chapter 7) and simulate it against the correct system specification. For the first case we initialize the belief state uniformly. Hence the first recovery action will not be chosen in accordance of the fault prevalence. In this configuration we expect the

Action	**Success Rate**	**Cost**	Faults
NOOP	$p_e = 0.99$	2 s	$\emptyset$
WAS Reboot	0.9	120 s	busy, sleep
WAS Reconfigure	0.9	30 s	auth, connpool, thdpool
J2EE Reinstall	0.9	30 s	exception, null, remjsp
DB2 Reboot	0.9	100 s	dblock

Table 8.11: Properties of simulated recovery actions

controller to optimize the recovery times of rare faults at the cost of prolonged recovery times for common faults. In the second case we set the simulated recovery success rates to 0.7 instead of 0.9 and specify a confidence of 0.9. This drift will make the controller more opportunistic in choosing the recovery actions. As a result it will wrongly assume the system to have recovered and hence prolong the recovery process. For the third case we assume deterministic observations and initialize the observation function as a diagonal matrix. This is expected to slow down the recovery process of aliased faults (i.e., faults that have multiple symptoms) because only one observation per fault will trigger the correct recovery action. Finally we consider a combination of all drift scenarios. The comparison of the total time to recovery is shown above. The average time-to-recovery for the optimal case in which we know all system parameters yields 121s.

Overestimating recovery action confidence has the biggest impact on our system. The results for deterministic observations are potentially caused by the system simulation of aliased faults. Since a new observation based on $\mathbf{\Omega}$ is generated at every simulation step the observations for aliased faults may fluctuate, and thus still produce the right observation that triggers a recovery over a finite number of steps. The reader should note that all faults could be recovered using the system specification in a finite amount of time.

8.5 Summary

In this chapter we have evaluated all aspects of the approach proposed in this thesis. We have demonstrated that the system model proposed for log files can be used to classify faults effectively using unstructured monitoring data. We evaluated our proposal against related work. We have shown that our approach performs as expected on data obtained from fault-injection experiments as well as failure data obtained from large distributed systems in the field.

For the model of discrete data proposed in Chapter 5 we have shown that it scales well against related proposals in the field. We were able to model and classify up to $40GB$ of raw log data (i.e., fault-injection data set) in a matter of minutes on a single machine. By

evaluating aspects of the model at the task-level we also highlighted areas that can improve the scalability of the approach even further because of the high degree of parallelism our model exposes. These claims were validated for the fault-injection data set as well as the BlueGene/L data set.

We have also shown that the features derived from the log-model can be used to extract and identify symptoms of recurrent faults from the fault-injection data set as well as the BlueGene/L data set. We can achieve the same (i.e., compared to Vaarandi [140]) or better (i.e., compared to Jiang *et al.* [64]) classification performance than related approaches in the field with a smaller number of features. The degree of over-fitting is in-line with other proposals [140].

For the fault-injection experiments that expose a high degree of perceptual aliasing, we have set up a simulated recovery controller to demonstrate the applicability of the proactive recovery approach proposed in Chapter 7. Under the assumption of full-controllability and full knowledge of side effects of recovery actions, we have shown that a system can be recovered in the presence of an uncertain fault-diagnosis. Minor misconceptions about fault prevalence and recovery actions do not have a major impact on the stability of the recovery process but can prolong the time to recovery. Although the simulated recovery looks promising, it requires in-depth system knowledge to work accurately. In practice faults may be transient or the system state may change due to conditions outside of the scope of the recovery model. In such scenarios the controller is likely to under perform. In addition, the knowledge of the transition function $\mathbf{T}$ as well as the observation function $\mathbf{\Omega}$ is often incomplete. We published a study [110] of decision learning addressing some of these drawbacks. The recovery approach should be viewed as optional addition to the other aspects of this thesis that can be used to mitigate the problem of perceptual aliasing in selected scenarios.

The reader should note that this thesis focuses on modelling discrete monitoring data and fault diagnosis based on that model. To that extend we have validated our claims.

Chapter 9

Discussion and Future Work

In this chapter we discuss our approach and the lessons learned from the evaluation. In this thesis we do not claim that log file analysis or the proactive approaches presented are suitable for all systems and faults.

In this thesis we have addressed the diagnosis of failures in information software systems using discrete monitoring data. Our focus is the diagnosis of recurrent faults as they account for a high percentage of the reported defects. So far the diagnosis of failures in these systems is still a manual, laborious process that provides limited support for automation. In this thesis we are offering a model that learns symptoms of recurrent faults automatically and is able to identify these symptoms. We also proposed an approach to recover from perceptually aliased faults when certain conditions are met.

We have shown in our evaluation that a range of common problems can be automatically diagnosed with no manual intervention. We validated the approach using extensive fault-injection experiments and real log data obtained from the BlueGene/L supercomputer. We have shown that we can learn and identify symptoms of recurrent faults for both data sets. We have also shown that our model for discrete monitoring data scales well compared to related work [34, 64, 140] and, unlike related work, exposes a high degree of parallelism.

For perceptually aliased faults, we have shown that we have shown through the use of simulation that we can construct a recovery process from the symptoms if the system is

fully controllable and the effects of the recovery actions are known.

In the next sections we describe various areas of improvement as part of future work.

9.1 Log Analysis Is Not Optimal for Error Detection

When we outlined the scope of this thesis in Section 1.2, we clearly assumed the presence of reliable error detection. Unfortunately, log data contains discrete events that usually reflect state changes of components of the system. Therefore, performance defects that have a slow build-up such as memory leaks, resource exhaustion and congestion are hard to detect promptly by analysing log data. These faults only trigger events and potentially log data when individual components have already entered a permanent state of failure. For example a memory leak in a JVM eventually triggers allocation failures that may be logged as exceptions in the log data.

In order to attain the assumption of reliable error detection, we propose to follow best practices and define service level objectives (SLO) for critical interfaces of the system (*see* Sturm *et al.* [132]). In production systems reliable error detection is essential to detect a state of failure automatically. In related work [58,59,61] we investigate several approaches to anomaly detection based on management and performance metrics that do not require explicit statements of SLOs.

9.2 Rare and Subtle Faults Are Hard to Diagnose

As we described in Section 2.3, discrete monitoring interfaces in general have a limited system coverage, which may limit the fault coverage too. Means to improve the fault coverage include the use of other monitoring interfaces such as trace information, performance metrics and proactive approaches. We have shown in related work [58, 59, 61] that a management-metric-based diagnosis approach can detect and diagnose a range of performance defects. Furthermore, we have shown that trace information can also be used to diagnose a range of subtle performance problems reliably [91, 108].

As we described in Chapter 7, faults having no manifestation in monitoring data could be seen as a special case of perceptual aliasing. One approach to cope with this scenario is to combine a set of dedicated probes that validate such faults with the discussed self-recovery model. In contrast to existing configuration and system audits, such probes may then be enabled adaptively to minimize the overhead for the system.

Our approach relies on historic monitoring data that contains samples of manifestations of faults. In order to establish a symptom of a fault we need a certain number of samples. We limited our scope (*see* Section 1.2) to recurrent faults.

We have shown that insufficient samples may make faults perceptually aliased. Using the proactive approach discussed in Chapter 7 one could dedicate probes for such faults, but this may increase processing overhead for rare faults. Based on analysis of PMR data such rare faults reported to IBM support usually result in manually encoded symptoms. Due to the rarity of such events we believe it is still viable to use a manual process for it. Moreover the rules extracted from the C4.5 decision tree generated as part of the approach described in Chapter 6 can be used to supplement such manual rules with automatically generated rules for frequent recurrent faults.

9.3 Diagnosing the Time When the Fault Became Active

An open problem in our, and all existing approaches we are aware of, is that it is hard to diagnose the actual time that a fault occurred. As described in Section 2.1, a fault becomes active when it creates an error and this error may propagate through several internal interfaces of the system before it becomes noticeable at a service interface of the system. Usually error detection is achieved through monitoring SLOs that are defined against an external service interface. When the error is detected at the service interface, the corresponding fault may have already been active for a prolonged period of time. Examples include resource exhaustion and performance defects. For example, a memory

leak may be active for a significant time-period before the service interface experiences an error.

In order to cope with that problem one needs to have an estimate of the time-period when the fault becomes active. The fault may manifest itself in the monitoring data long before an error is detected. To cope with that problem, our diagnosis takes into account a diagnosis of a fixed-time window before and after an error is detected. However, choosing the size of that time window is still a parameter that needs to be configured by expert knowledge, as such information is hard to mine from historic data. Even if such data would be available the period from fault activation to initial error detection may hardly be consistent or match a known distribution. As part of future work, we believe our approach can be substantially improved if a mechanism for timely error reporting is found and used to determine the size of the window of interest.

Another challenge to our current approach is time skew. If the window size is chosen relatively small, such that the association of log records to the window becomes uncertain (i.e., w is within the confidence bounds of the error caused by time skew) the distribution of record types inside this window is near random and likely misses events that are associated to the actual failure.

9.4 Fully Parallel and Distributed Implementation of the Data Model

In Section 5.3 we outlined the parallel aspects of the monitoring record modelling. Our current implementation parallelises the filtering and density-clustering task as separate processes in a single machine. We believe the approach can be parallelised such that the individual tasks can execute on separate machines. Furthermore, one bottleneck is the parallel implementation of the single-linkage clustering. Olson [99] has shown preliminary approaches that parallelise hierarchical clustering for specific applications. We believe that an in-depth survey of the area of parallel clustering algorithms may reveal opportunities to parallelise this task as well.

Given a parallel implementation of our approach, it is suitable for monitoring distributed systems. The individual tasks can be spread across the distributed system to improve the scalability of the approach. For example, the filtering and density-based clustering could be performed at the monitored nodes and as such reduce the overhead of monitoring data that needs to be transmitted for further diagnosis once an error is detected. Furthermore, tasks that exhibit higher overhead can be implemented in distributed data-analysis frameworks, such as Hadoop [8] to improve the scalability of our approach.

The reader should note that we are not aware of other log-analysis approaches that exhibit the same high degree of parallelism that our approach exposes.

9.5 Optimized Parameter Tuning

The log model we introduced in Section 5.3 exhibits two free parameters. While we have a heuristic to determine the clustering threshold t, we do not have effective means to determine an upper-bound value for the parameter s. The sensitivity s controls the sensitivity towards the number of tokens and monitoring records to be considered as cluster candidate for Vaarandi's algorithm. While we have proposed a lower bound for this parameter, we do not yet have means to estimate an upper bound for modelling monitoring samples for symptom extraction. We believe by feeding back outcomes of a cross-validation of a classifier on a representative sample set, one can formulate the estimation of s as an optimization problem. We regard that problem as part of our future work.

9.6 Other Application Areas of the Data Model

One of the core contributions of this thesis is the log model proposed in Chapter 5. Although our objective is fault diagnosis, it may also serve other application areas. As part of future work, we believe it can also be adapted to intrusion detection and actionable event discovery.

Recent work by Vaarandi [141] has shown that his initial algorithm [140] can be adapted to structured monitoring data to detect intrusions. Because we extended his proposal, we also believe our approach could be adapted to this context.

Hellerstein *et al.* [45] have shown how burst-detection algorithms and frequent item-set mining can be used to discover actionable patterns from structured monitoring data. As part of future work, we believe our log model could also be adapted to that application area.

9.7 Side Effects of Recovery and Probe Actions

The proactive approach discussed in Chapter 7 assumes that side effects of probe and recovery actions are known in advance. However, it is possible that a recovery action applied to the system may cause unforeseen side-effects. To address that, we diagnose symptoms of recurrent faults again, whenever a probe or recovery action was attempted and failed. That way our proactive approach is able to cope with side-effects that match known faults.

Unfortunately, if the side effects of recovery actions do not match known faults the proactive recovery process may become unstable. Therefore, we terminate the recovery process after many failed automatic recovery attempts. Current operator practice is to manually investigate the cause of such failures. Even in that context our recovery process can be beneficial, since we can log the trail of attempted recovery and recovery actions and provide the operator with context.

9.8 Diagnosing Previously Unseen Faults

We assumed that our model can learn, and subsequently diagnose, recurrent faults from historic monitoring data. If a new fault is discovered that does not match the signature of any previously learned fault our approach will fail to diagnose it. In order to include such new faults in our model, we need to first obtain a sufficient number of samples and relearn.

Depending on the nature of the fault, this may cause significant overhead. If the fault has a manifestation that cannot be matched by existing record types, the record types themselves need to be relearned (*see* Chapter 5). If the fault does not map to an existing fault label (i.e., different component or different type of fault) then the decision-tree classifier that reflects symptoms of recurrent faults needs to be relearned (*see* Chapter 6) and the proactive model recomputed (*see* Chapter 7). If the discovery of the new fault changes the prevalence of recurrent faults significantly or requires new probe- or recovery actions, the proactive model also need to be recomputed also.

So far we have not addressed the problem of novelty detection and attribution. As part of future work, we believe the area of active learning, for example the approach by Rubens [114], may provide useful approaches to cope with that challenge of sample novelty detection of samples. In addition the approximated record types may also need to be updated with new samples. Other approaches that use grammar-inference techniques, such as Memon [86], can be integrated to detect the occurrence of new record types and trigger model relearning.

9.9 Adapting to Concept Drift and Online Learning

The model presented in Chapter 7 assumes that properties such as the types of faults, fault prevalence, effects (including side-effects) of recovery and probe actions are known. Unfortunately, data to establish these properties may be difficult to obtain or turn out to be invalid. In previous work [110] we analyzed the impact of certain types of concept drift. So far the approaches evaluated fail to account for invalid or drifting specifications about perceptually aliased faults. POMDP-learning approaches [84, 122] may be an avenue for future research to cope with learning perceptually aliased faults.

9.10 Recovery under Partial Controllability

The model we described in Chapter 7 assumes full controllability of the system to attempt recovery. Because the state coverage by most system interfaces is limited, that assumption can only be made for a small set of systems or a coarse set of recovery actions, such as full reboot. In order to expand possible application areas of the model we need to extend the model to function under partial controllability too.

As part of future work, we believe that the area of expert systems may provide some suitable approaches to expand the model. The approach we discussed in Chapter 7 may operate automatically for all system states in which full controllability is guaranteed and may provide expert advice to a manual investigator, which recovery attempts to make next, for cases in which full controllability cannot be asserted.

9.11 Incorporating Reliable Error Detection

In this thesis we assumed the presence of reliable error detection. However, in order to deploy the discussed approach in the field, reliable error detection must be developed and integrated into the suite. The fault coverage will also be determined by the quality of the error detection. We have shown [91, 108] for tracing, it is possible for diagnosis to be sensitive to a variety of subtle faults, but error detection may require further tuning. Integrating the approach with tracing the scope of limitation 9.3 will be easier to assess for specific systems.

9.12 Incorporating Dependency Models

So far our approach viewed the system as a set of independent components. However, fault propagation depends on the structure of the system. The diagnostic accuracy, especially for perceptually aliased faults, may be improved by incorporating knowledge of the system structure. In this thesis we did not make explicit assumptions about the system structure

as such data may be invalid, outdated or simply not available. In related work [62] we have shown how incorporating system structure can improve the diagnostic accuracy of metric-based fault diagnosis. We believe that such models may also improve the performance of the approach described in this thesis.

Appendix A

List of Main Publications

- Thomas Reidemeister, Mohammad A. Munawar, and Paul A.S. Ward. Model-based symptoms for problem determination in enterprise software systems; Workshop on Engineering Autonomic Software Systems, 2007
- Mohammad A. Munawar, Thomas Reidemeister, Michael Jiang, Allen George, and Paul A.S. Ward. Adaptive monitoring with dynamic tracing-based diagnosis; International Workshop on Distributed Systems: Operations and Management, 2008
- Thomas Reidemeister, Mohammad Ahmad Munawar, Miao Jiang, Paul A.S. Ward; Diagnosis of recurrent faults using log files; Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, 2009, 12–33
- Thomas Reidemeister, Miao Jiang, Paul A.S. Ward; An extensible framework for repair-driven monitoring; Proceedings of the International Conference on Network and Service Management, 2010, 142–149
- Thomas Reidemeister, Mohammad A. Munawar, Paul A.S. Ward; Identifying symptoms of recurrent faults in log files of distributed information systems; Proceedings of the IEEE/IFIP Network Operations and Management Symposium, 2010, 187–194
- Thomas Reidemeister, Miao Jiang, Paul A.S. Ward; Learning to self-recover; IFIP/IEEE International Workshop on Business-driven IT Management, 2011, 1066–1069
- Thomas Reidemeister, Miao Jiang, Paul A.S. Ward; Mining unstructured log files for recurrent fault diagnosis; Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management, 2011, 377–384

Bibliography

[1] Hervé Abdi and Lynne J. Williams. Principal component analysis. *Wiley Interdisciplinary Reviews: Computational Statistics*, 2(4):433–459, 2010.

[2] N. R. Adiga, G. Almasi, G. S. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A. A. Bright, J. Brunheroto, C. Cascaval, J. Castanos, W. Chan, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. Chiu, T. M. Cipolla, P. Crumley, K. M. Desai, A. Deutsch, T. Domany, M. B. Dombrowa, W. Donath, M. Eleftheriou, C. Erway, J. Esch, B. Fitch, J. Gagliano, A. Gara, R. Garg, R. Germain, M. E. Giampapa, B. Gopalsamy, J. Gunnels, M. Gupta, F. Gustavson, S. Hall, R. A. Haring, D. Heidel, P. Heidelberger, L. M. Herger, D. Hoenicke, R. D. Jackson, T. Jamal-Eddine, G. V. Kopcsay, E. Krevat, M. P. Kurhekar, A. P. Lanzetta, D. Lieber, L. K. Liu, M. Lu, M. Mendell, A. Misra, Y. Moatti, L. Mok, J. E. Moreira, B. J. Nathanson, M. Newton, M. Ohmacht, A. Oliner, V. Pandit, R. B. Pudota, R. Rand, R. Regan, B. Rubin, A. Ruehli, S. Rus, R. K. Sahoo, A. Sanomiya, E. Schenfeld, M. Sharma, E. Shmueli, S. Singh, P. Song, V. Srinivasan, B. D. Steinmacher-Burow, K. Strauss, C. Surovic, R. Swetz, T. Takken, R. B. Tremaine, M. Tsao, A. R. Umamaheshwaran, P. Verma, P. Vranas, T. J. C. Ward, M. Wazlowski, W. Barrett, C. Engel, B. Drehmel, B. Hilgart, D. Hill, F. Kasemkhani, D. Krolak, C. T. Li, T. Liebsch, J. Marcella, A. Muff, A. Okomo, M. Rouse, A. Schram, M. Tubbs, G. Ulsh, C. Wait, J. Wittrup, M. Bae, K. Dockser, L. Kissel, M. K. Seager, J. S. Vetter, and K. Yates. An overview of the BlueGene/L Supercomputer. *ACM/IEEE 2002 Supercomputing Conference*, page 60, 2002.

[3] Charu C. Aggarwal, Joel L. Wolf, Philip S. Yu, Cecilia Procopiuc, and Jong Soo Park. Fast algorithms for projected clustering. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 61–72, New York, NY, USA, 1999. ACM.

[4] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 94–105, New York, NY, USA, 1998. ACM.

[5] Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. NP-hardness of euclidean sum-of-squares clustering. *Machine Learning*, 75(2):245–248, May 2009.

[6] Amazon Web Services, LLC. Summary of the Amazon EC2 and Amazon RDS service disruption in the US East Region. `http://aws.amazon.com/message/65648/`. Visited 2011-06-02.

[7] Apache Software Foundation. Apache HTTP server project. `http://httpd.apache.org/`. Visited 2008-05-21.

[8] Apache Software Foundation. Hadoop. `http://hadoop.apache.org/`. Visited 2012-02-12.

[9] K.J. Astrom. Optimal control of markov processes with incomplete state information. *Mathematical Analysis and Applications*, 10:174–205, 1965.

[10] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing*, 1(1):11–33, 2004.

[11] Kirk Bauer. Logwatch. `http://www.logwatch.org/`. Visited 2008-05-21.

[12] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. *String Processing and Information Retrieval*, 2000.

[13] Michael W. Berry and Malu Castellanos. *Survey of Text Mining II: Clustering, Classification, and Retrieval*. Springer, 1st edition, 2008.

[14] Kenneth P. Birman. *Reliable Distributed Systems: Technologies, Web Services, and Applications*. Springer, 2005.

[15] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[16] Peter Bodik, Greg Friedman, Lukas Biewald, Helen Levine, George Candea, Kayur Patel, Gilman Tolle, Jon Hui, Armando Fox, Michael I. Jordan, and David Patterson. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *Proceedings of the International Conference on Autonomic Computing*, pages 89–100, 2005.

[17] Mark Brodie, Sheng Ma, Leonid Rachevsky, and Jon Champlin. Automated problem determination using call-stack matching. *Network and Systems Management*, 13(2):219–237, 2005.

[18] Aaron Brown and David Patterson. Embracing failure: A case for recovery-oriented computing (ROC). In *Proceedings of the High Performance Transaction Processing Symposium*, 2001.

[19] George Candea and Armando Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, page 125, 2001.

[20] George Candea and Armando Fox. Crash-only software. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, page 12, 2003.

[21] Anthony Cassandra, Michael L. Littman, and Nevin L. Zhang. Incremental pruning: A simple, fast, exact method for partially observable Markov decision processes. In *Conference on Uncertainty in Artificial Intelligence*, pages 54–61, 1997.

[22] Mike Chen, Alice X. Zheng, Jim Lloyd, Michael I. Jordan, and Eric Brewer. Failure diagnosis using decision trees. In *Proceedings of the International Conference on Autonomic Computing*, pages 36–43, 2004.

[23] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 595–605, 2002.

[24] D.A. Cieslak, D. Thain, and N.V. Chawla. Short paper: Troubleshooting distributed systems via data mining. In *Proceedings of the ACM International Symposium on High Performance Distributed Computing*, pages 309–312, 2006.

[25] William W. Cohen. Learning trees and rules with set-valued features. In *Proceedings of the National Conference on Artificial Intelligence*, pages 709–716, 1996.

[26] Joyce Coleman and Tony Lau. Set up and run a Trade6 benchmark with DB2 UDB. `http://www.ibm.com/developerworks/edu/dm-dw-dm-0506lau.html`. Visited 2008-06-10.

[27] Maxime Crochemore, Borivoj Melichar, and Zdenek Tronícek. Directed acyclic subsequence graph: overview. *Journal of Discrete Algorithms*, 1:255–280, 2003.

[28] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[29] Distributed Management Task Force. Common Information Model (CIM). `http://dmtf.org/standards/cim`. Visited 2011-06-02.

[30] Susan T. Dumais. Latent semantic analysis. *Annual Review of Information Science and Technology*, 38(1):188–230, 2004.

[31] Ted Dunning. Statistical Identification of Language. Technical Report MCCS-94-273, New Mexico State University, 1994.

[32] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266, 6585.

[33] Free Software Foundation. diff manpage. `http://linux.die.net/man/1/diff`. Visited 2012-02-12.

[34] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems though unstructured log analysis. In *Proceedings of Ninth IEEE International Conference on Data Mining*, pages 149–158, 2009.

[35] Venkatesh Ganti, Johannes Gehrke, and Raghu Ramakrishnan. Cactus: clustering categorical data using summaries. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '99, pages 73–83, New York, NY, USA, 1999. ACM.

[36] Jie Gao, Gautam Kar, and Parviz Kermani. Approaches to building self healing systems using dependency analysis. In *Proceedings of the Network Operations and Management Symposium*, pages 119–132, 2004.

[37] Wayt Gibbs. Software's chronic crisis. *Scientific American*, (3):72–81, 1994.

[38] Sanjay Goil, Harsha Nagesh, and Alok Choudhary. Mafia: Efficient and Scalable Subspace Clustering for Very Large Data Sets. Technical Report CPDC-TR-9906-010, North Western University, 1999.

[39] E.M. Gold. Complexity of automatic identification from given data. *Information and Control*, pages 302–320, 1978.

[40] Moises Goldszmidt, Mihai Budiu, Yue Zhang, and Michael Pechuk. Towards automatic policy refinement in repair services for large distributed systems. In *Proceedings of the ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*, pages 47–51, 2009.

[41] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley, 2000.

[42] D. Harrington, R. Presuhn, and B. Wijnen. An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks. RFC 3411 (Standard), December 2002. Updated by RFCs 5343, 5590.

[43] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer, 2003.

[44] D. M. Hawkins. The problem of overfitting. *Journal of Chemical Information and Computer Sciences*, 44(1):1–12, 2004.

[45] Joseph L. Hellerstein, Sheng Ma, and Chang-Shing Peng. Discovering actionable patterns in event data. *IBM System Journal*, 3:475–493, 2002.

[46] Brenda Hinkemeyer and Bryant A. Julstrom. A genetic algorithm for the longest common subsequence problem. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 609–610, 2006.

[47] R.A. Howard. Information value theory. In *Transactions on Systems Science and Cybernetics*, pages 22–26, 1966.

[48] Jin Huang, Jingjing Lu, and Charles X. Ling. Comparing naive bayes, decision trees, and SVM with AUC and accuracy. In *Proceedings of the 3rd IEEE International Conference on Data Mining*, pages 553–556, 2003.

[49] Jing Huang, Patrick Martin, Wendy Powley, Paul Bird, and Dmitri Abrashkevich. Lightweight problem determination in DBMSs using data stream analysis techniques. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, pages 199–211, 2010.

[50] IBM Corporation. IBM DB2 UDB. `http://www.ibm.com/software/data/db2/udb/`. Visited 2008-05-21.

[51] IBM Corporation. IBM Tivoli Monitoring. `www.ibm.com/software/tivoli/products/monitor/`. Visited 2008-05-21.

[52] IBM Corporation. IBM Websphere Application Server. `http://www.ibm.com/software/webservers/appserv/`. Visited 2008-05-21.

[53] IBM Corporation. IBM Websphere Application Server logging. `http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.base.doc/info/aes/ae/rtrb_readtrc.html`. Visited 2012-03-18.

[54] IEEE. *1003.1-2004, IEEE Standard Interpretations of IEEE Standard Portable Operating System Interface for Computer Environments*. IEEE, 2004.

[55] Michael Isard. Autopilot: Automatic data center management. *ACM SIGOPS Operating Systems Review*, 41(2):60–67, 2007.

[56] ISO/IEC 23270:2003. *C# Language Specification*. ISO, 2003.

[57] Thomas Jansen and Dennis Weyland. Analysis of evolutionary algorithms for the longest common subsequence problem. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 939–946, 2007.

[58] Miao Jiang, Mohammad Munawar, Thomas Reidemeister, and Paul Ward. Automatic fault detection and diagnosis in complex software systems by information-theoretic monitoring. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 285–294, 2009.

[59] Miao Jiang, Mohammad Munawar, Thomas Reidemeister, and Paul A.S. Ward. Heteroscedastic models to track relationships between management metrics. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management*, pages 375–381, 2009.

[60] Miao Jiang, Mohammad Ahmad Munawar, Thomas Reidemeister, and Paul A.S. Ward. Detection and diagnosis of recurrent faults in software systems by invariant

analysis. In *Proceedings of the High Assurance Systems Engineering Symposium*, pages 323–332, 2008.

[61] Miao Jiang, Mohammad Ahmad Munawar, Thomas Reidemeister, and Paul A.S. Ward. System monitoring with metric correlation models: Problems and solutions. In *Proceedings of the International Conference on Autonomic Computing*, pages 13–22, 2009.

[62] Miao Jiang, Mohammad Ahmad Munawar, Thomas Reidemeister, and Paul A.S. Ward. Dependency-aware fault diagnosis with metric-correlation models in enterprise software systems. In *Proceedings of the 6th International Conference on Network and Service Management*, pages 134–141, 2010.

[63] Tao Jiang and Ming Li. On the approximation of shortest common supersequences and longest common subsequences. *SIAM Journal on Computing*, 24(5):1122–1139, 1995.

[64] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. An automated approach for abstracting execution logs to execution events. *Journal of Software Maintenance*, 20(4):249–267, 2008.

[65] Mark W. Johnson. Monitoring and diagnosing applications with ARM 4.0. In *Proceedings of the CMG International Conference*, pages 473–484, 2004.

[66] Ian T. Joliffe. *Principal Component Analysis*. Springer, 2002.

[67] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger. Monitoring distributed systems. *Computer Systems*, 5(2):121–150, 1987.

[68] Klaus Julisch. *Using Root-Cause Analysis to Handle Intrusion Detection Alarms*. PhD thesis, Fachbereich Informatik Universität Dortmund, Germany, 2003.

[69] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134, 1998.

[70] Soila P. Kavulya, Kaustubh Joshi, Matti Hiltunen, Scott Daniels, Rajeev Gandhi, and Priya Narasimhan. Practical experiences with chronics discovery in large telecommunications systems. In *Managing Large-Scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, SLAML '11, pages 7:1–7:8, New York, NY, USA, 2011. ACM.

[71] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[72] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

[73] Tao Li, Feng Liang, Sheng Ma, and Wei Peng. An integrated framework on mining logs files for computing system management. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, pages 776–781, 2005.

[74] Tao Li, S. Masoud Sadjadi, Juan Carlos Martinez, Lokesh Sasikumar, and Manoj Pillai. Data mining for autonomic system management: A case study at FIU-SCIS. Technical report, Florida International University, 2006. FIU-SCIS-2006-03-01.

[75] Yinglung Liang, Anand Sivasubramaniam, and Jose Moreira. Filtering failure logs for a BlueGene/L prototype. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 476–485, 2005.

[76] Yinglung Liang, Yanyong Zhang, Anand Sivasubramaniam, Morris Jette, and Ramendra Sahoo. BlueGene/L failure analysis and prediction models. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 425–434, 2006.

[77] Yinglung Liang, Yanyong Zhang, Hui Xiong, and Ramendra Sahoo. Failure prediction in IBM BlueGene/L event logs. In *Proceedings of the IEEE International Conference on Data Mining*, pages 583–588, 2007.

[78] Linux Information Project. The dmesg command. `http://www.linfo.org/dmesg.html`. Visited 2008-05-21.

[79] Steve Lohr. Amazon's trouble raises Cloud Computing doubts. http://www.nytimes.com/2011/04/23/technology/23cloud.html. Visited 2011-06-02.

[80] C. Lonvick. The BSD Syslog Protocol. RFC 3164 (Informational), August 2001. Obsoleted by RFC 5424.

[81] David Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25(2):322–336, 1978.

[82] Masoud Mansouri-Samani. *Monitoring of Distributed Systems*. PhD thesis, University of London, 1995.

[83] Naoya Maruyama and Satoshi Matsuoka. Model-based fault localization in large-scale computing systems. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pages 1–12, 2008.

[84] Andrew Kachites McCallum. *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, The University of Rochester, 1996.

[85] Bořivoj Melichar and Tomás Polcar. The longest common subsequence problem: A finite automata approach. In *Implementation and Application of Automata*, pages 294–296, 2003.

[86] A. Memon. Log File Categorization and Anomaly Detection Using Grammar Inference. Master's thesis, Queen's University at Kingston, 2008.

[87] Microsoft Corporation. Microsoft .NET framework. http://www.microsoft.com/NET/. Visited 2008-05-21.

[88] Microsoft Corporation. WMI tasks: Computer software. http://msdn2.microsoft.com/en-us/library/aa394588.aspx. Visited 2009-12-02.

[89] Natwar Modani, Rajeev Gupta, Guy Lohman, Tanveer Syeda-Mahmood, and Laurent Mignet. Automatically identifying known software problems. In *Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshop*, pages 433–441, 2007.

[90] Mohammad Ahmad Munawar. *Adaptive Monitoring of Complex Software Systems using Management Metrics*. PhD thesis, University of Waterloo, 2009.

[91] Mohammad Ahmad Munawar, Thomas Reidemeister, Miao Jiang, Allen Ajit George, and Paul A.S. Ward. Adaptive monitoring with dynamic differential tracing-based diagnosis. In *Proceedings of the International Workshop on Distributed Systems: Operations and Management*, pages 162–175, 2008.

[92] National Institute of Standards and Technology. *NIST/SEMATECH e-Handbook of Statistical Methods*. 2006.

[93] OASIS. WSDM Event Format. `http://www.oasis-open.org/standards#wsdmv1.1`. Visited 2011-06-02.

[94] James Oberg. Why the Mars probe went off course. *IEEE Spectrum*, 36(12), 1999.

[95] Object Management Group, Inc. Common object request broker architecture (CORBA). `http://www.omg.org/corba/`. Visited 2008-05-21.

[96] David Ogle, Heather Kreger, Abdi Salahshour, Jason Cornpropst, Eric Labadie, Mandy Chessel, Bill Horn, John Gerken, James Schoech, and Mike Wamboldt. *Canonical Situation Data Format: The Common Base Event V1.0.1*. IBM Corporation, 2004.

[97] Adam Oliner, Alex Aiken, and Jon Stearley. Alert detection in system logs. In *Proceedings of the IEEE International Conference on Data Mining*, pages 959–964, 2008.

[98] Adam Oliner and John Stearley. What supercomputers say: A study of five system logs. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 575–584, 2007.

[99] Clark F. Olson. Parallel algorithms for hierarchical clustering. Technical Report UCB/CSD-94-786, EECS Department, University of California, Berkeley, Dec 1993.

[100] Open Group. Regular expressions. `http://www.opengroup.org/onlinepubs/007908799/xbd/re.html`. Visited 2009-04-01.

[101] Wei Peng, Tao Li, and Sheng Ma. Mining log files for data-driven system management. *SIGKDD Exploration Newsletter*, 7(1):44–51, 2005.

[102] Soila Pertet and Priya Narasimhan. Causes of failure in web applications. Technical Report CMU-PDL-05-109, Carnegie Mellon University, 2005.

[103] PICS Research Group. 2swatch. `ftp://ftp.sdsc.edu/pub/security/PICS/2swatch/README`. Visited 2008-05-21.

[104] Python Software Foundation. The Python programming language. `http://www.python.org/`. Visited 2012-03-18.

[105] Kevin Quan. Problem-resolution dissemination. Master's thesis, University of Waterloo, 2007.

[106] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann, 1993.

[107] Lawrence R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. In *Readings in Speech Recognition*, pages 267–296, 1990.

[108] Thomas Reidemeister, Miao Jiang, and Paul A.S. Ward. Differential tracing for self-monitoring in enterprise software systems. Technical Report 2008-3, University of Waterloo, 2008.

[109] Thomas Reidemeister, Miao Jiang, and Paul A.S. Ward. An extensible framework for repair-driven monitoring. In *Proceedings of the International Conference on Network and Service Management*, pages 142–149, 2010.

[110] Thomas Reidemeister, Miao Jiang, and Paul A.S. Ward. Learning to self-recover. In *Proceeedings of 6th IFIP/IEEE International Workshop on Business-Driven IT Management*, pages 1066–1069, 2011.

[111] Thomas Reidemeister, Mohammad A. Munawar, and Paul A.S. Ward. Identifying symptoms of recurrent faults in log files of distributed information systems. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium*, pages 187–194, 2010.

[112] Thomas Reidemeister, Mohammad Ahmad Munawar, Miao Jiang, and Paul A.S. Ward. Diagnosis of recurrent faults using log files. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, pages 12–23, 2009.

[113] Thomas Reidemeister, Mohammad Ahmad Munawar, and Paul A.S. Ward. Identifying symptoms of recurrent faults in log files of distributed information systems. In *Proceedings of the Network Operations and Management Symposium*, pages 187–194, 2010.

[114] Neil Rubens, Dain Kaplan, and Masashi Sugiyama. Active learning in recommender systems. In *Recommender Systems Handbook*, pages 735–767, 2011.

[115] Sivan Sabato, Elad Yom-Tov, Aviad Tsherniak, and Saharon Rosset. Analyzing system logs: A new view of what's important. In *Proceedings of the 2nd USENIX Workshop on Tackling Computer Systems Problems with Machine Learning Techniques*, pages 1–7, 2007.

[116] R. K. Sahoo, A. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, pages 426–435, 2003.

[117] Ramendra K. Sahoo, Anand Sivasubramaniam, Mark S. Squillante, and Yanyong Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *Proceedings of the International Conference on Dependable Systems and Networks*, page 772, 2004.

[118] F. Salfner. *Event-based Failure Prediction: An Extended Hidden Markov Model Approach*. PhD thesis, Fakultät für Informatik, Humboldt-Universität zu Berlin, Germany, 2008.

[119] Felix Salfner, Maren Lenk, and Miroslaw Malek. A survey of online failure prediction methods. *ACM Computing Surveys*, 42(3), 2010.

[120] Gerard Salton and Chris Buckley. Term weighting approaches in automatic text retrieval. Technical report, Cornell University, 1987.

[121] S. Ratna Sandeep, M. Swapna, Thirumale Niranjan, Sai Susarla, and Siddhartha Nandi. Cluebox: A performance log analyzer for automated troubleshooting. In *Proceedings of the USENIX Workshop on the Analysis of System Logs*, 2008.

[122] Guy Shani. *Learning and Solving Partially Observable Markov Decision Processes*. PhD thesis, Ben Gurion University, 2007.

[123] C. E. Shannon. Prediction and entropy of printed English. *Bell Systems Technical Journal*, 30:50–64, 1951.

[124] R. Sibson. SLINK: An optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, 16(1):30–34, 1973.

[125] Edward J. Sondik. *The Optimal Control of Partially Observable Markov Decision Processes*. PhD thesis, Stanford University, 1971.

[126] Edward J. Sondik. The optimal control of partially observable Markov processes over the infinite horizon: Discounted costs. *Operations Research*, 26:282–304, 1978.

[127] John Stearley. Supercomputer event logs. `http://www.cs.sandia.gov/~jrstear/logs/`. Visited 2012-03-01.

[128] John Stearley. Towards informatic analysis of syslogs. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid*, pages 309–318, 2004.

[129] Jon Stearley and Adam Oliner. Bad words: Finding faults in Spirit's syslogs. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid*, pages 765–770, 2008.

[130] Malgorzata Steinder and Adarshpal S. Sethi. A survey of fault localization techniques in computer networks. *Science of Computer Programming*, 53(2):165–194, 2004.

[131] Stanley Smith Stevens. On the theory of scales of measurement. *Science*, 103(2648):677 – 680, 1946.

[132] Rick Sturm, Wayne Morris, and Mary Jander. *Foundations of Service Level Management*. SAMS Publishing, Apr 2000.

[133] Sun Microsystems, Inc. *Java 2 Platform Enterprise Edition Specification, v1.4*. Sun Microsystems, Inc., 2003.

[134] PADS Team. Examples of the PADS project. `http://www.padsproj.org/examples.html`. Visited 2012-02-12.

[135] D. Thoenen, J. Riosa, and J.L. Hellerstein. Event relationship networks: A framework for action-oriented analysis in event management. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management*, pages 593–606, 2001.

[136] Kerry Thompson. Logsurfer. `http://www.crypt.gen.nz/logsurfer/`. Visited 2008-05-21.

[137] Zdenek Tronícek. Common subsequence automaton. In *Implementation and Application of Automata*, pages 270–275, 2002.

[138] United States Bureau of Labor Statistics. Projections data: Computer support specialists and systems administrators. `http://www.bls.gov/oco/ocos268.htm#projections_data`. Visited 2008-09-01.

[139] University of Waikato. Weka-3: Data mining software in Java. `http://www.cs.waikato.ac.nz/ml/weka/`. Visited 2009-04-01.

[140] Risto Vaarandi. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the IEEE Workshop on IP Operations and Management*, pages 119–126, 2003.

[141] Risto Vaarandi. Methods for detecting important events and knowledge from data security logs. In *Proceedings of the 2011 European Conference on Information Warfare and Security*, pages 261–267, 2012.

[142] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.

[143] Stefan Weigert, Matti Hiltunen, and Christof Fetzer. Mining large distributed log data in near real time. In *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, SLAML '11, pages 5:1–5:8, New York, NY, USA, 2011. ACM.

[144] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.

[145] Daniel Worden and Nicholas Chase. Using the Generic Log Adapter with the Log and Trace Analyzer. `https://www6.software.ibm.com/developerworks/education/ac-gla1/index.html`. Visited 2008-05-21.

[146] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Mining console logs for large-scale system problem detection. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–6, 2008.

[147] Kenji Yamanishi and Yuko Maruyama. Dynamic syslog mining for network failure monitoring. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, pages 499–508, 2005.