Optimizing System Monitoring Configurations for Non-Actionable Alerts

Liang Tang and Tao Li
School of Computer Science
Florida International University
Miami, FL, USA
Email: {ltang002, taoli}@cs.fiu.edu

Florian Pinel and Larisa Shwartz
IBM T.J. Watson Research Center
Hawthorne, NY, USA
Email: {pinel,lshwart}@us.ibm.com

Genady Grabarnik
Dept. Math & Computer Science
St. John's University
Queens, NY, USA
Email: genadyg@gmail.com

***Abstract*—**
Today's competitive business climate and the complexity of IT environments dictate efficient and cost effective service delivery and support of IT services. This is largely achieved through automating of routine maintenance procedures including problem detection, determination and resolution. System monitoring provides effective and reliable means for problem detection. Coupled with automated ticket creation, it ensures that a degradation of the vital signs, defined by acceptable thresholds or monitoring conditions, is flagged as a problem candidate and sent to supporting personnel as an incident ticket. This paper describes a novel methodology and a system for minimizing non-actionable tickets while preserving all tickets which require corrective action. Our proposed method defines monitoring conditions and the optimal corresponding delay times based on an off-line analysis of historical alerts and the matching incident tickets. Potential monitoring conditions are built on a set of predictive rules which are automatically generated by a rule-based learning algorithm with coverage, confidence and rule complexity criteria. These conditions and delay times are propagated as configurations into run-time monitoring systems.

I. Introduction

IT Service Providers are facing an increasingly intense competitive landscape and growing industry requirements. In their quest to maximize customer satisfaction, Service Providers seek to employ business intelligent solutions, which provide deep analysis, orchestration of business processes and capabilities for optimizing the level of service and cost. IT Infrastructure Library (ITIL) addresses monitoring as a continual cycle of monitoring, reporting and subsequent action that provides measurement and control of services [1].

Modern forms of distributed computing (say, cloud) provided some standardization of the initial configuration of the hardware and software. However, in order to enable most enterprise level applications, an individual infrastructure for the given application must be created and maintained on behalf of each outsourcing customer. This requirement creates great variability in the services provided by IT support teams. The aforementioned issues contribute largely to the fact that routine maintenance of the information systems remains semi-automated, and manually performed. Significant initiatives like autonomic computing led to awareness of the problem in the scientific and industrial communities and helped to introduce more sophisticated and automated procedures, which increase the productivity and guarantee the overall quality of the delivered service. System monitoring is an automated reactive system that provides effective and reliable means of ensuring that degradation of the vital signs, defined by acceptable thresholds or monitoring conditions (situations), is flagged as a problem candidate (monitoring event) and sent to the service delivery teams as an incident ticket marking harmful change. The teams identify a problem, determine its cause and find solutions.

There has been a great deal of effort spent on developing the monitoring conditions (situations) that can identify potentially unsafe functioning of the system [2] [3]. However, it is understandably difficult to recognize and quantify influential factors in malfunctioning of a complex system. Therefore classical monitoring tends to rely on periodical probing of a system for conditions which could potentially contribute to the system's misbehavior. Upon detection of the predefined conditions, the monitoring systems trigger events that automatically generate incident tickets. Defining monitoring conditions (situations) requires the knowledge of a particular system and its relationships with other hardware and software systems. It is a known practice to define conservative conditions in nature thus tending to err on the side of caution. This practice leads to a large number of tickets that require no action (non-actionable).

The contribution of this paper relates to transformation of the detected-problem candidate into an open ticket. The innovation consists in reducing the number of non-actionable tickets generated from monitoring alerts, while all actionable tickets are retained. This is achieved by deriving optimal monitoring conditions and alert delays through the analysis of historical alerts and tickets. Potential monitoring conditions and their optimal combinations are built on a set of predictive rules that are automatically generated by a rule based learning algorithm with coverage, confidence and rule complexity criteria. These conditions and alert delays are propagated as configurations into run-time monitoring systems. We also assessed the proposed approach against similar approaches and demonstrated its effectiveness in reducing the number of

978-1-4673-0269-2/12/$31.00

non-actionable tickets while retaining all real tickets with the minimal delay.

The paper is organized as follows. Section II provides a description of the problem settings and introduces the main notations used in the paper. Section III is the main technical chapter. It explains how to construct predictive rules for non-actionable alerts, how to identify timewise prolongation of the pattern, and how to choose optimal parameters according to user preferences and the Service Level Agreement (SLA). Section IV presents the experimental studies on the real alert events and tickets obtained from IBM Tivoli production servers [4]. Section V describes the related work, and finally Section VI concludes the paper.

II. Background

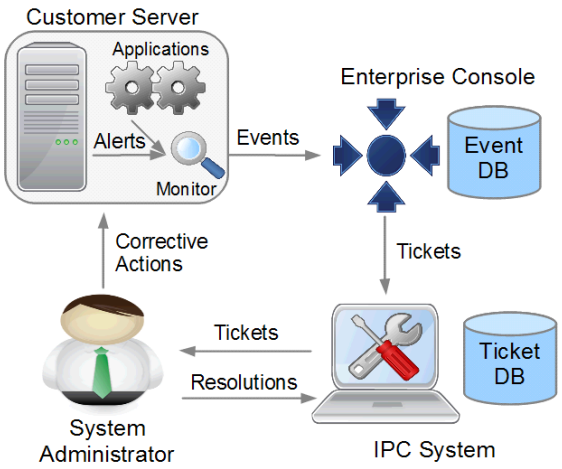


Fig. 1: Problem Detection, Determination and Resolution

The typical workflow of problem detection, determination, and resolution for the IT service provider is prescribed by the ITIL specification [1], and illustrated in Figure 1. Detection is usually provided by monitoring software running on the servers of an account, which computes metrics for the hardware and software performance at regular intervals. The metrics are then compared to acceptable thresholds, known as *monitoring situations*, and any violation results in an alert being raised. If the alert persists beyond a certain delay specified in the situation, the monitor emits an event. Events coming from an account's entire IT environment are consolidated in an enterprise console. The console uses rule-, case- or knowledge-based engines to analyze the monitoring events and decide whether to open a service ticket in the Incident, Problem, Change (IPC) system. Additional tickets are created upon customer requests. The information accumulated in the ticket is used by the System Administrators (SAs) for problem determination and resolution. As part of the service contracts between the customer and the service provider, the SLA specifies the maximum resolution times for various categories of tickets.

Performing a detailed analysis of IT system usage is time-consuming, so SAs often rely on default monitoring situations. Furthermore, IT system usage is likely to change over time. This often results in a large number of alerts and tickets, which can be categorized using the definitions provided in Table I.

TABLE I: Definitions for Alert and Ticket

Non-Actionable Alert	An alert for which the system administrator does not need to take any action.
Real Alert	An alert that requires the system administrator to take actions to fix the corresponding problem on the server.
Alert Duration	The length of time from an alert creation to its clearing.
Transient Alert	An alert that is automatically cleared before the technician opens its corresponding ticket.
Event	The notification of an alert to the Enterprise Console.
Non-Actionable Ticket	A ticket created from a non-actionable alert.
Real Ticket	A ticket created from a real alert.

Whether a ticket is actionable or not is determined by the resolution message entered in the ticket tracking database by the system administrator it was assigned to. It is not rare to observe entire categories of alerts, such as CPU or paging utilization alerts, that are almost exclusively non-actionable. Reading the resolution messages one by one, it can be simple to find an explanation: anti-virus processes cause prolonged CPU spikes at regular intervals; databases may reserve large amount of disk space in advance, making the monitors believe the system is running out of storage. With only slightly more effort, one can also fine-tune the thresholds of certain numerical monitored metrics, such as the metrics involved in paging utilization measurement. However, there are rarely enough human resources to correct the monitoring situations one system at a time, and we need an algorithm capable of discovering these usage-specific rules.

We analyzed historical tickets generated by IBM Tivoli monitoring system in recent months in 2011. The vast majority of the non-actionable alerts were transient, such as CPU and paging utilization temporary spikes, service restarts, and server reboots. These transient alerts automatically disappeared after a while, but their tickets were created into the ticketing system. When system administrators opened the tickets and logged on the server, they did not find the problem described by those tickets. Figure 2 shows the duration histogram of non-actionable alerts raised by one monitoring situation. This particular situation checks the status of a service and generates an alert without delay if the service is stopped or shutdown. These non-actionable alerts are collected from one customer's servers in 3 months. As shown by this figure, more than 75% of the alerts can be cleared automatically by waiting 20 minutes.

It is possible for a transient alert to be caused by a real system problem. However, from the perspective of the SA, if the problem cannot be found when logging on the server, there is nothing they can do with the alert, no matter what happened before. Some transient alerts may be indications of future real alerts and may be useful. But if those real alerts rise later on, the monitoring system will detect them even if the transient

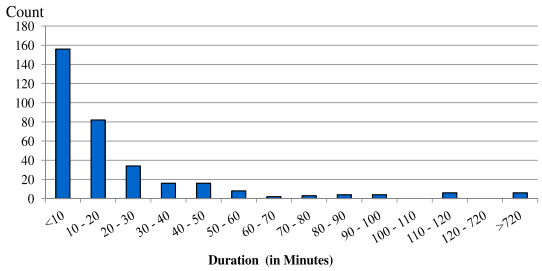


Fig. 2: Non-Actionable Alert Duration for Software Service Status Monitoring

alerts were ignored. Therefore, in our monitoring system, all transient alerts are considered non-actionable.

III. Monitoring Situation Optimization

In this section, we present our approach for optimizing monitoring situations. Section III-A introduces the goal with the challenge for our approach. Section III-B presents an overview of our approach. Sections III-C and III-D discuss two components of the approach in detail.

A. Challenge

Our goal is to refine original monitoring situations, eliminating as many as possible non-actionable alerts while retaining all real alerts. A naive solution is to build a predictive classifier into the monitoring system. Unfortunately, no prediction approach can guarantee 100% success for real alerts, and even a single missed one may cause a serious problem, such as a system crash or loss of data.

B. Solution Overview

Our solution does not predict whether an alert is real or non-actionable. Instead, we decide whether to postpone the creation of the ticket or not, and how long is it to be postponed. Even if a real alert is incorrectly classified as non-actionable, its ticket will eventually be created before violating the SLA. Figure 3 shows a flowchart of our method.

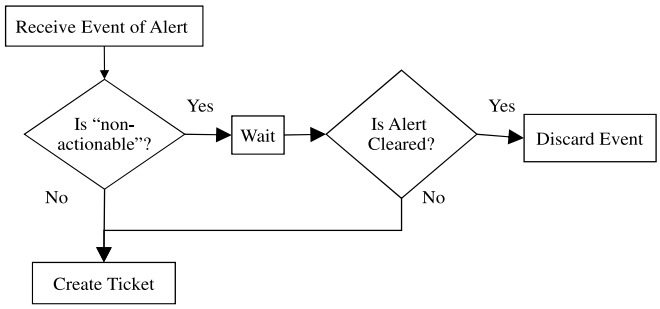


Fig. 3: Flowchart of Optimizing Monitoring Situations

There are two key problems in this approach:

- How to identify whether an alert is non-actionable or real?
- If an alert is identified as non-actionable, what waiting time should be applied to before the ticket creation?

To solve the above two problems, we propose a system which consists of five components as shown in Figure 4. Recent

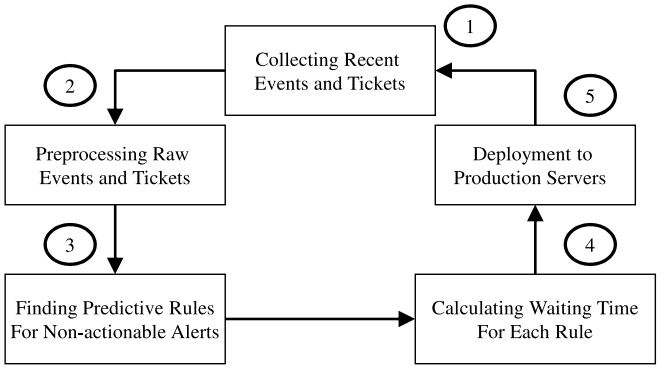


Fig. 4: Flowchart of Our System

events and alert tickets are collected in Component 1 and preprocessed in Component 2. In Component 3, we search for predictive rules to build the non-actionable alert predictor, then calculate the waiting time in Component 4. After those results have been checked by the system administrators, the predictive rules and the waiting time are deployed to production servers in Components 5, and Component 1 is revisited to collect new events and tickets. As the processing loop is designed for an once-a-month approach, all processes in the system are off-line. Components 1, 2 and 5 are straightforward, we only present the details of Component 3 and 4.

Our target system is static networking and system. Our work is based on the off-line learning. The configurations of production systems are known to change as often as every 1 or 3 months. This is the reason for monitoring life cycle process where the effectiveness of monitoring configuration re-assessed quarterly. Day-to-day operation is not suitable for our static network and system, because there are two practical issues: 1) One day's alerts and tickets are very few, so system administrators would not have enough confidence to change the monitoring configuration. 2) Day-to-day operation would increase the amount of unnecessary work for system administrators, because every learned rule would trigger a change request for applying it to the production server. In case of an unexpected event burst it is possible to run our solution again when the system administrators detect it.

C. Finding Predictive Rules For Non-actionable Alerts

Predictive Rule

The alert predictor roughly assigns a label to each alert, "non-actionable" or "real". It is built on a set of predictive rules that are automatically generated by a rule-based learning algorithm [5] based on historical events and alert tickets . Example 1 is an example of the predictive rule, where "*PROC_CPU_TIME*" is the CPU usage of a process. "*PROC_NAME*" is the name of the process.

Example 1: ***if*** *PROC_CPU_TIME > 50%* ***and*** *PROC_NAME = 'Rtvscan',* ***then*** *this alert is non-actionable.*

A predictive rule consists of a rule condition and an alert label. A rule condition is a conjunction of *literals*, where each *literal* is composed of an event attribute, a relational operator and a constant value. In Example 1, "*PROC_CPU_TIME* $>$ *50%*" and "*PROC_NAME = 'Rtvscan'*" are two *literals*, where "*PROC_CPU_TIME*" and "*PROC_NAME*" are event attributes, "$>$" and "$=$" are relational operators, and "50%" and "*Rtvscan*" are constant values. If an alert event satisfies a rule condition, we call this alert covered by this rule. Since we only need predictive rules for non-actionable alerts, the alert label in our case is always "non-actionable".

Predictive Rule Generation

The rule-based learning algorithm [5] first creates all *literals* by scanning historical events. Then, it applies a breadth-first search for enumerating all *literals* in finding predictive rules, i.e., those rules having predictive power. This algorithm has two criteria to quantify the minimum predictive power: the minimum confidence $minconf$ and the minimum support $minsup$ [5]. In our case, $minconf$ is the minimum ratio of the numbers of non-actionable alerts and of all alerts covered by the rule, and $minsup$ is the minimum ratio of the number of alerts covered by the rule and the total number of alerts. For example, $minconf = 0.9$ and $minsup = 0.1$, then for each predictive rule found by the algorithm, at least 90% covered historical alerts are non-actionable, and at least 10% historical alerts are covered by this rule. Therefore, this rule has a certain predictive power for non-actionable alerts. The two criteria govern the performance of our method, defined as the total number of removed non-actionable alerts. To achieve the best performance, we loop through the values of $minconf$ and $minsup$ and compute the performance for each pair.

Predictive Rule Selection

According to the SLA, real tickets must be acknowledged and resolved within a certain time. Our method has to know the maximum allowed time for postponing a ticket. In addition, for each monitoring situation, our method also needs to know the maximum ratio of real tickets that can be postponed, which is mainly determined by the severity of the situation. Therefore, there are two user-oriented parameters:

- The maximum ratio of real alerts that can be delayed, $ratio_{delay}$, $0 \leq ratio_{delay} \leq 1$.
- The maximum allowed delay time for any real alert, $delay_{max}$, $delay_{max} \geq 0$.

$ratio_{delay}$ and $delay_{max}$ are specified by the system administrators according to the severity of the monitoring situation and the SLA.

Although the predictive rule learning algorithm can learn many rules from the training data, we only select those with strong predictive power. Laplace accuracy is a widely used measure for estimating the predictive power of a rule [6] [7] [8], which is defined as follows:

$$LaplaceAccuracy(c_i, \mathcal{D}) = \frac{N(c_i) + 1}{N_{non} + 2},$$

where $\mathcal{D}$ is the set of alert events, c_i is a predictive rule, $N(c_i)$ is the number of events in $\mathcal{D}$ satisfying rule c_i, and N_{non} is the total number of non-actionable events in $\mathcal{D}$. Laplace accuracy is seen as an estimate of a conditional probability [7] [8]. For example, if a rule c_1 in $\mathcal{D}$ has $LaplaceAccuracy(c_1, \mathcal{D}) = 0.9$, it implies that given an alert e which is covered by c_1, the probability that e is non-actionable is 0.9.

Algorithm 1 FindPredictiveRules($\mathcal{D}$, $delay_{max}$, $ratio_{delay}$)

Parameter: $\mathcal{D}$: set of labeled historical alert events; $delay_{max}$: maximum allowed delay time for any real alert; $ratio_{delay}$: maximum ratio of real alerts that can be delayed;
Result: $\mathcal{P}$: set of predictive rules

1: // Step 1: Remove non-actionable and long-duration alerts from $\mathcal{D}$
2: $\mathcal{F} \leftarrow \{e | e \in \mathcal{D}, e.label = `non-actionable', e.duration \leq delay_{max}\}$
3: $\mathcal{R} \leftarrow \{e | e \in \mathcal{D}, e.label = `real'\}$
4: $\mathcal{D}' \leftarrow \mathcal{F} \cup \mathcal{R}$
5: // Step 2: Invoke learning algorithm to learn predictive rules
6: $\mathcal{C} \leftarrow$ *MineQuantativeRules*($\mathcal{D}'$)
7: $\mathcal{P} \leftarrow \emptyset$
8: // Step 3: Compute Laplace accuracy of each rule
9: **for** $c_i \in \mathcal{C}$ **do**
10: $Acc_i \leftarrow$ *LaplaceAccuracy*(c_i,$\mathcal{D}'$)
11: **end for**
12: Sort $\mathcal{C}$ by Laplace expected accuracy to list L in descending order
13: // Step 4: Select top and non-redundant predictive rules
14: $\mathcal{R}_{delay} \leftarrow \emptyset$
15: $i \leftarrow 0$
16: **while** $|\mathcal{P}| < k$ **and** $i < |\mathcal{C}|$ **do**
17: $c_i \leftarrow L[i]$
18: $i \leftarrow i + 1$
19: **for** $p_j \in \mathcal{P}$ **do**
20: **if** *isMoreSpecific*(c_i, p_j) **then**
21: **continue**
22: **end if**
23: **end for**
24: **for** $e \in \mathcal{D}'$ **do**
25: **if** $isCovered(e, c_i)$ **and** $e.label = `real'$ **then**
26: $\mathcal{R}_{delay} \leftarrow \mathcal{R}_{delay} \cup \{e\}$
27: **end if**
28: **end for**
29: **if** $|\mathcal{R}_{delay}| > |\mathcal{R}| \cdot ratio_{delay}$ **then**
30: **break**
31: **end if**
32: $\mathcal{P} \leftarrow \mathcal{P} \cup \{c_i\}$
33: **end while**
34: **return** $\mathcal{P}$

Algorithm 1 lists the pseudocode for finding non-actionable alert predictive rules. For an alert event e, $e.label$ denotes its the label and $e.duration$ denotes its duration. The algorithm consists of four steps. Step 1 removes the non-actionable alerts whose duration is longer than $delay_{max}$, because they may not be transient. Step 2 invokes the learning algorithm *MineQuantativeRules* to obtain predictive rules [5] from historical data. Then in step 3, the Laplace accuracy is first computed for all predictive rules. Then the rules are sorted in decreasing order of the Laplace accuracy. The last step is selecting rules in the sorted order until the number of covered real alerts $|\mathcal{R}_{delay}|$ satisfies the criterion $|\mathcal{R}| \cdot ratio_{delay}$.

Another issue in selecting predictive rules is rule redundancy. For example, let us consider the two predictive rules:

X. *PROC_CPU_TIME* $>$ 50% ***and*** *PROC_NAME* = 'Rtvscan'
Y. *PROC_CPU_TIME* $>$ 60% ***and*** *PROC_NAME* = 'Rtvscan'

Clearly, if an alert satisfies Rule Y, then it must satisfy Rule X as well. In other words, Rule Y is more specific than Rule X. If Rule Y has a lower accuracy than Rule X, then Rule Y is redundant given Rule X (but Rule X is not redundant given Rule Y). In our work, we perform redundant rule pruning to discard the more specific rules with lower accuracies. The pseudocode from line 9 to line 18 of Algorithm 1 describes the rule selection, in which *isMoreSpecific*(c_i, p_j) is a subroutine to check whether c_i is more specific than p_j. If rule c_i is more specific to any previous rule in $\mathcal{P}$, the algorithm ignores c_i since every previous rule has a higher estimated accuracy.

Why choose a rule-based predictor?

There are two reasons. First, each monitoring situation is equivalent to a quantitative association rule, so the predictor can be directly implemented in the existing system. Other sophisticated classification algorithms, such as *support vector machine* and *neural network*, may have a higher precision in predicting non-actionable alerts. However, their classifiers are very difficult to implement as monitoring situations in real systems. The second reason is that a rule-based predictor can be easily verifiable by customers. For instance, Example 1 implies that high CPU utilization alerts from 'Rtvscan' are non-actionable. This does not create a problem for the customers because they can check with the server and verify that 'Rtvscan' is from the Norton Anti-Virus software. In contrast, a linear/non-linear equation or a neural network formed by several system attributes is very hard for a customer to verify.

D. Calculating Waiting Time for Each Rule

Waiting time is the duration by which tickets should be postponed if their corresponding alerts are classified as non-actionable. It is not unique for one monitoring situation. Since an alert can be covered by different predictive rules, we set up different waiting times for each of them. For example, the situation described in Example 1 predicts non-actionable alerts about CPU utilization of 'Rtvscan'. We can also find another predictive rule as follows:

if *PROC_CPU_TIME* $>$ *50%* ***and*** *PROC_NAME = 'perl logqueue.pl',* ***then*** *this alert is non-actionable.*

However, the job of 'perl' is different from that of 'Rtvscan', and their durations are not the same, and the waiting time will differ accordingly.

In order to remove as many as possible non-actionable alerts, we set the waiting time of a selected rule as the longest duration of the transient alerts covered by it. For a selected predictive rule p, its waiting time is

$$wait_p = \max_{e \in \mathcal{F}_p} e.duration,$$

where

$$\mathcal{F}_p = \{e | e \in \mathcal{F}, isCovered(p, e) =' true'\},$$

and $\mathcal{F}$ is the set of transient events. Clearly, for any rule $p \in \mathcal{P}$, $wait_p \leq delay_{max}$. Therefore, no ticket can be postponed for more than $delay_{max}$.

IV. Evaluation

This section presents empirical studies for our proposed method.

A. Implementation and Testing Environment

Our proposed system is designed as a component of the IBM Tivoli Monitoring system. We implemented it in Java 1.6. This testing machine is Windows XP with Intel Core 2 Duo CPU 2.4GHz and 3GB of RAM.

B. Data Collection

Experimental alert events and tickets are collected from production servers of the IBM Tivoli Monitoring system [4], summarized in Table II. The data set of each account covers a period of 3 months. $|\mathcal{D}|$ is the number of events that generated tickets in the ticketing systems. N_{non} is the number of non-actionable events in all ticketed events. # Situations is the number of monitoring situations. # Nodes is the number of monitored servers.

TABLE II: Data Summary

Data Set	$\|\mathcal{D}\|$	N_{non}	# Attributes	# Situations	# Nodes
Account1	18,974	9,281	33	73	989
Account2	50,377	39,971	1082	320	1212

Figures 5a and 6a show the durations of all non-actionable alerts for Account1 and Account2 respectively. As we claimed in Section II, most non-actionable alerts are transient alerts whose durations are less than 360 minutes. Note that 360 minutes usually cover the entire set of transient alerts generated by all monitoring situations, while in Section II that 75% non-actionable alerts durations are less than 20 minutes is only for one monitoring situation as an example shown by Figure 2.

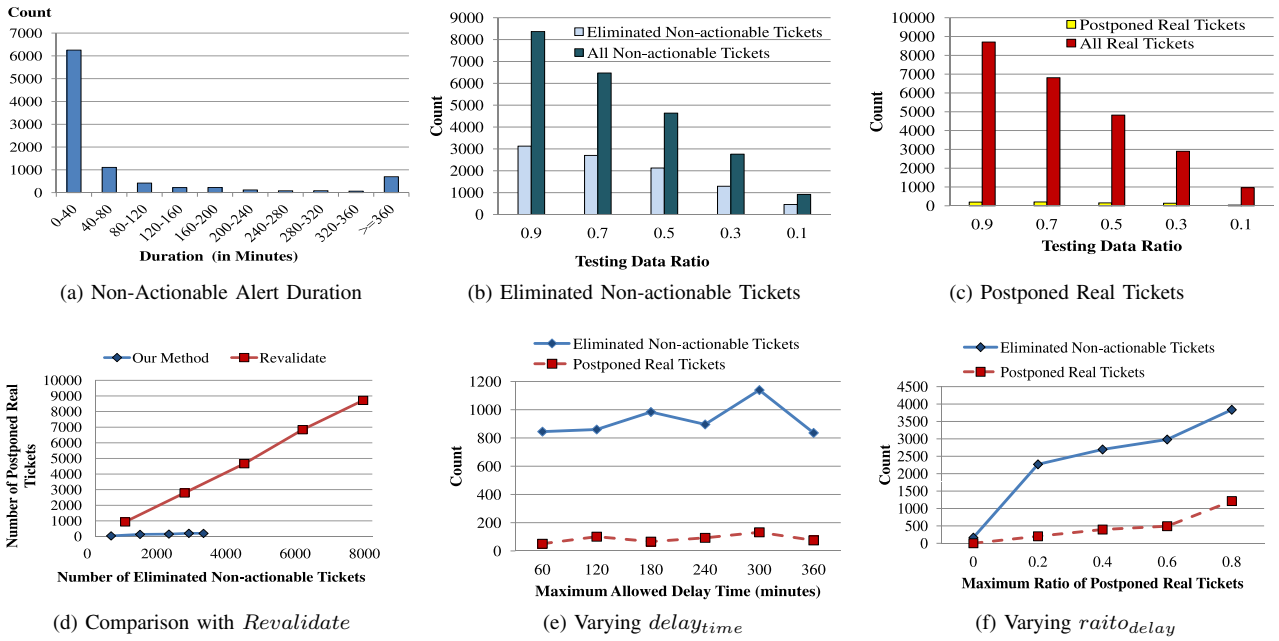
C. Performance Measure

There are two performance measures:

- FP: The number of non-actionable tickets eliminated.
- FD: The number of real tickets postponed.

To achieve a better performance, a system should have a larger FP with a smaller FD.

We split each data set into the training part and the testing part. "Testing Data Ratio" is the fraction of the testing part in the data set, and the rest is the training part. For example, "Testing Data Raito=0.9" means that 90% of the data is used for testing and 10% is used for training. All FP and FD are only evaluated for the testing part.



(a) Non-Actionable Alert Duration

(b) Eliminated Non-actionable Tickets

(c) Postponed Real Tickets

(d) Comparison with $Revalidate$

(e) Varying $delay_{time}$

(f) Varying $raito_{delay}$

Fig. 5: Results for Account1

D. Overall Performance

Based on the experience of the system administrators regarding these two data sets, we set $delay_{max} = 360$ minutes and $ratio_{delay} = 0.25$ for all monitoring situations. Figures 5 and 6 present the experimental results for Account1 and Account2 respectively. Figures 5b and 5c show the number of eliminated non-actionable tickets and the number of postponed real tickets for Account1. Our method eliminates more than 25% of the non-actionable alerts and only postpones less than 3% of the real tickets. Figures 6b and 6c show the two performance measures for Account2. Our method eliminates more than 75% of the non-actionable alerts and only postpones less than 3% of the real tickets. Clearly, our method has a better performance on Account2 than on Account1 since Account2 has more non-actionable alerts and more event attributes.

Comparing with $Revalidate$

Since most alert detection methods cannot guarantee no false negatives, we only compare our method with the idea mentioned in [9], $Revalidate$, which revalidates the status of events and postpones all tickets. $Revalidate$ has only one parameter, the postponement time, which is the maximum allowed delay time $delay_{max}$. Figuress 5d and 6d compare the respective performance of our method and $Revalidate$. While $Revalidate$ is clearly better in terms of elimination of non-actionable alerts, it postpones all real tickets, the postponement volume being 1000 to 10000 times larger than our method.

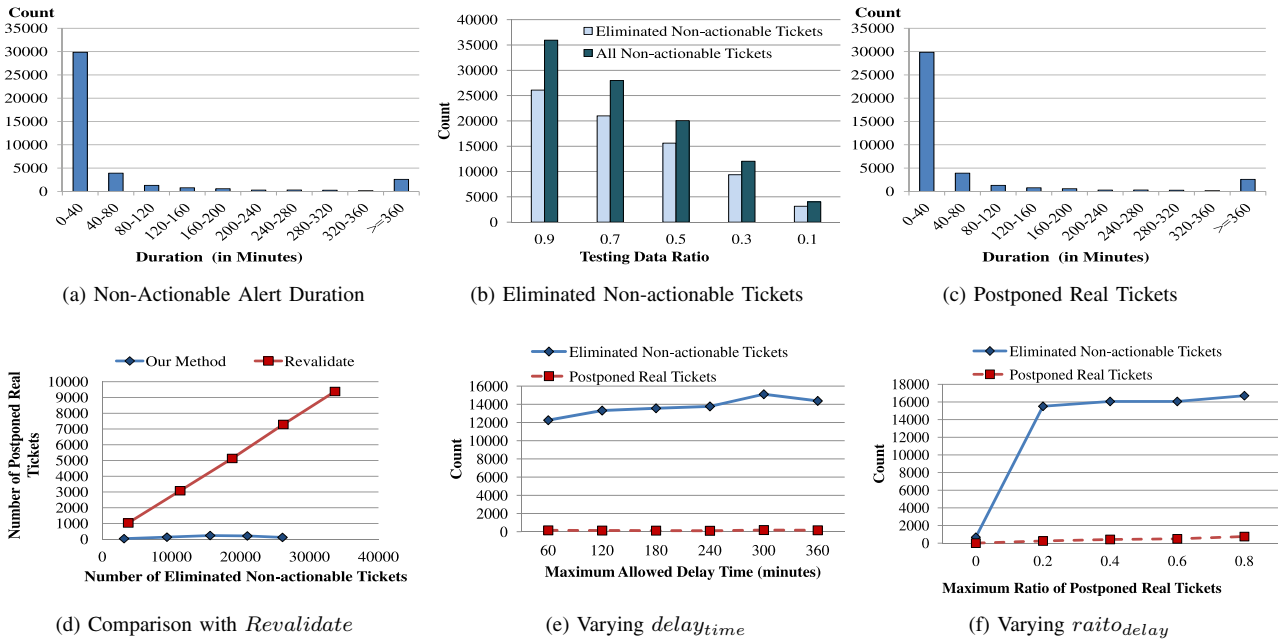
E. Predictive Rules

Tables III and IV list several discovered predictive rules for non-actionable alerts in Account1 and Account2 respectively, where $wait_p$ is the delay time for a rule, FP_p is the number of non-actionable alerts eliminated by a rule in the testing data, and FD_p is the number of real tickets postponed by a rule in the testing data.

In Table III, the first rule is the high CPU utilization of "conhost", a normal process in Microsoft Windows. Windows server administrators usually execute batch files and scripts in console windows leading to frequent CPU alerts (most of which are non-actionable). Once the executions are over, these alerts disappear automatically.

In Table IV, the first rule is the overall CPU utilization. Surprisingly, almost all of those alerts are non-actionable. Therefore, the rule for this situation is "N/A", meaning that there is no condition branch for this situation. By waiting 355 minutes, most alerts will disappear by themselves. The last two rules in this table are for the unix file system space. The folder "/logs" is the place to generate temporary logs by some application, which will be deleted automatically by the same application. Therefore, even if the used space of this folder is high, it is unlikely to lead to an overflow for this folder if the number of used inodes is less than or equal to 1616.

From the experiment results, we find that some discovered predictive rules are duplicated, because some event attributes have the same content but different names. For example, in Table IV for situation "fss_xuxw_std", "mount_point_u" is the same as "sub_origin". Therefore, the 5th rule is a duplicate of the 4th rule. Currently, our method does not handle this kind of duplicated rules. This should not affect our overall performance since those duplications are readily identified by the system administrators, who are required to check every discovered predictive rule before it is deployed into the production servers. And in any case, duplicated attributes are uncommon in the real data.



(a) Non-Actionable Alert Duration (b) Eliminated Non-actionable Tickets (c) Postponed Real Tickets

(d) Comparison with *Revalidate* (e) Varying $delay_{time}$ (f) Varying $raito_{delay}$

Fig. 6: Results for Account2

TABLE III: Sampled Rules for Account1 with Testing Data Ratio = 0.7

Situation	Rule Condition	$wait_p$	FP_p	FD_p
prccpu_3ntw_std5	CPU_Utilization_Proc = conhost	120 min	96	6
dcss_33zc_stdv3	NODE = nimwpdc01	120 min	172	0
dcss_33zc_stdv3	NODE = ridc0001	15 min	115	3
svc_3ntc_INTEL01	Service_Name = Symantec AntiVirus	305 min	428	2
dsp_3ntc_std3v2	Disk_Name = H:	213 min	17	6

TABLE IV: Sampled Rules for Account2 with Testing Data Ratio = 0.3

Situation	Rule Condition	$wait_p$	FP_p	FD_p
cpu_xuxw_std	N/A	355 min	7093	5
monlog_3ntw_std	current_size_64 >= 0 **and** record_count >= 737161	80 min	23	0
svc_3ntw_vsa_std	binary_path = R:\IBMTEMP\VSA\VSASvc_Cli.exs	30 min	27	0
fss_xuxw_std	inodes_used <= 1616 **and** mount_point_u = /logs	285 min	12	2
fss_xuxw_std	inodes_used <= 1616 **and** sub_origin = /logs	285 min	12	2

F. Varying Parameters

Figures 5e and 6e show the performance by varying $delay_{time}$ (with $raito_{delay} = 0.25$). The two performance measures do not have a significant change by increasing $delay_{time}$. The reason is that, very few non-actionable alerts' durations are between 60 and 360 minutes (shown by Figures 5a and 6a). But when we vary $ratio_{delay}$ (with $delay_{time} = 360$), the number of eliminated non-actionable tickets and the number of postponed real tickets both increase along with $ratio_{delay}$ (shown by Figures 5f and 6f). Based on these results, the choices of $raito_{delay}$ and $delay_{time}$ have different influences on different monitoring situations.

V. Related Work

This section reviews prior research studies related to system monitoring as part of IT Service Management. System monitoring has become a significant research area of IT industry in the past few years. There are many commercial products such as IBM Tivoli [4], HP OpenView [10] and Splunk [11] focusing on the system monitoring. Numerous studies focused on network monitoring [12] [13] [14] [15] [16] [17] which is critical for distributed systems. A number of studies focused on analysis of historical events with the goal of improving the understanding of system behaviors. A significant amount of work was done on analysis of system log files and their monitoring events. Yet another area of related interest is identification of actionable patterns of events. Finally we discuss prior research efforts that deals with algorithmic parameters similar to those in our method such as coverage, confidence, and alert duration.

A. Network Monitoring

Network monitoring is used to check the "health" of communication by inspecting data transmission flow, sniffing data packets, analyzing bandwidth and so on [12] [13] [14] [15] [16] [17]. It is able to detect node failures, network intrusions,

or other abnormal situations in the distributed system. The main difference between network monitoring and our proposed method is the monitored target, which can be any component or subsystem of the system, the hardware of the system (such as CPU, hard disk) or the software (such as a database engine, or a web server).

B. Actionable Event Patterns

A significant amount of work in data mining has been done to identify actionable patterns of events, see for example [18], [19], [20]. Different types of patterns like (partially) periodic patterns, event bursts, mutually dependant patterns were introduced to describe system management events. Lots of efficient algorithms were developed to find and interpret such patterns. Our work is based on the part of event processing workflow that takes into account human processing of the tickets. This allowed us to identify non-actionable patterns with significant precision. In the event processing workflow, non-actionable events are transformed into non-actionable tickets thus creating a number of false positives. Identification of non-actionable events made it possible to significantly reduce the number of false positives.

Translation of actionable patterns into enterprise software rules is considered in [21] and [22]. We implemented our findings as a component prototype for Enterprise Console. The prototype gets information about user preferences and SLA, mines events and suggests monitoring conditions as well as their duration parameters.

C. Parsing structured and unstructured data

Specialized log parsers were created to parse and transform applications and information system operation logs. Usually, logs are semi-structured, containing both structured (e.g., log entry prefixes and timestamp) and unstructured text (e.g., exception, error or warning descriptions, and display of application state). Logs are highly heterogeneous, since they contain outputs of many different applications or components. The parsers transform them into relational or extensible (XML like semi-structured) formats and can operate in offline (like [23], [24], [25], [26]) or in online, streaming regime (like [27]). In our work, parsers are used to translate monitoring events into attribute-value pairs for further analysis. However, unlike existing works, we also include the analysis of ticket resolution descriptions for identifying real tickets where a non-trivial amount of work has been done. Such information was used to tag monitoring events as actionable or non-actionable.

D. Parameter Tuning

Parameter tuning in log patterns mining is studied in [12], [13]. Usually mining parameters describe how strongly elements of the pattern are interconnected or correlated (e.g., confidence), and what percentage of the data stream should be covered (e.g., support). Tuning up mining parameters is a delicate process. The parameters considered in our work include the percentage of non-actionable events covered and the number of events covered.

Discovering time related patterns from system logs is considered in [28], [29]. In our study, the duration time of a pattern depends on a couple of factors such as actual delay time and acceptable SLA thresholds. While the distribution of recognized non-actionable patterns depends only on historical data, we take the delay tolerance of a customer as additional input.

VI. Conclusion

This paper provides an automated refinement for monitoring conditions (situations) which facilitates a closed loop approach to system management. This solution can reduce the number of non-actionable (false positive) tickets generated from monitoring alerts while retaining all actionable (or real) tickets. It minimizes the cost of providing effective and reliable means for problem detection. A rule based learning algorithm with coverage, confidence and rule complexity criteria is involved in this solution. Furthermore, it can be used periodically to adjust monitoring situations after a system has gone through a change, thus helping to enhance the overall reliability in IT Service management.

In our future work, we will investigate and develop more advanced and efficient rule learning algorithms to improve the accuracy and efficiency.

Acknowledgment

The work is supported in part by NSF grants IIS-0546280 and HRD-0833093. The authors are indebted to Mr. E. Goldberg for editorial assistance.

References

[1] "ITIL," http://www.itil-officialsite.com/home/home.aspx.
[2] E. Hoke, J. Sun, and C. Faloutsos, "InteMon: Intelligent system monitoring on large clusters," in *Proceedings of VLDB*, 2006, pp. 1239–1242.
[3] R. V. Renesse, K. P. Birman, and W. Vogels, "Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining," *ACM Transactions on Computer Systems*, vol. 21, pp. 164–206, 2003.
[4] "IBM Tivoli : Integrated Service Management software," http://www-01.ibm.com/software/tivoli/.
[5] R. Srikant and R. Agrawal, "Mining quantitative association rules in large relational tables," in *Proceedings of ACM SIGMOD*, 1996, pp. 1–12.
[6] X. Yin and J. Han, "CPAR: Classification based on predictive association rules," in *Proceedings of SDM*, 2003.
[7] M. J. Pazzani, C. J. Merz, P. M. Murphy, K. Ali, T. Hume, and C. Brunk, "Reducing misclassification costs," in *Proceedings of ICML*, New Brunswick, NJ, USA, July 1994, pp. 217–225.
[8] J. Li, "Robust rule-based prediction," *IEEE Trans. Knowl. Data Eng. (TKDE)*, vol. 18, no. 8, pp. 1043–1054, August 2006.
[9] L. A. Castillo, P. D. Mahaffey, and J. P. Bascle, "Apparatus and method for monitoring objects in a network and automatically validating events relating to the objects," U.S. Patent, Dec. 2008, US 7,469,287 B1.
[10] "HP OpenView : Network and Systems Management Products," http://www8.hp.com/us/en/software/enterprise-software.html.
[11] "Splunk: A commerical machine data managment engine," http://www.splunk.com/.
[12] S. R. Kashyap, J. Ramamirtham, R. Rastogi, and P. Shukla, "Efficient constraint monitoring using adaptive thresholds," in *Proceedings of ICDE*, Cancun, Mexico, 2008, pp. 526–535.
[13] S. Agrawal, S. Deb, K. V. M. Naidu, and R. Rastogi, "Efficient detection of distributed constraint violations," in *Proceedings of ICDE*, Istanbul, Turkey, 2007, pp. 1320–1324.

[14] S. Mccanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture," in *USENIX Technical Conference*, 1993, pp. 259–270.

[15] K. Xu, Z.-L. Zhang, and S. Bhattacharyya, "Profiling internet backbone traffic: behavior models and applications," in *ACM SIGCOMM Conference*, 2005, pp. 169–180.

[16] C. Estan, S. Savage, and G. Varghese, "Automatically inferring patterns of resource consumption in network traffic," in *ACM SIGCOMM Conference*, 2003, pp. 137–148.

[17] M. J. Ranum, K. Landfield, M. T. Stolarchuk, M. Sienkiewicz, A. Lambeth, and E. Wall, "Implementing a generalized tool for network monitoring," in *USENIX Systems Administration Conference*, 1997, pp. 1–8.

[18] J. L. Hellerstein, S. Ma, and C.-S. Perng, "Discovering actionable patterns in event data," *IBM Systems Journal*, vol. 43, no. 3, pp. 475–493, 2002.

[19] W. Peng, C. Perng, T. Li, and H. Wang, "Event summarization for system management," in *Proceedings of ACM KDD*, San Jose, California, USA, August 2007, pp. 1028–1032.

[20] J. Kiernan and E. Terzi, "Constructing comprehensive summaries of large event sequences," in *Proceedings of ACM KDD*, Las Vegas, Nevada, USA, August 2008, pp. 417–425.

[21] J. Gao, G. Jiang, H. Chen, and J. Han, "Modeling probabilistic measurement correlations for problem determination in large-scale distributed systems," in *Proceedings of the 29th International Conference on Distributed Computing Systems (ICDCS'09)*, 2009, pp. 623–630.

[22] C. Perng, D. Thoenen, G. Grabarnik, S. Ma, and J. Hellerstein, "Data-driven validation, completion and construction of event relationship networks," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2003, pp. 729–734.

[23] L. Tang and T. Li, "LogTree: A framework for generating system events from raw textual logs," in *Proceedings of ICDM*, Sydney, Australia, December 2010, pp. 491–500.

[24] L. Tang, T. Li, and C. Perng, "LogSig: Generating system events from raw textual logs," in *Proceedings of ACM CIKM*, 2011.

[25] M. Aharon, G. Barash, I. Cohen, and E. Mordechai, "One graph is worth a thousand logs: Uncovering hidden structures in massive system event logs," *Machine Learning and Knowledge Discovery in Databases*, pp. 227–243, 2009.

[26] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Experience mining google's production console logs," *Proceedings of SLAML*, 2010.

[27] G. Grabarnik, A. Salahshour, B. Subramanian, and S. Ma, "Generic adapter logging toolkit," in *Autonomic Computing, 2004. Proceedings. International Conference on*. IEEE, 2004, pp. 308–309.

[28] T. Li, F. Liang, S. Ma, and W. Peng, "An integrated framework on mining logs files for computing system management," in *Proceedings of ACM KDD*, Chicago, Illinois, USA, August 2005, pp. 776–781.

[29] P. Wang, H. Wang, M. Liu, and W. Wang, "An algorithmic approach to event summarization," in *Proceedings of ACM SIGMOD*, Indianapolis, Indiana, USA, June 2010, pp. 183–194.