VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

Automated Performance Management in a Distributed System

Francis Alexander Gardner

Technical Report CS-TR-94-23
February 1994

Abstract

This thesis addresses automated performance management in a distributed system. The specific area of performance management being explored is the detection of resource contention.

This thesis describes methods for detecting symptoms of performance problems. The methods include ideas from statistical process control and time series modelling. This thesis also presents a framework which is used to detect performance problems in a distributed system. The framework allows the exploration of different methods for problem detection, and builds on another project, called Control Room, which displays information about resource use in a distributed system. The framework consists of a management language which is used to describe the system under management and symptoms of problems associated with the system, and an alarm manager which detects and reports these problems. An implementation of the framework is also described.

Publishing Information

A thesis submitted to the Victoria University of Wellington in fulfillment of the requirements for the degree of Master of Science in Computer Science in 1994.

Author Information

The author is now with the Network Engineering Group of Industrial Research Ltd. He may be reached at f.gardner@irl.cri.nz.

Contents

List of Figures

Acknowledgements

First and foremost I would like to thank my supervisor John Hine for his commitment, ideas, and guidance that was gentle when it needed to be and not so gentle when it needed to be, and always kept me on the straight and narrow.

I would also like to thank Alistair Gray for providing expert consultation on statistical process control and time series modelling.

Thanks to Zhiqing Zhang and Julian Anderson for explaining some of the finer points of Control Room; to Joe Hellerstein at the IBM Yorktown Heights Research Center, New York and Tihamer T. Toth-Fejel at Industrial Technology Institute, Michigan for sending me and discussing their work; and to Vivian McAlistair and Aino Krumins at IBM, Wellington for their help with obtaining IBM documents.

I would also like to thank the staff and students of the computer science department, past and present, for comments on, and contributions to my work.

This thesis was typeset with the LaTeX document preparation system. Figures were created with `xfig`, and from the `Splus` statistical package. Screen dumps were captured with `xgrab`.

Chapter 1

Introduction

The number and size of distributed systems are growing as they become more popular. A distributed system is comprised of a set of shared resources connected by a relatively slow network. The task of managing such a system is different from managing a centralised system because of differences between the systems [WM88]. The management task is complicated because a picture of system use must be pieced together from information gathered from the components of the system. This information may be collected at different times and may be contradictory.

The International Standards Organisation has identified five areas that should be addressed when managing a system [Kle88].

1. Accounting management, for determining resource usage and if necessary assigning costs to this usage,

2. Configuration management, used to control the configuration of the system,

3. Fault management, provides for the detection, isolation and correction of system faults,

4. Performance management, used to monitor and evaluate the performance of the system, and

5. Security management, provides services for authentication, access control and data encryption.

This thesis addresses the area of performance management in a distributed system, in particular detecting and reporting problems due to resource congestion. A resource can become congested for a number of reasons. The resource might have taken the work load from another resource that provided the same service, but is now unavailable due to a fault. A bad scheduling algorithm might be used to allocate the resource. For example, a job scheduler that doesn't take the amount of free memory into account may allocate so many processes that the system spends more time processing page faults than running user programs. A resource may be congested simply because there is a high demand for it.

Part of my work was to explore and evaluate different methods for detecting performance problems. The other part of my work was to design and implement a framework in which to use these methods and others for detecting performance problems in a distributed system. An important part of this development was the design and implementation of a language for describing resources and problems with them. This framework is part of Control Room, an ongoing project within the department, which is a system for displaying information about resource use in a distributed system.

Tool	Description
The Active Monitor and the TAX system	Two of the tools used to manage the Kansas University Packet Switch Network. The TAX (Traffic Analysis of X.25) system is a set of hardware monitors that non-obstructively views traffic on X.25 links. The active monitor is used to report network status and activity. [Kle91]
ALLINK Operations Coordinator	A manager of system managers. This tool can interface to management systems, including IBM's NetView, those conforming to AT&T's NMP, or SNMP to integrate heterogeneous management environments. [GHJ+89] [TWGJ91]
AUTOREC	An automated error recovery system designed and built for AT&T's packet switching networks. The system is designed to handle six different trouble reports that account for 85% of all messages. [CM90]
NETMATE	The Network Management, Analysis and Testing Environment provides a set of management tools which are the basis of a comprehensive network model. The tools include a modeller, a graphical user interface, a simulator and a network observation and control subsystem. [DSWY91]
Network Control Monitor	A tool to help manage Citibank's backbone network. The tool collects performance statistics from the network which it uses for performance evaluation. Logs are kept and used with a simulator to play back network scenarios, which greatly simplifies fault analysis. [DSY+89]

Table 1.1: Examples of management tools

1.1 Performance Management Tools

Operations staff are often employed to manage a system and support the users of the system. System problems are usually reported by users who are unable to do their work. The staff must determine what the problem is, if any, then isolate and fix it.

It would be better if the operations staff were notified of problems before users were affected by them, or were aware of symptoms that may lead to a problem so that preventive action can be taken. The system used to manage the Kansas University Packet Switch Network [Kle91] is reported to usually detect a problem before a user rings with the problem.

There are many tools available for dealing with the different facets of performance and fault management. Table 1.1 shows some of these tools. There are many commercial tools available for system management too. These include IBM's AIX NetView/6000 [CBH+92], HP's Openview, Sun's SunNet Manager, Novell's LANtern and Digital's MCCDirector to name a few.

The purpose of a performance management tool is to assist operations staff with managing a system. The tool needs to provide descriptions of the devices to be managed, metrics that are measures of resource use, and symptoms of problems.

The task of automated performance management involves the following three steps.

1. Gather performance metrics from devices under management. Metrics are either collected by polling an entity, or are reported by the entity itself.

2. Analyse these metrics to determine if there is a performance problem.

3. Should a problem be present, a suitable corrective action must be found and executed.

Each step is now considered in detail.

1.1.1 Metric Collection

While some performance managers are attached transparently to a system, most use the resources of the system. For metric collection in a distributed system, the communications network is primarily used.

A number of network protocols exist for the communication of performance metrics. The most well known of these are the simple network management protocol (SNMP) and the common management information protocol (CMIP). Others include CMOT (CMIP over TCP/IP), and a protocol called network management protocol (NMP) which is used for information exchange between components of the automated network management system, ANM, developed at BBN Laboratories [FLNO88].

The advantage of using these protocols is that they provide a uniform interface with which to gather metrics.

Another tool for gathering metrics is `rstat`[1], which gathers a fixed set of metrics from a UNIX host. Rstat is an example of a client-server application and is implemented by remote procedure call.

1.1.2 Performance Evaluation

Metrics are analysed to determine if there is a performance problem, and analysed for trends which are used to estimate long term resource use to help plan system upgrades. This analysis needs to be fast when symptoms that may lead to a problem are to be detected before the problem eventuates, or when problems are to be detected before users are affected by them.

The simplest method for detecting a performance problem, and the one most commonly used in management tools, is to compare metric values with a threshold value. The threshold represents a boundary between an acceptable and an unacceptable level of resource use.

1.1.3 Corrective Action

It has traditionally been the job of operations staff to decide what corrective action is needed to fix a problem once it has been detected. This often requires a detailed understanding of the system being managed. More recently this expertise has been engineered into rules allowing an expert system to assume some roles of the operations staff.

Introductions to the requirements and use of expert systems in the area of system management can be found in Cronk et al. [CCB88], Goldman et al. [GHJ+89], and Potier [Pot87]. Some expert systems currently in operation managing systems are:

- A PC based expert system implemented in Prolog that helps manage the Kansas University Packet Switch Network. It implements procedures to handle common network problems and forwards unresolvable problems to human operators. [Kle91]

[1] See man rstat(3R).

- YES/MVS is an expert system developed at IBM for managing the MVS operating system. Problems are detected by MVS and reported on the operator's console. The expert system processes these reports to solve problems that do not require operator intervention. YES/MVS is implemented with a modified version of OPS5, an expert system shell. [EGH+86] [MCE+86]

- NETREX is an expert system for automatically maintaining the internal data networks at Pacific Bell. NETREX examines trouble tickets and uses backward chaining to determine the "actual problem" and "recommended solution" attributes. It is implemented with the Teknowledge S1/Copernicus expert system shell augmented with C code. [GHJ+89]

1.2 Thesis Outline

This thesis is organised in two parts.

Part I, consisting of chapter 2, contains a discussion of a number of methods for detecting performance problems. These range from the simple threshold alarm to more sophisticated techniques borrowed from areas of statistics.

Part II, consisting of chapters 3–6, discusses the alarm manager. Chapter 3 gives an overview of the Control Room project and briefly describes how the alarm manager and management language interact with Control Room. Chapter 4 presents the management language, called Bark. The structures provided by the language to describe a distributed system and how it is managed are explained. Chapter 5 discusses the implementation of the alarm manager, in particular, its interaction with the rest of Control Room. Chapter 6 presents my discussion and conclusions, and the scope for future work.

Part I

Beyond Simple Threshold Alarms

Chapter 2

Performance Evaluation Methods

Performance evaluation [Lav83] [Kob87], as the name implies, is used to evaluate the performance of computer systems. In some cases this evaluation is used to determine how well a system meets the needs of its users, or to determine how changes to a system's configuration affects its performance, or in planning the expansion of a system. Performance evaluation is also used extensively in the design and implementation of new systems.

Many methods are available for performance evaluation, but not all of them are suitable for performance evaluation by a management tool. For example, queueing theory is a popular analytical tool in which a system is modelled by a set of tasks, a service, and a queue of tasks waiting for the service. The model is described by quantities such as the average rate at which tasks arrive into the queue, the number of queues, the number of servers, and the average service rate. These quantities are used to calculate performance measures such as the average task through-put, the average amount of time a task waits in a queue for service, and the average queue length. This method is unsuitable for a performance management tool because performance is described in terms of "average" quantities.

Another method frequently employed for performance evaluation is to simulate the system in operation. The nature of simulation being an "off-line" application makes it inappropriate for real-time performance evaluation. However, simulation can be used as an aid to fault analysis.

The following sections present performance evaluation methods suitable for use by a performance management tool.

2.1 Threshold Alarm Functions

This section presents the simplest of all performance evaluation techniques, the threshold alarm, and variations of it.

Let $\bar{v} = v_1, v_2, \ldots$ be a sequence of metric values such that v_i is the value of the metric at time i. Let $\bar{a} = a_0, a_1, a_2, \ldots$ be the sequence of alarm values that result from applying values from $\bar{v}$ to the alarm function, such that a_i is the value of the alarm function after it is applied to v_i. Alarm values belong to $\{0, 1\}$ where zero indicates that there is no performance problem and one indicates a possible performance problem. Metrics are said to be in an *alarm state* whenever an alarm function returns one. Let $a_0 = 0$.

2.1.1 Simple Threshold Alarms

A performance problem is indicated by the threshold alarm when a metric value crosses an alarm threshold value, $\mathcal{T}$, corresponding to a transition from an acceptable to an unacceptable

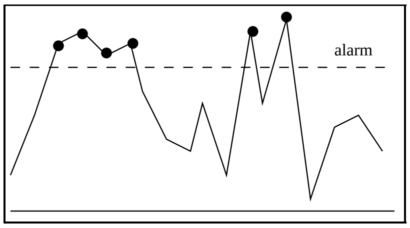

Figure 2.1: The threshold alarm

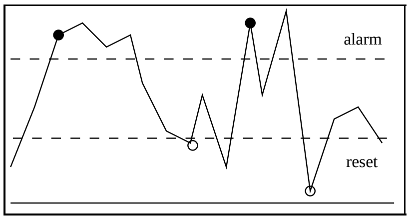

Figure 2.2: The augmented threshold alarm

level of performance. Equation 2.1 shows this alarm function for the metric value exceeding the alarm threshold,

$$a_i = \begin{cases} 1 & \text{if} \quad v_i > \mathcal{T} \\ 0 & \quad \text{otherwise} \end{cases} \tag{2.1}$$

Once a sequence of values have crossed the alarm threshold it may be some time before the sequence falls below the threshold again. The problem with the threshold alarm is that during this period the function will generate multiple alarms for the same problem.

To solve this problem the threshold alarm may be augmented with a second threshold, the reset threshold $\mathcal{R}$. As before, an alarm is generated when the alarm threshold is crossed. The difference is that no further alarms are generated until the reset threshold is crossed[1].

Equation 2.2 shows this alarm function,

$$a_i = \begin{cases} 1 & \text{if} \quad t_{i-1} = 0 \text{ and } t_i = 1 \\ 0 & \quad \text{otherwise} \end{cases} \tag{2.2}$$

where t_i is used to indicate whether the metric value is in an alarm state. An alarm is only raised on a transition from not being in an alarm state to being in an alarm state. The values of t_i are defined as follows, with $t_0 = 0$,

$$t_i = \begin{cases} 1 & \text{if} \quad v_i > \mathcal{T} \text{ and } a_{i-1} = 0 \\ 0 & \text{if} \quad v_i < \mathcal{R} \\ t_{i-1} & \quad \text{otherwise} \end{cases}$$

Figures 2.1 and 2.2 illustrate the different alarms generated by the threshold alarm functions described by equations 2.1 and 2.2 respectively. Both figures show the alarm functions applied to the same data. The black dots denote the alarm function returning one, indicating a possible performance problem. The white dots in figure 2.2 denote the augmented threshold alarm being reset.

2.1.2 The Frequency Alarm

The frequency alarm is used when the frequency with which a metric is above the alarm threshold is important. This alarm is activated when a specified number of metric values, $\mathcal{F}$, from the last n values exceed an alarm threshold, $\mathcal{T}$.

Let $a(v,t)$ denote the threshold alarm described by equation 2.1. Equation 2.3 shows the frequency alarm,

[1] This idea is stolen from IBM's AIX NetView/6000 management system. The authors use the terminology "re-arm value" where I use "reset threshold"

$$a_i = \begin{cases} 1 & \text{if} \quad \sum_{t=i-n}^{i-1} a(v_t, \mathcal{T}) > \mathcal{F} \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

To illustrate how this alarm might be used, consider a cache. When a fast device requires information from a slow device its performance is impaired by having to wait for the information. A cache is placed between the two devices to reduce the number of times the fast device must request information from the slow device. Occasionally the cache will not hold the required information which must then be retrieved from the slow device. These cache misses are to be expected and the number of cache misses may vary greatly depending on the work being done, however a large number of cache misses ($\mathcal{T}$) observed frequently ($\mathcal{F}$) may indicate a performance problem.

2.1.3 The Integration Alarm

This alarm is similar to the frequency alarm. Instead of just considering the frequency that metric values are above the alarm threshold, the degree by which they exceed the threshold is also important. The alarm is activated when the sum of the extent by which metric values exceed the alarm threshold $\mathcal{T}$, exceeds a sum threshold, $\mathcal{S}$. Only the last n metric values are used.

Let $a(v, t)$ denote the threshold alarm described by equation 2.1. Equation 2.4 shows the integration alarm,

$$a_i = \begin{cases} 1 & \text{if} \quad \sum_{t=i-n}^{i-1} a(v_t, \mathcal{T}) * (v_t - \mathcal{T}) > \mathcal{S} \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

To illustrate an application of the integration alarm, consider the packet buffers in a network router. Packets arrive at the router for redirection. When more packets arrive than can be redirected, buffers are used to hold the excess until such time when they can be redirected. The size of each packet determines how many buffers are used. Performance problems occur when the buffers become full because subsequent packets sent to the router are dropped. The integration alarm is used because both a high frequency of small packets and a low frequency of large packets may fill the buffers[2].

2.1.4 Time Variant Thresholds

An extension to the threshold alarm is to allow the threshold to vary with time. Such a threshold can be used to represent more than just a transition from an acceptable to an unacceptable level of resource use. For example, a time variant threshold can represent different "levels" of resource use. A metric crossing the threshold may still be at an acceptable level of use, but shows use outside the norm. If we observe that metric values are consistently above the threshold, say, we can extrapolate into the future to find if there is a point at which the resource becomes depleted.

[2]This can lead to an extreme degradation in performance, especially for reliable network connections where the protocols that ensure reliability repeatedly retransmit packets and packet acknowledgements that are only to be dropped. In TCP/IP, a router can "throttle" the frequency at which a host sends packets, reducing the pressure on the router.

2.2 Statistical Process Control

Statistical Process Control (SPC) [Oak86] is a tool used in industry to monitor a manufacturing process to ensure it is producing good quality merchandise.

Historically, poor quality merchandise was not usually detected until it was assembled, at which point it was scrapped or reworked at further expense to the manufacturer. The idea of statistical process control is to inspect for quality at the point of manufacture. If the products are of poor quality, the reason for this degradation is traced and corrected. This minimises the number of products that need to be reworked or scrapped.

2.2.1 Foundations of Statistical Process Control

From a manufacturer's point of view, a product is of poor quality if the parameters[3] that describe the product, such as diameter, length and colour, do not meet the product specifications.

A manufacturing process is said to be "in control" if the only factors that cause variation in a product's quality are random, and "out of control" otherwise.

Inspecting the quality of all produce is usually not possible. When done manually 100% inspection is rarely 100% effective, also the cost of 100% inspection may be too great, and if inspection is destructive then clearly 100% inspection is impractical. This means that the quality of produce must be inferred from samples.

From product samples we calculate the mean and variance of product parameters. If they differ significantly from the process mean and variance, the process is assumed to be out of control. The process mean and variance are the mean and variance of product parameters sampled from a process that is in control. They are determined when the process is first put in control. (The procedure for getting a process into control is not described here, refer to any book on statistical process control, see references section 2.2.4.) Let $\overline{\mathrm{X}}$ denote the process mean and $(\sigma_{\mathrm{X}})^2$ denote the process variance where σ_{X} is the standard deviation of a process parameter.

To determine whether or not the sample mean and variance differ significantly from the process mean and variance relies on the Central Limit Theorem. The theorem states that the means and variances of samples taken from any well-behaved probability distribution are (approximately) normally distributed.

The following sections describe techniques for detecting changes in the process.

2.2.2 Shewhart Control Charts

A Shewhart control chart, named after its inventor, is a plot of either the mean or range[4] of sample parameters verses the sample number.

The chart is annotated with two sets of control limits[5]. These are the warning limits set at $\overline{\mathrm{X}} \pm 2\sigma_{\mathrm{X}}$, and the action limits set at $\overline{\mathrm{X}} \pm 3\sigma_{\mathrm{X}}$. When the process is in control, the probability of a sample falling outside the action limit (three standard deviations from the mean) is one in one thousand. The chance of a sample falling between the warning and action

[3]The word "parameter" is used here to refer to the terms "variable" and "attribute" which are used in SPC to refer respectively to a continuous quantity (such as length) and a discrete quantity (such as number of surface scratches).

[4]Range is used instead of variance because it is easier to calculate. This was decided in the 1920s, before the invention of hand held calculators.

[5]This discussion is of the British control system which differs in detail from the American system but is equivalent.

limits is one in twenty five[6].

Based on these probabilities the following rules are used to indicate when a process needs correcting.

- When a sample falls outside either of the action limits the process is stopped for resetting.
- When a sample falls between the action and warning limits a second sample is taken and if it also falls between the action and warning limits, the process is stopped for resetting. The probability of such an occurrence is $1/25*1/25 = 1/625$ when the process is in control.
- Five points in a row in the same direction between $\overline{\mathrm{X}}$ and one of the warning limits indicates a process shift, as the probability of such a run is about 5% when the process is in control.

In practice, a control chart is used for both mean and range of process parameters because the process is in control only when both the mean and the spread are in control.

2.2.3 Cumulative Sum Control Charts

Shewhart charts essentially use each sample plot separately to determine if a process is out of control. Other charts have been developed that take more information into account but the chart that uses all available information is the cumulative sum, or cusum chart.

The cusum chart is a plot of cusum score (S_n) verses sample number. The cusum score is calculated as follows,

$$\begin{aligned} S_n &= \sum_{i=1}^{n}(x_i - t) \\ &= S_{n-1} + (x_n - t) \end{aligned} \tag{2.5}$$

where

S_n is the cusum score for the n^{th} sample,

x_i is the result from sample i (such as the sample mean, $\overline{\mathrm{x}}_i$), and

t is called the target value.

The choice of target value depends on the application. It might be the process mean, the specification mean or the forecast for the result of the sample.

Process change is determined by the gradient of the plot. When the process is in control $x_i \approx t$ and the plot is horizontal, otherwise the plot has some non-zero gradient. The steeper the gradient, the greater the change in the process parameters.

Methods for detecting significant statistical change, indicating a process change, include the 'V' mask, parabolic mask and parallel mask. Masks require careful design. Change is indicated when the mask covers some part of the control chart. One method for detecting change that doesn't use a mask, the decision interval, is now described.

[6] Oakland ([Oak86], page 71) states this probability is 1 in 40. However, for $N(0,1)$, $P(2 \leq z < 3) = 0.0215 \approx 1/50$

The Decision Interval

This method is used to detect a process change in one direction only. The amount of change can be compared with a predetermined amount, the decision interval h, and corrective action taken when this value is exceeded.

A new formula is used to calculate cusum scores,

$$\begin{aligned} S_n &= \sum_{i=1}^{n}(x_i - k) \\ &= S_{n-1} + (x_n - k) \end{aligned} \tag{2.6}$$

where k is a reference variable set half way between the target value and a level called the reject quality level (RQL), a point at which an unacceptable quantity of reject merchandise is produced (such as three standard deviations from the process mean).

A different charting method is used with the decision interval. When the process is in control the chart will grow in one direction. The chart grows positively if we want to detect a decrease in x_i (RQL $< k < t$ so $x_i - k > 0$), and negatively if we are detecting an increase in x_i (RQL $> k > t$ so $x_i - k < 0$). For this reason, the chart is discontinued whenever the cusum score rises above, or drops below zero respectively, and restarted with the first positive or negative result only.

2.2.4 References

Berger and Hart [BH86] give a short but concise overview of the topic, Oakland [Oak86] provides a very readable detailed study of statistical process control and Murdoch [Mur79] has a detailed discussion of cumulative sum charts and uses decision intervals extensively in his examples.

2.3 Performance Evaluation with Statistical Process Control

Statistical process control is used in industry to ensure a manufacturing process is producing good quality merchandise. Control charts are used to show when a process is out of control and needs correcting. The aim of this section is to look at how statistical process control can be used to monitor a distributed system and determine when the system is not performing well.

2.3.1 Previous Work

The MAP[7] Fault Management system (MAPFaM) [JKM+89], developed at the Industrial Technology Institute, is an evolving expert system for fault management of MAP and other OSI networks.

The MAP fault management system is designed to use adapted SPC techniques to monitor trends in error parameters and detect unusual patterns of activity which may be analysed further to determine the presence of a fault.

The system developed at the Industrial Technology Institute has two components, for configuration and fault management. The fault manager has a component for analysing trends and identifying excessive changes in counter values. The authors observe

[7] The manufacturers automation protocol (MAP) is a communications protocol used for data exchange in manufacturing networks.

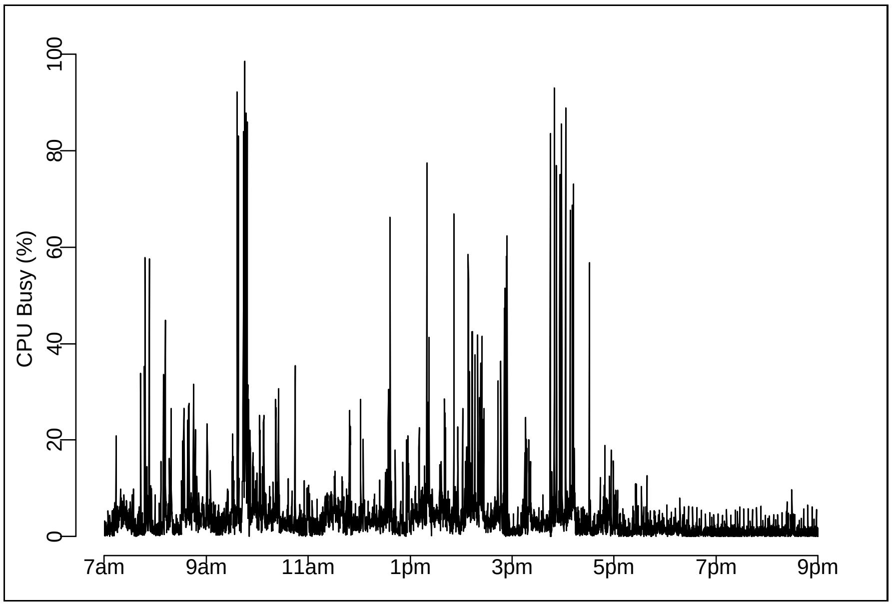

Figure 2.3: Cpu use for the host `debretts` on 30 October, 1992

> *The trend analysis mechanisms are not appropriate for all network counters. For example, some counters (e.g., number of octets sent and received) can normally be highly variable and generated a large number of "false" events.*

No further discussion of adapted SPC techniques is given [TFJS90]. The trend analysis component of the fault manager is a small part of what is primarily an expert system. Statistical process control was only experimented with to a limited degree [TF93].

2.3.2 Differences between Industrial and Performance SPC

There are a number of important differences between SPC used in industry and SPC used for monitoring the performance of a distributed system.

1. Industrial processes fluctuate with small variance about a mean value whereas a number of performance metrics have a large variance. This is illustrated by figures 2.3 and 2.4. Figure 2.3 shows the cpu busy metric for the host called debretts[8] on the date shown, and figure 2.4 shows the Shewhart control chart that results from this cpu usage. The chart was constructed by taking fixed sized samples of the metric at regular intervals and plotting the mean of each sample verses the sample number. A system administrator would probably not be concerned by this sample of resource use and conclude that there was no performance problem, however, it is clear that in Shewhart's terms, the process is not in statistical control.

2. Complete inspection of the system is possible in the sense that metrics may be gathered from the system as desired. To literally have 100% inspection would require reading counters every time a new value was stored, which would significantly load the machine. This raises the issue of how frequently the counters should be sampled, taking into consideration the load placed on the system verses the level of detail required.

[8] Debretts is a Sun Sparc station IPX running SunOS UNIX release 4.1.3

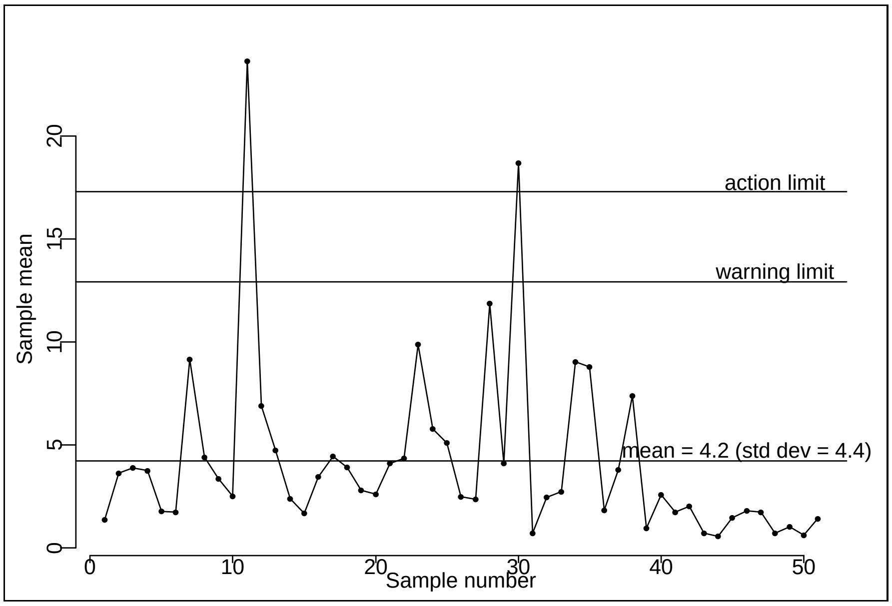

Figure 2.4: The Shewhart control chart for cpu use

3. In industry, product sampling is independent of the manufacturing process. However, in computer systems, performance managers generally use system resources (cpu, disk, network, etc.) and as such, both impacts the performance of the system and has its performance degraded when the system becomes loaded. Performance managers are usually run on dedicated machines to reduce this problem.

2.3.3 Observations

As mentioned in the previous section, some metrics are unsuitable subjects for statistical process control because they vary too much. This observation supports that made by the authors of MAPFaM.

The reason why metrics have this high variance is that they reflect the interaction of the users with the system, which is highly variable, not nearly as simple and regular as machine behaviour.

However, all is not lost. Alwan and Roberts [AR89] discuss the idea of combining time series models with statistical process control techniques. The authors state that using models, such as those described by Box and Jenkins, provides a picture of what the process is doing which may give insight into the process.

Two charts are now used,

- the *common cause chart* that is the time series model for the process, and
- the *special cause chart* (a plot of the residuals from the model) which is used in the traditional way to detect the process going out of control.

To use this scheme for system performance evaluation, a model of metric use is required. These models, called time series models, are discussed in the next section.

2.4 Time Series Models

A great deal of data consists of observations made over time. The way in which these observations are related is often of interest. Time series analysis is applied to observations to derive a model which describes the relationship between the observations and time. The goal of building such models is usually to make forecasts for future values and it is this property that is useful in performance evaluation. If accurate forecasts are available, they can be used to evaluate the performance of the system by detecting problems before the problems occur.

There are many different types of model. The selection of a model depends on a number of factors that include the complexity of the data being modelled, the accuracy with which the models should represent the data, and the confidence of the user with a model.

2.4.1 Box and Jenkins Models

Some of the more sophisticated models are described by Box and Jenkins [BJ70]. Their objective is to derive models of maximum simplicity and with a minimum number of parameters.

Models can be described as stationary, which assumes the underlying process remains in equilibrium around a fixed mean, or non-stationary, where trends and characteristics in the underlying process change with time. Non-stationary models can be described as seasonal or periodic.

The two basic models are the moving average and autoregressive models. In the autoregressive model, the current value is expressed as a finite linear aggregate of previous values of the process and an error term a_t. An *autoregressive* (AR) *process of order p* is denoted by

$$v_t = \phi_1 v_{t-1} + \phi_2 v_{t-2} + \ldots + \phi_p v_{t-p} + a_t \tag{2.7}$$

where $\phi_1, \ldots, \phi_p$ are unknown parameters that are estimated from the data.

In the moving average model, the current value is linearly dependent on a finite number of previous error terms. A *moving average* (MA) *process of order q* is denoted by

$$v_t = a_t - \theta_1 a_{t-1} - \theta_2 a_{t-2} - \ldots - \theta_q a_{t-q} \tag{2.8}$$

where $\theta_1, \ldots, \theta_q$ are unknown parameters that are estimated from the data.

The main criticism of these models is that they are not intuitive. The same models are applied to data from many different sources ranging from stock prices to the number of sun spots. Andrew C. Harvey [Har81] developed a modelling paradigm that is equivalent to, but more intuitive than the models of Box and Jenkins.

The model described by Harvey consists of three components, a trend which is the "average level" of the data over time, a cycle which is how the data fluctuates about the trend, and noise which is whatever is left over.

2.5 Time Series Models in Performance Evaluation

2.5.1 Previous Work

The following paragraphs describe the application of models in performance evaluation.

- Fassina and Quaroni [FQ87] describe work they did in predicting the use of different resources on an IBM 3083 running MVS, based on the workload to process.

  Resource use for a number of different resources were sampled every half second during a normal days operation. A strong correlation was found between the number of

TSO[9] connections and the use of other resources indicating that the number of TSO connections was a good estimate of the workload to process.

Regression analysis was successfully used to produce a model of busy cpu from which forecasts, based on the number of TSO connections, were accurately made.

- Hellerstein [Hel93] gives a tutorial introduction to Box-Jenkins models. He then presents a case study illustrating the use of time series analysis in diagnosing a performance problem in a large time sharing system.

  System response times were measured every half minute on days when performance was bad. After several attempts at modelling the time series, an AR(1)[10] seasonal model was fitted to the data. This model indicates the presence of cycles, which appeared every ten minutes in the response time data. This information allowed operations staff to track down and the fix the problem, which was caused by an inefficiently written program that executed every ten minutes.

2.5.2 Modelling Performance Metrics

To use the extension to statistical process control described by Alwan and Roberts requires a model of metric usage. Building such a model also allows metric values to be forecast. This section describes how I tried to build a general model for metric values.

This model is unlike the models described above. The model of Fassina and Quaroni produced forecasts for values of a metric, a_i, given the values of another metric, b_i. To use this model to forecast values of a_i requires accurate forecasts for b_i. The model of Hellerstein describes the performance of a system on one particular day and is not a model for the general performance of the system.

The metrics for the general model were collected from one host, `debretts`, over a period of three months[11]. These metrics were sampled every 15 seconds and included cpu use, network traffic to and from debretts, disk use and device interrupts. The longest unbroken run of data was 59 days long and this run was used to build the model.

The number of device interrupts and the busy cpu metrics were used to build models. I produced summaries of these metrics, given by the median metric value over hourly and daily intervals. The `sabl` function of the Splus statistical package was then applied to the summaries.

The `sabl` function decomposes a time series into three component time series, the trend, seasonal and irregular time series, like those in the models of Harvey. Figure 2.5 shows the results of decomposing the summarised cpu busy metric.

The data plot shows the original time series for the 59 days, and the other three plots show the trend, seasonal, and irregular components. The time series was not transformed before being modelled.

Notice that the scale for the trend and seasonal plots is much smaller than the scale for the irregular plot; the irregular component is not insignificant.

After fitting a function to the trend and seasonal components of the model, the next step was to model the irregular component, which consists of high valued metrics, showing high resource use. Modelling should be done for the original data as forecasts for the median use of a resource in the next hour or day are of limited use in detecting performance problems.

[9] The time sharing option (TSO) provides users with interactive editing, testing and debugging capabilities. TSO can also be used to view output from a batch job, and to submit work to the system ([Dei90] page 674).

[10] AR(1): an autoregressive model of order one, i.e., $v_t = \phi v_{t-1} + a_t$

[11] It should be noted that this "general" model of resource use is only applicable to `debretts`. To produce models for other hosts requires sampling data from those hosts.

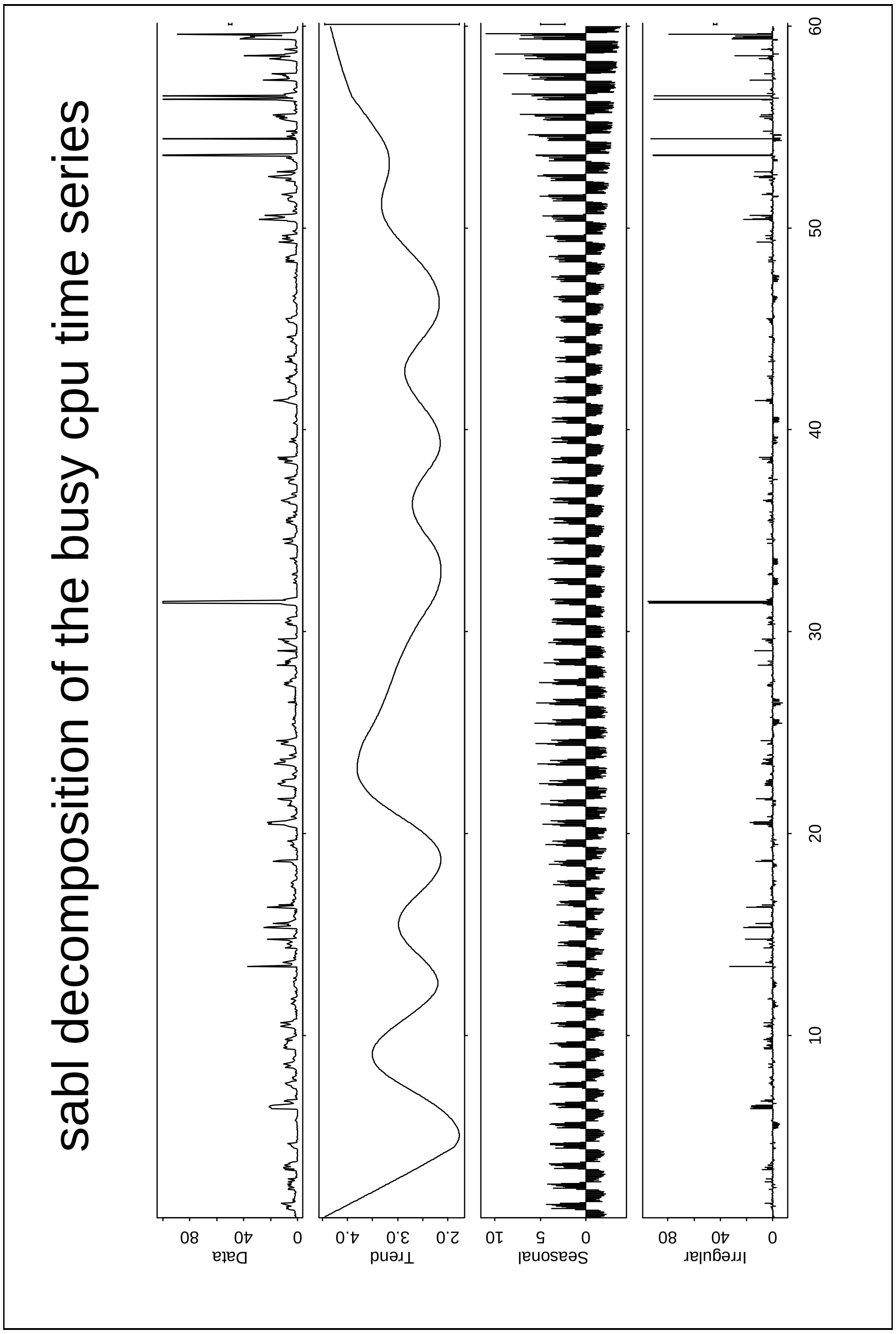

Figure 2.5: Sabl decomposition of the cpu busy metric

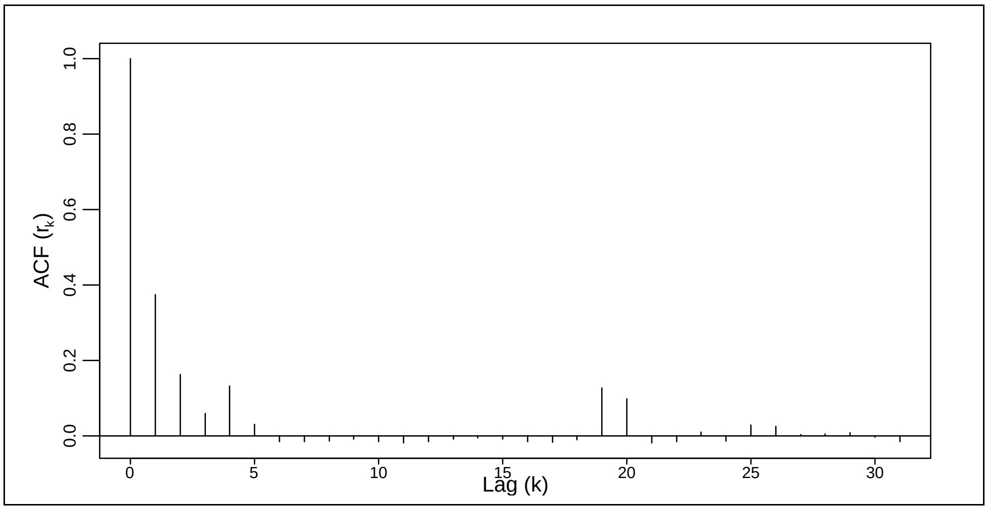

Figure 2.6: Autocorrelation function for the irregular cpu busy

The autocorrelation function (acf) was applied to the irregular component to determine if successive values were correlated. The acf is analogous to the correlation function that finds the degree of correlation between pairs of values for two variables. An autocorrelation coefficient, r_k, is calculated for pairs of values at lag k (k time steps apart). The formula for the autocorrelation coefficient at lag k is

$$r_k = \frac{\sum_{t=1}^{N-k} (x_t - \overline{x})(x_{t+k} - \overline{x})}{\sum_{t=1}^{N} (x_t - \overline{x})^2} \tag{2.9}$$

where

N is the number of observations,

$x_1, \ldots, x_N$ are the observations, and

$\overline{x}$ is the mean of these observations.

An aid to examining these coefficients is the correlogram, which is a plot of r_k verses k. Figure 2.6 shows the correlogram for the irregular component of the summarised cpu busy metric. If the data were randomly distributed we would expect $r_k \approx 0$ for all non-zero k. The correlogram shows there is a correlation between consecutive values (lag 1) and a weak correlation between values at lags 2 and 4. There is also a weak correlation at lags 19 and 20, that is between values 5 minutes apart. Interpreting a correlogram, reducing the effects of noise, and building a model from the information it provides requires considerable experience, and is beyond the scope of this work.

2.6 Conclusions

Alwan and Roberts describe a way of combining statistical process control techniques and modelling techniques (see the Observations section 2.3.3). The assumption made by Alwan and Roberts is that significant statistical deviation is due to the process going out of control, as indicated by the special cause chart (the residuals of the process model). In the context of automated manufacturing, the topic of the book in which their paper appears, this assumption is reasonable.

Another cause of deviation is that the model of the process is not very good. Processes that involve considerable human interaction or don't limit the extent of interaction are hard to model because human behaviour is highly variable, not nearly as simple and regular as machine behaviour. This variability is reflected in metric values which measure resource use resulting from human activity.

Despite this, some metrics, such as the amount of free disk space, may be suitable for statistical control or modelling as they are not so variable over a short period of time. This is a topic for future work.

More work can be done transforming metric values to reduce the high variation. Transformations such as the logarithm and exponential smoothing could be used. These transformations may also be of help in trying to improve the decomposition of a time series into component series.

Other observations were made while carrying out this work.

- It would also be interesting to look at how certain factors, such as the number of users and certain jobs or job mixes affect the performance of the system.

- Another point not addressed by Alwan and Roberts is the apparently conflicting assumptions of statistical process control and modelling. Statistical process control requires that data is sampled randomly so that any cyclic factors which affect production are detected, whereas model building requires that data is sampled at fixed intervals. This is not really as problem as values can be interpolated from the common cause chart (the model).

  It is possible to build a model from data sampled at irregular intervals, but it is more difficult. If the distribution is known, samples can be taken non-periodically for efficiency. For example, samples can be taken from a computer system less frequently at night when the system is used less, and more frequently during the day when the system is used more.

- Building a model of resource use requires a considerable amount of work. Ideally, a performance manager would automatically identify and build them, otherwise an expensive human is required. Tools are available for automatically identifying and modelling data. Autobox [Shu86] is an example of such a tool and builds Box and Jenkins models.

  We also want model parameters to adapt to reflect changes in the system, for example, when the system has a higher capacity disk installed or is upgraded to a faster communication link. The area of Kalman filtering [CC87] [Har89] might be applicable. Kalman filtering was originally used in rocket guidance systems to correct the flight path of a rocket that strayed from its course.

- The discussion in this chapter is biased towards the detection of resource overuse. These techniques can be used equally well to detect resource underuse, and may be used to guide system reconfiguration so that load between components is more evenly distributed.

2.6.1 Comparison of some Alarm Functions

This section presents a brief comparison of some of the alarm functions described in this chapter. The threshold alarm, augmented threshold alarm, frequency alarm, integration

i	x_i	x_i - k	S_i	i	x_i	x_i - k	S_i
1	12.1		12.1	12	99.3	14.3	103.8
2	11.3	-72.9	0.0*	13	99.6	14.6	118.4
3	6.4		6.4	14	99.2	14.2	132.6
4	17.2	-67.8	0.0*	15	99.1	14.1	146.7
5	99.4		99.4	16	58.7	-26.3	120.4
6	100.0	15	114.4	17	16.9	-68.1	52.3
7	63.8	-21.2	93.2	18	16.6	-68.4	0.0*
8	37.2	-47.8	45.4	19	82.7		82.7
9	99.2	14.2	59.6	20	26.7	-58.3	24.4
10	99.9	14.9	74.5	21	22.4	-62.6	0.0*
11	100.0	15	89.5				

Table 2.1: Cusum scores for the cpu busy metric

alarm, and a cusum alarm were applied to some metric values sampled from debretts. The bottom of figure 2.7 shows the cpu busy metric for 23 October, 1992 between 11:15am and 12:15pm.

The plots above cpu busy show the alarms generated using the different alarm functions. Line (a) shows the alarm generated by the threshold alarm with the threshold set at 85, as indicated on the figure. Line (b) shows the alarms generated by the augmented threshold alarm with threshold 85 and reset threshold set at 60, as indicated on the figure. Line (c) shows the frequency alarm. An alarm was triggered if four or more of the last eight values were above 85. Line (d) shows the alarms generated by the integration alarm. An alarm was generated when the degree by which eight consecutive values that exceeded 85, exceeded 99. The alarms generated by the frequency alarm and integration alarm can be changed by using different values for the number of values sampled, n, and the thresholds associated with the alarms. Finally, line (e) shows the alarms produced by the cusum alarm function. The choice of the value for the reject quality level, 170, was ad hoc. This gives $k = 85$. The cusum scores were calculated from the metric data and an alarm was generated for each score above the decision interval, 99.

Figure 2.8 shows the cusum scores and the metric values over the five minute interval in which alarms were generated. The solid line is a plot of the cusum scores and the dotted line is the plot of the metric values. Both use the same scale.

Table 2.1 shows the data used to derive this chart. The column labeled x_i has the metric values, and the column labeled S_i has the cusum scores. The cusum scores marked with an asterisk denote where the chart was discontinued because the cusum score fell below zero. The metric value x_1 was sampled at 11:43:00am, x_2 was sampled at 11:43:15am, and so on.

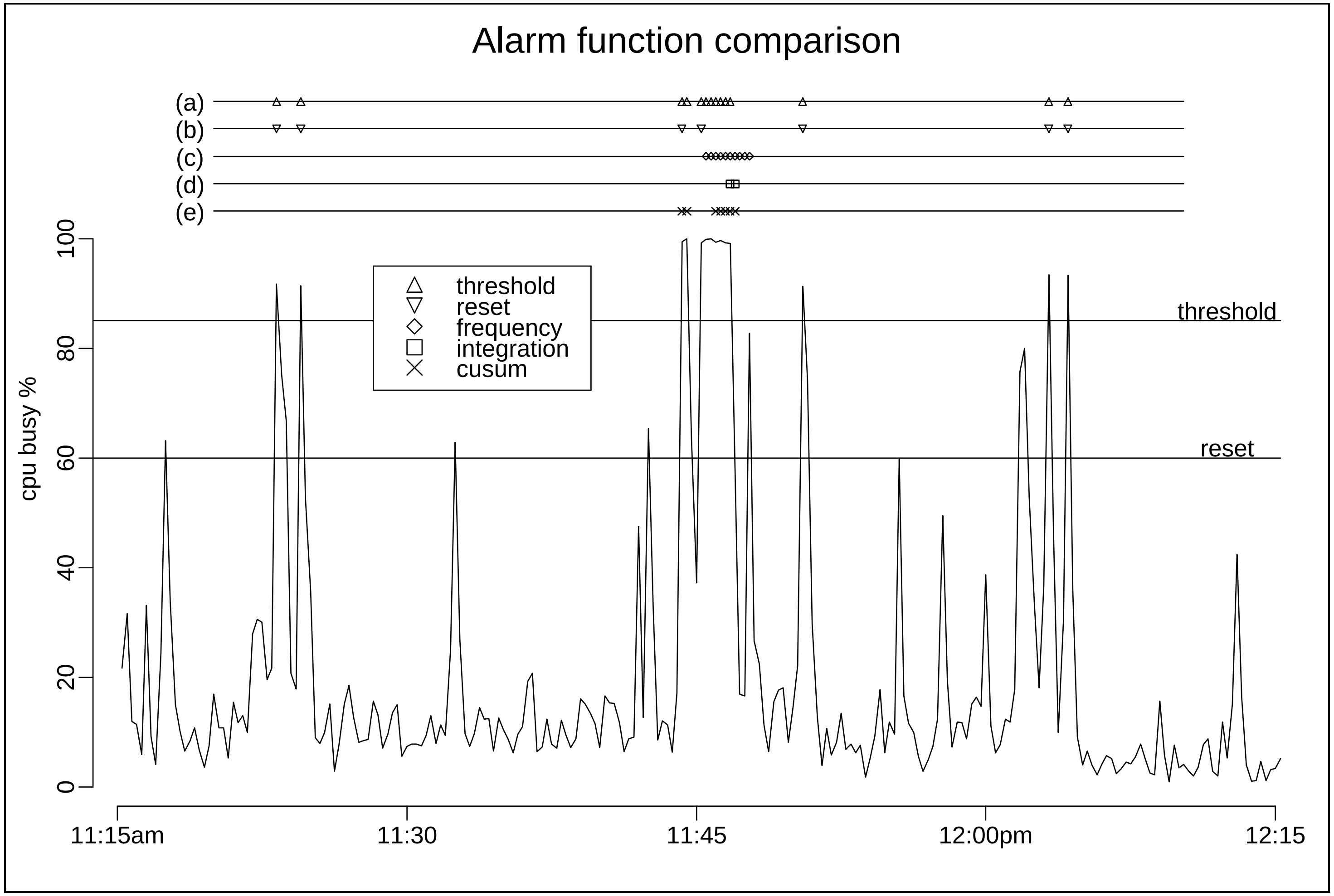

Figure 2.7: Comparison of the alarm functions

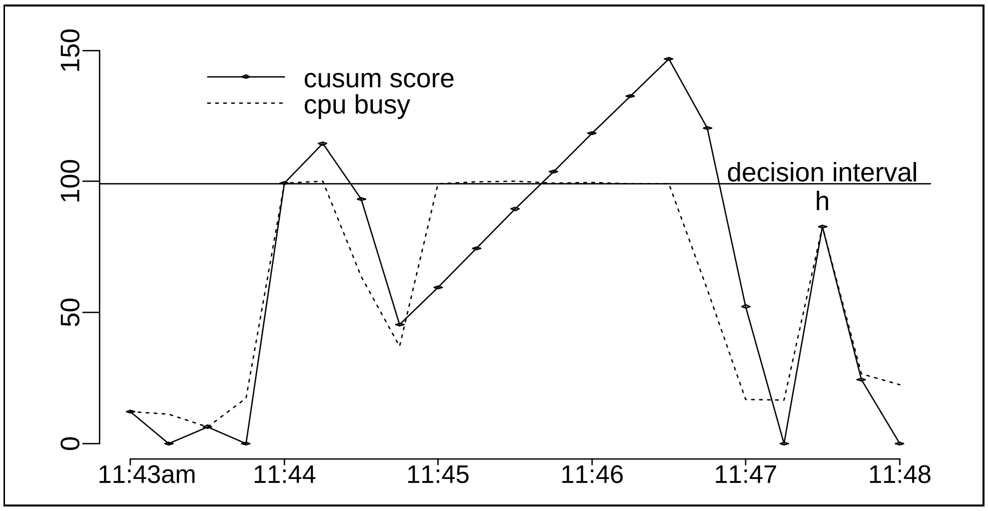

Figure 2.8: Cpu busy data and cusum scores

Part II

The Alarm Manager

Chapter 3

Control Room

3.1 Control Room

Control room (CR) [Zha93] is a system for displaying information about resource usage in a distributed system.

The basic system consists of three processes, the gatherer, manipulator and display connected by two shared memory segments. Figure 3.1 shows the organisation of these processes.

The gatherer collects performance metrics from entities under management. Multiple gatherers can be used to collect metrics from different sources. Currently one gatherer uses rstat to collect data, and another uses the simple network management protocol. The gatherer stores these metrics, called *raw* metrics in the raw metrics shared memory segment. The manipulator applies user supplied functions to raw metrics and stores the new derived metrics, called *cooked* metrics, in the cooked metrics shared memory segment. The display process displays cooked metrics on dial, bargraph and stripchart meters. The display is organised in such a way to show how the system is being used and allow easy comparison of resource usage.

The file used to configure the gatherer specifies metrics to be gathered and the entities from which they are to be gathered, and is also used to produce the index for the location of metrics in the raw metrics memory segment. The configuration file for the manipulator specifies various transformation functions that are applied to these metrics, and is also used to generate the index for the location of cooked metrics in the cooked metrics shared memory segment.

Control Room also has a logging process for storing metrics from either the raw or cooked metric memory segments, and a playback process for injecting logged metrics into the CR pipeline.

Control Room is an ongoing project within the computer science department. The eventual goal of the project is to have a tool that regulates the use of resources so that problems due to contention are reduced. The system is designed so that the processes are independent of each other. Components can be modified or completely replaced without affecting other components in the system. The ability to easily plug in and play with different components makes control room a very useful research tool.

My contribution is the alarm manager, which is used to detect and report performance problems, and a management language which allows symptoms of problems to be expressed, using tests like those described in chapter 2. The language integrates the components of CR by producing a configuration file for each component of the system. The alarm manager and management language are briefly presented in the following sections, but first is a discussion of the display.

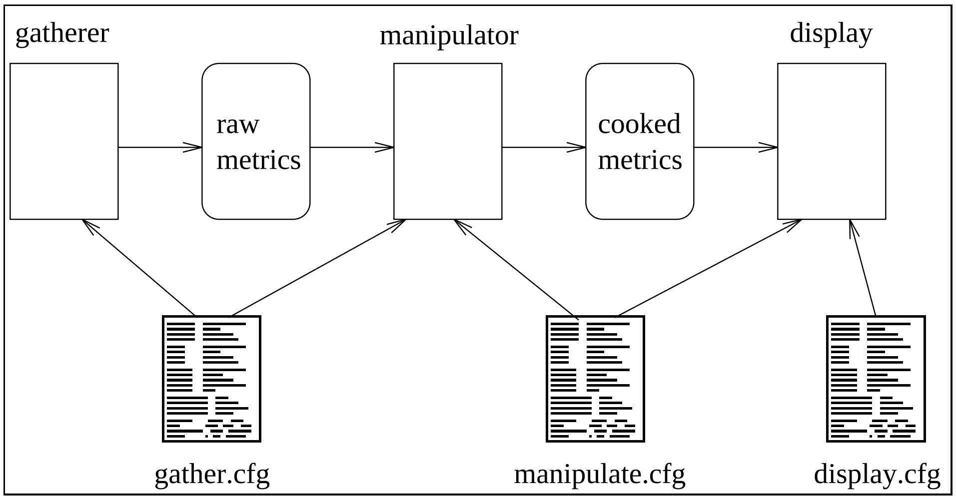

Figure 3.1: The original Control Room project

3.1.1 The Display

The display process uses meters to present information about resource use in a distributed system. The display is organised as a grid, with all resources associated with a host in the same row, and the same metric displayed with the same meter down the columns.

Figure 3.2 shows a display of nine metrics for two hosts.

Metrics can be displayed with either a bargraph, dial or stripchart meter, each of which are illustrated in the figure. Each meter also has a label. The row of meters displaying metrics from a host also has a label.

So that the display isn't cluttered, rows and columns of meters can be temporarily hidden and brought into view when desired.

3.2 The Parser

The management language, called Bark, is used for describing entities that provide resources, metrics that are measures of resource use, and rules consisting of conditions that describe the symptoms of performance problems and actions to execute when these conditions are true. A description of how to display metrics is also provided by the language.

A file written in this language is transformed by the Bark parser to produce new configuration files for all of the control room processes and the alarm manager process. The parser also produces the user supplied library of manipulation functions for the manipulator, and a set of functions to extend the functionality of the alarm manager. Both the manipulator and the alarm manager must be recompiled to incorporate these functions. Figure 3.3 shows the bark parser and the files it creates.

Chapter 4 discusses the bark management language in detail.

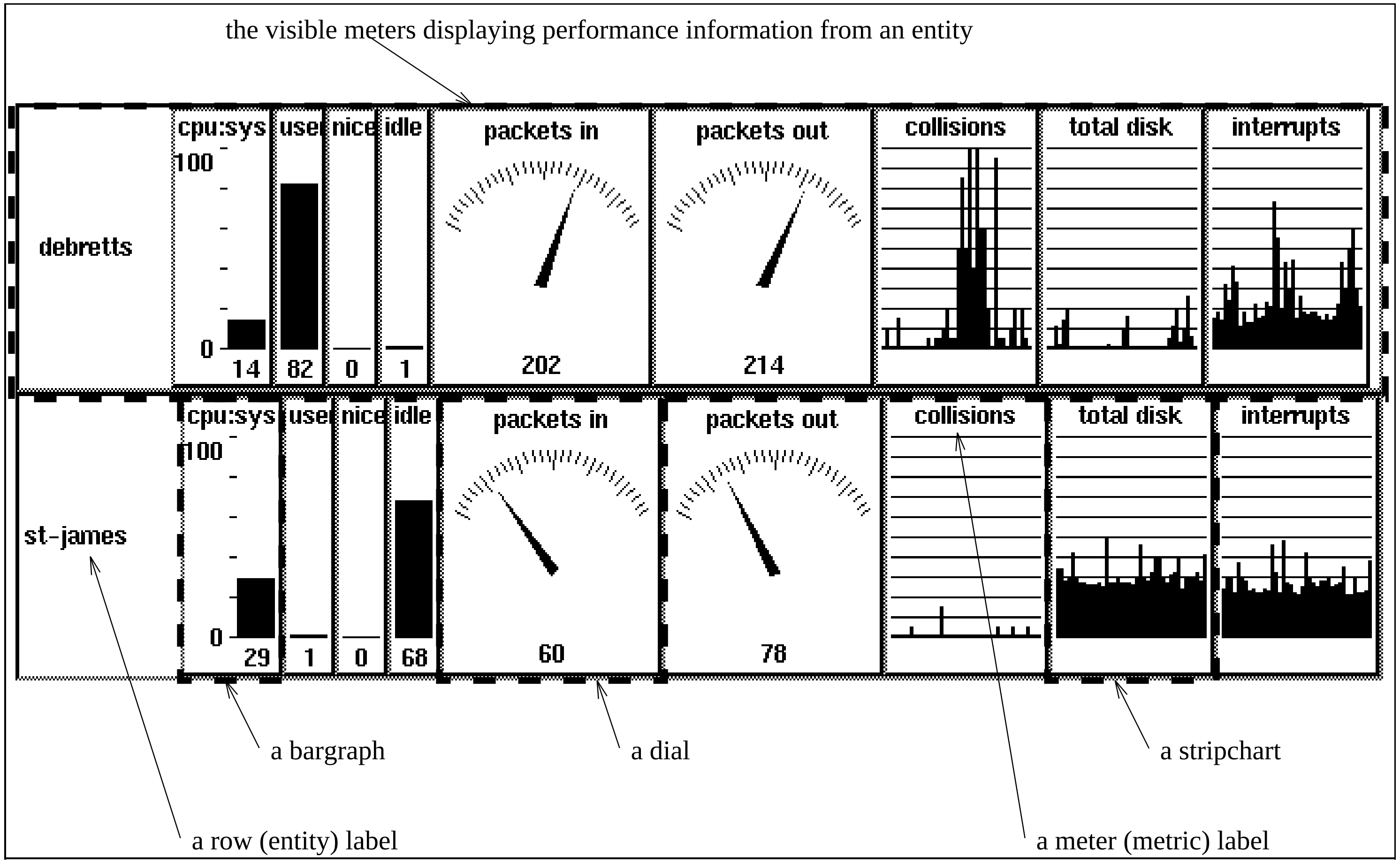

Figure 3.2: The organisation of the display

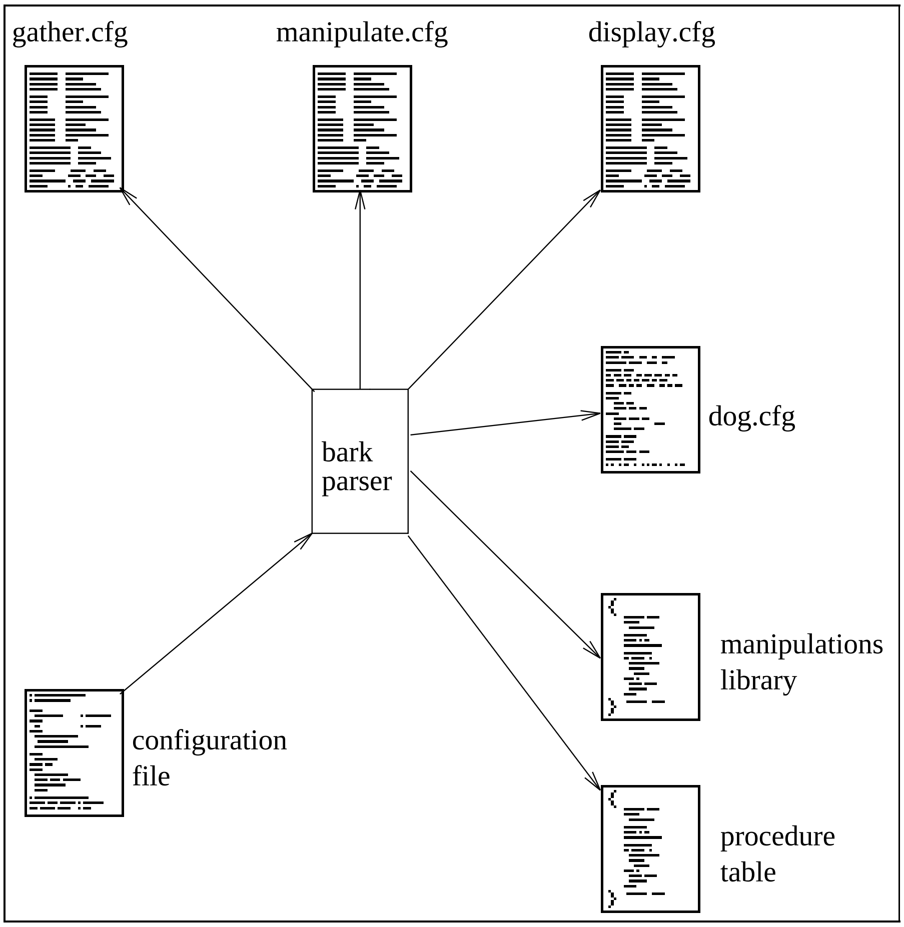

Figure 3.3: The Bark parser

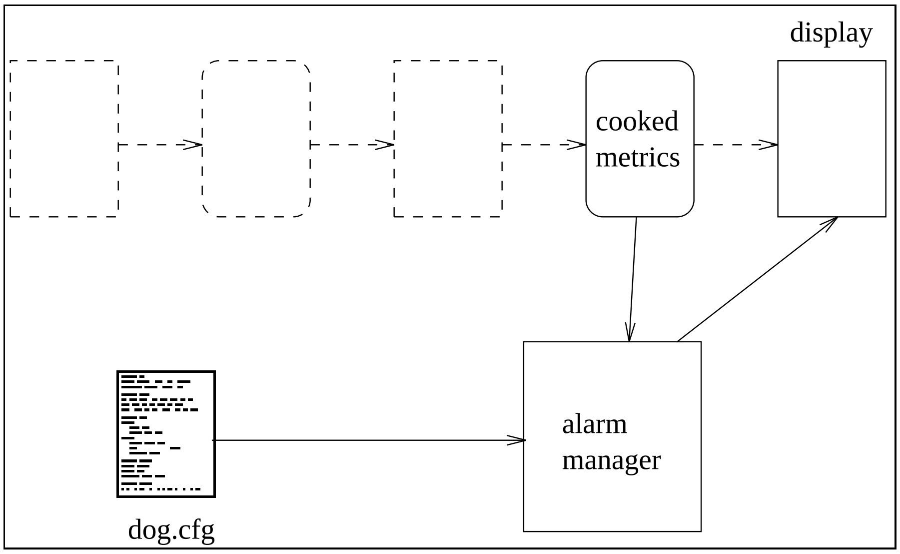

Figure 3.4: The alarm manager

3.3 The Alarm Manager

The alarm manager, called Dog[1], is the component that detects and reports performance problems. The basic function of the alarm manager is to monitor metrics and raise alarms when these metrics indicate a performance problem, based on the rules written by operations staff. Figure 3.4 shows the alarm manager in relation to control room.

The information used by the alarm manager to detect symptoms of performance problems is obtained from the cooked metrics shared memory segment. The method used to access the segment is the same as used by the CR processes. Because all processes communicate through shared memory they must all run on the same machine.

The alarm manager is also able to communicate with the CR display process to change the colour of display meters. This is a visual aid to indicate the presence of performance problems.

Chapter 5 discusses the implementation of the alarm manager.

[1]The name came from the saying that goes something like "A dog is Mans' best friend". This dog is the system administrator's best friend.

Chapter 4

The Bark Management Language

The Bark management language allows system administration staff to describe entities to be managed, metrics that are measures of resource use, symptoms of performance problems, and actions to execute when the symptoms are observed. Administrative staff can also describe how entities and metrics are to be displayed by the CR display process.

The language is designed to be flexible so that a wide variety of conditions can be expressed, using arithmetic, logical and boolean operators and if necessary augmented with code written in the C programming language. Bark is also designed to assist the process of developing a configuration file by allowing information common to many entities to be declared once in a class and having this information inherited by the entities.

The complete Bark grammar is provided in Appendix A.1. Portions of the grammar are included in the discussion of the language, please refer to the appendix for an explanation of the notation.

4.1 Declarations and Constants

4.1.1 Names

Names are used to identify components of the language such as entities, metrics and rules. Names can only contain upper and lower case letters (which are treated distinctly), digits and the underscore character (_). A name must begin with a letter or the underscore character.

4.1.2 Numbers

Numbers can be integer or real.

Integers can be expressed as binary, octal, decimal or hexadecimal sequences of digits. Unless otherwise indicated, decimal numbers are assumed. To express a number in binary, octal or hexadecimal a prefix must be attached to the number. For binary numbers this prefix is `0b`, for octal numbers the prefix is `0c` and for hexadecimal numbers it is `0x`. For example, the decimal number `107` is expressed in binary as `0b1101011`, in octal as `0c153` and in hexadecimal as `0x6b`.

Real numbers comprise a mantissa and an exponent which are expressed with decimal digits. The mantissa has a whole part and a fractional part separated by a period. The mantissa and exponent are separated by either a lower case `e` or an upper case `E`. The exponent may be proceeded by a hyphen (−) to indicate that it is negative. There are three legal combinations of the mantissa and exponent used to form a real number:

$$w_1 \ldots w_n \,.\, f_1 \ldots f_m \;\; \mathrm{E} \; e_1 \ldots e_p \qquad \text{e.g. } 6.626\mathrm{e}{-}34$$
$$w_1 \ldots w_n \,.\, f_1 \ldots f_m \qquad \text{e.g. } 3.141592654$$
$$w_1 \ldots w_n \;\; \mathrm{E} \; e_1 \ldots e_p \qquad \text{e.g. } 3\mathrm{E}8$$

where w_i is a digit in the whole part of the mantissa, f_i is a digit in the fractional part of the mantissa and e_i is a digit in the exponent.

4.1.3 Strings

A string is declared by enclosing text in quotation marks. For example `"this is a string"` is a valid string. There are special "escape sequences" for inserting non-printable characters into strings. An escape sequence is a backslash (\) followed by a letter. The table shown below summarises these sequences.

Sequence	Replaced with
\a	alert (bell) character
\b	backspace
\f	form feed
\n	newline
\r	carriage return
\t	horizontal tab
\v	vertical tab

Any other character prefixed with a backslash is replaced with that character. In particular, the sequence \\ is replaced with the character \ and the sequence \" is replaced with the character ".

4.1.4 Time Periods

A time period is specified by giving an integer followed by a time unit. The time unit is specified by one of the following keywords: `millisecond`, `second`, `minute`, `hour` or `day` which may be abbreviated respectively to `msec`, `sec`, `min`, `hr`, and `dy`. The time units and their abbreviations can also be appended with the letter `s`.

4.1.5 Specification of Output File Names

The Bark parser produces nine output files in the process of translating a configuration file written in the Bark language. Six of these files are for configuring the gatherer, manipulator, display and dog processes, and the other three are user supplied C code to be compiled into the manipulator and dog. There are many configuration files because each process is configured separately to maintain process independence.

The names of the configuration files and the directories to place files containing user supplied functions can be specified. A new name or path is given in a string following one of the variables given below.

`gather_file` This file configures the gatherer process. The default name for the configuration file is `./gather.cr`.

`manip_file` This file configures the manipulator process. The default name for the configuration file is `./manipulator.cr`.

`display_file` This file configures the display process. The default name for the configuration file is `./display.cr`.

`manip_names` This file contains a list of the manipulation functions used by the manipulator. The default name for this file is `./manipMetricNames`.

`config_file` This file contains a description of the classes, entities, rules, metrics, etc. that were declared in the configuration file written in the Bark language. The default name for this file is `./dog.cfg`.

`action_file` This file describes the actions to execute when rule conditions are true. The default name for this configuration file is `./actions.cfg`.

`cr_path` This path specifies where the user supplied manipulation library is to be written. The name of the file containing these manipulations is `manip_funs.c`. By default this file is written to the current working directory.

`dog_path` This path specifies where the file of functions to extend the functionality of dog are to be written. The file called `procedure_code.cxx` contains the function code and the file called `procedure_table.cxx` declares an array of pointers to these functions. By default these files are written to the current working directory.

For example,

```
gather_file
"my_gather.cr"
dog_path "/u/dog/src/"
```

specifies that the configuration file for the gatherer process is called `my_gather.cr`, and the files called `procedure_code.cxx` and `procedure_table.cxx` are written to the directory called `/u/dog/src/`.

4.2 Expressions

Expressions are used in metric declarations for deriving the value of cooked metrics, and in rules to express symptoms of problems.

An expression is made up by applying operators to operands.

4.2.1 Operators

The operators consist of arithmetic, boolean and relational operators. The basic arithmetic operators are *add* (+), *subtract* (−), *multiply* (∗) and *divide* (/). There is a *unary minus* operator, the *bitwise and* (&) and *bitwise or* (|) operators, and the *absolute value* operator. The absolute value of an expression is specified by putting vertical bar (|) characters around the expression. For example, the absolute value of $x - t$ is denoted by $\mid x - t \mid$. There are the relational operators *less than* (<), *less than or equal to* (<=), *equal to* (==), *not equal to* (!=), *greater than or equal to* (>=) and *greater than* (>). Lastly are the boolean operators *and* (&&), *or* (||) and *not* (!).

Arithmetic, boolean and relational operators can be freely intermixed as governed by their associativity laws. The operand resulting from a boolean expression is interpreted by an arithmetic operator as the number one or zero corresponding to a true or false result respectively. Similarly, a non-zero result is interpreted as true by a boolean operator, and the value zero is interpreted as false.

The associativity and precedence of these operators is given in the table below, except for absolute value which is unambiguous. Operator precedence increases down the table.

Operator	Associativity
|| |	left
&& &	left
< <= == != >= >	non-associative
+ −	left
− (unary)	right
* /	left
!	right

4.2.2 Operands

Operands can be numbers, described in section 4.1.2, or metrics.

A metric specification consists of a metric name and the entity from which the metric is to be taken. Refer to section 4.8 for details of metric operands.

4.3 Entities, Classes and Inheritance

The declaration of an entity contains the declaration of a set of metrics which are a measure of resource use, a set of rules to manage the entity and information specifying how each metric should be displayed by the CR display process.

It is almost certain that some entities will be related, in the sense that they provide the same type of resource, and thus have common metrics and rules. For example, all file servers have a metric for the amount of unused disk space. To avoid duplicating information when declaring related entities, the duplicated information may be declared once in a class. For an entity to have its own copy of this information, it *inherits* the information from the class. It is possible for entities and classes to inherit from many classes, which is called multiple inheritance. Metric declarations, rules and display information can all be inherited from classes.

The way in which information is inherited depends on what is being inherited. Let e_i denote either an entity or a class and let $C_i = c_{i1} \ldots c_{in}$ denote the n classes from which e_i inherits information.

Metrics

All metrics declared in the classes in C_i are inherited by e_i unless e_i already has a metric with the same name or the inheritance of the metric is expressly inhibited.

When more than one class in C_i have metrics with the same name, the metric is inherited from the class listed first in C_i that declares the metric.

Rules

All rules declared in the classes in C_i are inherited by e_i unless the inheritance of the rule is expressly inhibited.

Display Information

If e_i has its own display information then none is inherited, otherwise the display information is inherited from the first class in C_i which has display information declared.

Class_decl :–
  'class' class_name$_{name}$ Class_body

Class_body :–
  'inherits' ':' class_name$_{name}$ { ',' class_name$_{name}$ }
  ('metrics' ':' Metric_declaration)$_{opt}$
  ('rules' ':' Rule_namelist)$_{opt}$
  $\Big($'display' ':' show_immediately (label$_{str}$)$_{opt}$ $\Big)_{opt}$
  'end'
 |
  ('inherits' ':' class_name$_{name}$ {',' class_name$_{name}$})$_{opt}$
  'metrics' ':' Metric_declaration
  'rules' ':' Rule_namelist
  $\Big($'display' ':' show_immediately (label$_{str}$)$_{opt}$ $\Big)_{opt}$
  'end'

Figure 4.1: Class declaration syntax

Inhibiting Inheritance

Even though entities may be related and inherit from the same class there may be subtle differences between them. For example, system metric names may differ slightly[1], or systems may have metrics with the same name which have slightly different meanings. Instead of declaring a new class with the different information, it may be declared in the relevant entities and the information it replaces is inhibited from being inherited.

E_i inhibits the inheritance of the rules and metrics declared in the classes of C_i that it doesn't want.

4.4 Entity and Class Declaration

This section presents a syntax for the declaration of classes and entities. Figure 4.1 shows the syntax for a class declaration. The values taken by fields are described in later sections.

The entity declaration syntax is very similar to the class declaration syntax. The differences are that, instead of a class name there is a comma separated list of entity names, the keyword `class` is replaced with the keyword `entity`, and there is a field for specifying the address of an entity. Figure 4.2 shows this syntax.

The following sections describe the fields of class and entity declarations.

4.4.1 Entity and Class Names

Names are given to entities and classes to identify them. All class names must be unique and all entity names must be unique but classes and entities may share the same names.

Many names can be given in a single entity declaration. This is a compact way of declaring many identical entities.

[1] A good example of this is the name used to identify the load average in a UNIX kernel. The most commonly used names are `_avenrun`, `avenrun`, and `sysinfo`.

Entity_decl :–
  'entity' entity_name$_{name}$ {',' entity_name$_{name}$ }
  'address' ':' entity_address$_{str}$
  Class_body

Figure 4.2: Entity declaration syntax

4.4.2 Specifying Inheritance

This field is a comma separated list of the class names from which metrics, rules and display information are inherited. The order in which class names are specified is important because this order determines which metrics are inherited when many classes declare metrics with the same name, and determines which display information is inherited when many classes have display information declared.

4.4.3 Specifying Rules

This field is a comma separated list of rule names that are used to detect and report performance problems with this entity. When a rule name is preceded by an exclamation mark (!), the rule is not inherited from the class declaring the rule. Rule declaration is described in section 4.7.

4.4.4 Display Information

This information is used to configure the CR display process. The `show_immediately` field is either `yes` or `no` and specifies whether the entity and its metrics should be displayed immediately. The `label` field is a textual label for a row of metrics displaying metric values for an entity. When this is omitted the entity name is used. If the label contains the substring `%entity`, the substring is replaced with the name of the entity.

4.4.5 Entity Address

This field specifies the address of the entity. In the context of this research, only Internet domain names are understood and which must be given in full. If the address contains the substring `%entity`, the substring is replaced with the name of the entity.

4.5 Metric Declaration

The metric declaration section, which is part of class and entity declarations, contains declarations for data associated with the entities and classes. Primarily these declarations will be of metrics, but data such as thresholds and time series can also be declared.

Figure 4.3 shows the declaration syntax for metrics.

When the inheritance of a metric is to be inhibited, the name of the metric and the class in which it is declared is preceded by an exclamation mark, as follows.

```
!  class_name.metric_name
```

Metric_decl :–
  metric_name$_{name}$ { ',' metric_name$_{name}$ }
  '{' Metric_body '}'

Metric_body :–
  ('type' ':' Type_spec)$_{opt}$
  ('starts' ':' Start_value)$_{opt}$
  ('derivation' ':' Expression)$_{opt}$
  'update' ':' Metric_update_frequency
  $\Big($'display' ':' widget$_{name}$ label$_{str}$ show_immediately (Widget_options)$_{opt}$ $\Big)_{opt}$

Figure 4.3: Metric declaration syntax

4.5.1 Metric Names

Metric names are used to identify a metric and must be unique within a class or entity. Names and descriptions of the metrics gathered by the CR collector process are given in Appendix A.2.

Many names can be given in a single metric declaration. This is a compact way of declaring many metrics, each with the same metric body.

4.5.2 Metric Type

This field is used to specify the type of the metric. At present only two types are supported, the integer type specified by the keyword `int` and the cycle type specified by the keyword `cycle`. The type declaration is optional and when it is omitted the integer type is assumed.

A `real` type is being added to the system.

Cycles

The cycle type is used to represent data such as time series, or thresholds that change periodically with different levels of resource use at different times of the day.

When a cycle is declared, further information is required. The full declaration of the cycle type is

```
cycle (duration, ticks)
```

where `duration` is a time interval that specifies the period of the cycle, and `ticks` is an integer that specifies how many values there are in each period.

Let $v_1, \ldots, v_{ticks}$ be those values. At time 0, the cycle has the value v_1. After `duration/ticks` time has elapsed the value of the cycle is v_2. After $(\texttt{duration} * (\texttt{ticks} - 1))/\texttt{ticks}$ time has elapsed from time 0, the value of the cycle is v_{ticks}. After `duration` time has elapsed from time 0 the value of the cycle is v_1 again.

The cycle is so named because the values of the cycle are repeated whenever `duration` time elapses. The value that the cycle metric starts with depends on the length of `duration` and how far through this period the current time is when the alarm manager is run. The table below shows the time used for time 0 given the length of the cycle duration.

For example, suppose a cycle has a duration of 1 hour and has 15 ticks (values), $v_1, \ldots, v_{15}$. If dog is run at 10:27, time 0 is 10:00 and the starting value of the cycle is v_7, which becomes v_8 at 10:28.

Duration	Time Zero
<= 7 days	Midnight on the previous Monday
<= 24 hours	The previous midnight
<= 60 minutes	The beginning of the previous hour
<= 60 seconds	The beginning of the previous minute

4.5.3 Metric Starting Value

The starting value field specifies the value a metric should start with. When the type of the metric is `int`, the starting value is an expression. When the type is `cycle`, the starting value is a list of numbers, one for each element of the cycle. When more numbers are given than the number of ticks in the cycle, the excess are ignored and the Bark parser produces a warning. When fewer numbers than ticks are given, zeros are used to make up the deficit.

When this field is omitted the starting value of a metric is undefined.

4.5.4 Metric Derivation

This field is optional. When the derivation is absent the metric declaration is of a cooked metric with the value of a raw metric. The metric name must be a recognised raw metric.

If the field is present it is either an expression used to derive the value of the metric, or the keyword `external`. The expressions for derived metrics are converted into C code and are the manipulations used by the manipulator.

Sometimes metrics may come from a source other than the gatherer. The logical place to store these "external" metrics is in one of the shared memory segments where it is available to all the components of Control Room. As the structure of the raw metrics shared memory segment is fixed, the Bark parser configures the manipulator so that external metrics are stored in the cooked metrics memory segment. The manipulator is also configured so that it doesn't write the values of these metrics.

4.5.5 Metric Update Frequency

This field specifies how often the metric value is updated. This field is either a time interval specifying the time that will elapse between consecutive metric updates, or the special time unit `once`.

When a metric has an update frequency of `once`, the metric value is collected or derived only once and thereafter never updated. A metric name can be given after the `once` keyword ensuring that this metric is collected or derived before the one containing the declaration. This allows the collection and derivation of metrics that are updated once to be ordered.

4.5.6 Metric Display Information

This field is used to specify how the metric should be displayed by CR. The `widget` field is one of the display meters: `dial`, `bargraph`, or `stripchart`. The `label` field is a textual label attached to the meter. If the label contains the substring `%metric`, the substring is replaced with the name of the metric. The `show_immediately` field is either `yes` or `no` specifying whether the meter should be displayed immediately, and the options enclosed in square parentheses are meter widget configuration options. Please refer to Zhang [Zha93] for more information on meters and the CR display.

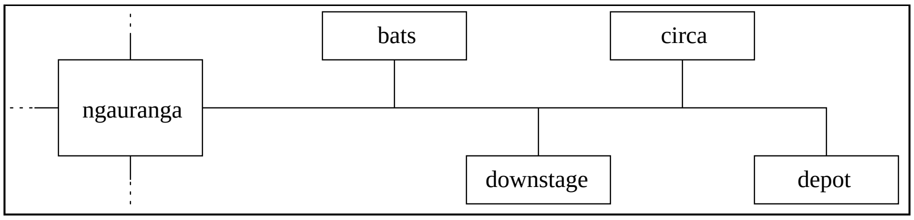

Figure 4.4: Topology of the example system

4.6 Configuration File Example

This section presents an example configuration file. The file describes how the system illustrated in figure 4.4 should be managed. The system consists of four hosts called `bats`, `circa`, `depot` and `downstage`, and a router called `ngauranga`, connected by a network. The hosts provide cpu resources for running jobs and access to the network for inter-process communication. The router provides access to other networks.

The class named `host` declares the metrics that measure the resource use of hosts. Figure 4.5 shows this declaration. The raw metrics collected from hosts are `cpu_user`, `cpu_sys`, and `cpu_idle`, which count how many times the cpu is executing user code, system code and no code respectively during the 15 second update interval; `load`, which is a measure of the number of jobs running on a host; and `avg_pkt_retransmissions`, which is a moving average of the number of time a packet needed to be sent in order to reach its destination.

The host class also has declarations for two cooked metrics, `cpu_total` and `cpu_busy`. The `cpu_total` metric is the sum of the cpu counts, and the `cpu_busy` metric is the percentage time a host is busy, defined as

$$\frac{\text{cpu\_user} + \text{cpu\_sys}}{\text{cpu\_user} + \text{cpu\_sys} + \text{cpu\_idle}} * 100\% \qquad (4.1)$$

The `cpu_busy` metric is displayed by the CR display process with a dial meter. The remaining metric declaration in the class is a constant, called `busy_threshold`, with the value 95. The rules used to detect the symptoms of performance problems are specified in the rules field. The rules are called `host_too_busy`, `load_unbalanced`, `retransmissions_too_high`, and `packets_dropped_too_high`.

Section 4.11 presents the declaration of these rules, and a discussion of the problems they address.

Figure 4.6 shows the declaration of the host entities. All hosts inherit the metrics, rules and display information from the host class, except `depot` and `downstage` which have their own declaration of `busy_threshold` that overrides the one declared in the host class.

Figure 4.7 shows the declaration of the router entity. All metric declarations are of raw metrics. The `collisions` metric is a count of the number of packet collisions detected on all networks during the one minute sampling interval. The `in_pkt` and `out_pkt` metrics are the number of packets received and forwarded respectively. Both these metrics are displayed with stripchart meters. The `packets_dropped` metric is the number of packets that were dropped because the router had too many other packets to process. Finally, the `cksum_errors` metric counts how many check sum errors were detected in packets processed by the router. The rules used to manage the router are called `collision` and `cksum`.

```
/* The host class */
CLASS host
METRICS:
    cpu_user, cpu_sys, cpu_idle {
        update :  15 seconds
    }
    cpu_total {
        derivation:  cpu_user + cpu_sys + cpu_idle
        update:  15 seconds
    }
    cpu_busy {
        derivation:  100 * (cpu_user + cpu_sys) / cpu_total
        update:  15 seconds
        display:  dial "cpu busy %" yes
    }
    busy_threshold {
        starts:  95
        derivation:  external
        update:  once
    }
    load {
        update:  1 minute
        display:  stripchart "load" yes
    }
    avg_pkt_retransmissions {
        update:  30 secs
        display:  bargraph "pkt retransmissions" yes
    }
RULES: host_too_busy, load_unbalanced,
     retransmissions_too_high, packets_dropped_too_high
DISPLAY: yes
END
```

Figure 4.5: Declaration of the `host` class

4.7 Rule Declaration

A rule consists of two parts, an expression comprised of metrics taken from entities under management and a list of actions to execute. The expression is used to describe a metric state corresponding to symptoms of a performance problem.

A rule declaration describes two rules. The first is the alarm rule, which consists of an expression that describes the symptoms of a problem, and a set of actions to execute when the alarm is triggered, i.e., when the condition is true. The second rule, called the reset rule, consists of an expression that describes the state of the metrics that show the symptoms are no longer present. Each reset rule has an optional list of actions that are executed when the reset condition is triggered.

Once the condition of an alarm rule is true, it may be true for some time. To prevent the alarm actions being repeatedly executed, the alarm condition in not evaluated again until the reset condition is true. The idea of a rule having an alarm condition and a reset condition is a generalisation of the idea that the augmented threshold alarm has a threshold and a reset threshold.

Figure 4.8 shows the syntax for the declaration of an alarm rule and its complementary

```
/*
 * The hosts under management
 */

ENTITY depot
ADDRESS: "depot.comp.vuw.ac.nz"
INHERITS: host
METRICS:
    busy_threshold {
        starts:  90
        derivation:  external
        update:  once
    }
END
```

```
ENTITY bats, circa
ADDRESS: "%entity.comp.vuw.ac.nz"
INHERITS: host
END

ENTITY downstage
ADDRESS: "downstage.comp.vuw.ac.nz"
INHERITS: host
METRICS:
    busy_threshold {
        starts:  80
        derivation:  external
        update:  once
    }
END
```

Figure 4.6: Declaration of the host entities

reset rule.

4.7.1 Rule Names

The rule name is used to refer to the rule and must be unique as rules are declared globally.

4.7.2 The Active Rule Set and the starts Keyword

The active rule set is the set of rules whose conditions are currently being evaluated. The active rule set will change with time as rules are moved into and out of it, such as when the conditions of alarm rules are satisfied and are replaced with their reset rules, and vice versa. The active rule set initially has those alarm rules in which the `starts` keyword is given.

Rules are evaluated periodically to detect symptoms of performance problems. When a condition is true, some action is taken. When a rule condition is false, the rule is rescheduled for evaluation at some future time. When many rules are scheduled for evaluation at the same time, the order of these evaluations is determined by rule priorities. Refer to section 5.2.1 for more information on rule scheduling.

4.7.3 Rule Priority

Rule priorities determine the order of condition evaluation for alarm rules whose conditions are scheduled to be evaluated at the same time. A priority is an integer between 0 and 255, with 0 being the lowest priority. If this field is omitted the rule is assigned the default priority, 127. The order of condition evaluation for rules with the same priority is arbitrary.

Reset rules that are scheduled for the same time as other rules have their conditions evaluated last, which is as though they have a priority lower than the lowest rule priority (which in fact they do). The reason for this is that a reset condition is used to determine when a symptom is no longer observable. This function is not considered as important as detecting new symptoms.

```
/* The router under management */
ENTITY ngauranga
ADDRESS: "ngauranga.comp.vuw.ac.nz"
METRICS:
    collision {
        update:  1 minute
    }
    in_pkt, out_pkt {
        update:  1 min
        display:  stripchart "%metric" yes
    }
    packets_dropped {
        update:  20 seconds
        display:  bargraph "Dropped Packets" yes
    }
    cksum_errors {
        update:  5 mins
    }
RULES: collision, cksum
END
```

Figure 4.7: Declaration of the router entity

4.7.4 Alarm Conditions

Each rule has a condition which is an expression comprising metrics from a group of entities that the rule manages. The expression is used to describe symptoms of a performance problem. Refer to section 4.2 for details on constructing an expression.

4.7.5 Actions

The action field is a list of actions that are to be executed when the alarm condition is true. Each action and its arguments are terminated by a semi-colon character (`;`). Refer to section 4.9 for descriptions of the available actions.

Under some circumstances the system administrator may not want an action to be executed. An action can be preceded by the keyword `confirm` and it will not be executed

Rule_decl :–
 ‘rule’ rule_name$_{name}$
 (‘starts’)$_{opt}$
 (‘priority’ priority$_{num}$)$_{opt}$
 ‘condition’ Expression
 ‘action’ Action_list
 ‘reset’ Expression
 (‘action’ Action_list)$_{opt}$
 ‘end’

Figure 4.8: Rule declaration syntax

without the system administrator's permission. By default, action execution doesn't require the system administrator's confirmation.

4.7.6 Reset Conditions and Actions

The `reset` field specifies the reset condition, used to describe the absence of symptoms of a problem. The reset condition is constructed in the same way as the alarm condition. If an alarm condition contains substitutions, the same substitution identifier must be used in the reset condition to refer to the same entity.

The `action` field following the reset condition is a list of actions that are executed when the reset condition is true.

4.8 Metric Operands

This section describes how metric values are specified in expressions, which are used for metric derivations and rule conditions. A metric specification consists of a metric name and the entity from which the metric is taken, which is either *explicit*, *implicit*, or given as a *substitute*.

Explicit

When a metric is to be taken from a particular entity, the entity specification is explicit. The metric specification is denoted by prefixing the metric name with the name of the entity from which the metric is to be taken and separating them with a period, as in,

```
entity_name.metric_name
```

This type of metric specification can be used in both metric derivations and rule conditions.

Implicit

It is allowed, and often necessary, to omit the name of the entity from which the metric is collected. In such cases only the metric name is given and it is the job of the parser to determine the entity from which a value should be taken. Sometimes this is not known until dog is running.

Let m_i be a metric declaration whose derivation contains an implicit metric specification, m_j. Let e_i be the entity or class in which m_i is declared. Metric m_j must either be declared in e_i or declared in a class from which e_i inherits.

When a metric is declared implicitly in a rule condition, the metric is taken, in turn, from each of the entities that the rule manages.

Involving Substitutions

When a metric is used implicitly, the name of the entity from which the metric value is taken is unknown until run time. Sometimes the argument of a rule action requires the name of the entity from which a metric value was taken. In such cases a substitute name, `%entity`, is used. The substitute name can be used by itself to refer to the name of the entity, or used to refer to a metric value, as follows,

```
%entity.metric_name
```

Some rules express symptoms of performance problems in terms of metrics from several entities, unknown until run-time. Unique substitute names can be given for each entity by using the substitute name `%entity` followed immediately by a non-negative integer. For example,

```
| %entity1.load - %entity2.load |
```

denotes the absolute value of the difference of the load from two different entities taken from the set of entities this rule manages.

Any number can be used after `%entity` with the provision that when the same number is used in different metric specifications (in the same expression) it refers to the same entity. The substitution `%entity` is equivalent to `%entity0`.

The implicit metric specification, `metric_name`, described in the previous section, is equivalent to `%entity.metric_name`.

Specification involving substitutions can only be used in rule conditions.

4.9 Builtin Actions

This section describes the actions that can be used in the action lists of rule declarations and describes how action arguments are processed.

4.9.1 Arguments

Some actions require arguments which are specified as a comma separated list of strings, entity names and metric values. These are concatenated into a single textual argument at run time before being passed to the action.

When the name of an entity is unknown the substitute name `%entity` can be used to refer to the entity's name, or if the condition has metric specifications involving substitutions, these substitutions are used to refer to the entity's name.

To include a metric value as an argument, the name of the entity from which the metric value is taken is specified along with the metric name and separated by a period, as in `entity_name.metric_name`. When the entity name is unknown the substitute name is used.

Variable substitution is illustrated in the following example. The action argument

```
%entity, "'s value of cpu_sys is ", %entity.cpu_sys
```

gets replaced with

```
"debretts's value of cpu_sys is 73"
```

assuming that the rule with this action is being used to check for performance problems on the entity called `debretts` and the value of the cpu_sys metric for debretts is 73.

Another example. Consider a rule with the condition,

```
%entity4.disk_free + %entity5.disk_free < 1000
```

Suppose this rule has an action with the arguments

```
"Warning:  the file servers ", %entity4, " and ", %entity5, "have less than
1000k between them!"
```

If the above condition was true when applied to the file servers `cancer` and `capricorn`, the argument passed to the action is

```
"Warning:  the file servers cancer and capricorn have less than 1000k
between them!"
```

4.9.2 Actions for Raising Alarms

This command is used when notification of some event is required. There are two flavours of notification.

- `notify message`
  This command prints `message` to the console.

- `alarm message`
  This command prints `message` to the console and rings the keyboard bell.

Any messages produced by the `notify` and `alarm` actions include a time stamp of the time the action was executed. For example, when the action

```
alarm "Power failure!!";
```

is executed it will produce output something like

```
14:39:26 Sun 04-Jul 1993 Power failure!!
```

4.9.3 Communicating with CR's Display

The meters used to display metric values can be manipulated by sending commands to the CR display process. Currently, commands are available for changing the colour scheme of a meter as a visual aid to indicate that the symptoms of a problem have been detected, and for hiding and showing meters.

Meter highlights

The `highlight` action changes the colour scheme of a meter from a normal colour scheme to a contrasting colour scheme, and the `unhighlight` action reverts the colour scheme back to the normal colour scheme. A meter is identified by an entity name and a metric name.

- `highlight entity_name metric_name`

- `unhighlight entity_name metric_name`

Figure 4.9 shows a meter in an unhighlighted state and figure 4.10 shows the same meter after being highlighted.

When the actions of several rules highlight a meter to indicate several problems, we do not want the colour scheme reverted with the first unhighlight action because the meter should still show outstanding problems. For this reason highlights "accumulate" on a meter. The meter is not reverted to its normal colour scheme until as many unhighlight as highlight actions are executed for the meter.

Meter visibility

Actions are available to hide or display an entire row of meters, and to hide or display an entire column of meters.

- `showrow entity_name`

- `hiderow entity_name`

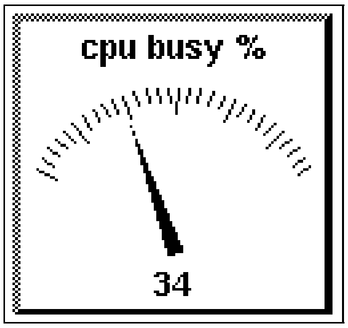

Figure 4.9: An unhighlighted meter

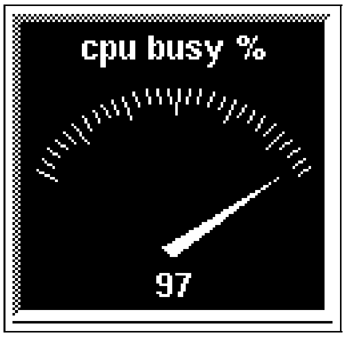

Figure 4.10: A highlighted meter

The `showrow` and `hiderow` actions show and hide the meters displaying information from the named entity.

- `showcolumn metric_name`
- `hidecolumn metric_name`

The `showcolumn` and `hidecolumn` actions show and hide the column of meters displaying the named metric.

When a meter is highlighted, the column containing the meter is automatically shown.

4.9.4 Executing Shell Commands

A mechanism is available for the execution of shell commands. This is done using the `execute` action.

- `execute command`

The `command` string contains the system command to execute and any arguments required.

For example, the action

```
execute "rxterm ", %entity, " -e top";
```

will open an xterm window, with a top process running on the machine that caused the rule condition to be true.

4.9.5 Modifying the Rule Base

Actions are available for adding rules to the active rule set, using the `enable` action, and removing rules from the active rule set, using the `disable` action.

- `enable rule_name`
- `disable rule_name`

The mechanism allowing rules to be added and removed from the active rule set is another way in which the system administrator can intelligently manage the system. For example, a rule with a condition that doesn't describe a problem accurately but is cheap to evaluate and uses few metrics can be used to enable one or more rules whose conditions need more metrics from the system and require more computation to evaluate, but can accurately determine a problem.

If the actions of a rule, r_1, enable or disable another rule, r_2, then r_2 will only be enabled or disabled for the set of entities that caused the condition of r_1 to be true. A rule can disable itself.

4.9.6 Nil Action

The `nil` action ignores any arguments and does nothing.

4.10 User Defined Procedures

User supplied procedures can be included in expressions to provide operations that cannot be expressed with the supplied operators, and can be used to create new actions. This section describes how procedures are declared and how they interact with dog.

The keyword `procedure` marks the beginning of a procedure declaration and the keyword `endprocedure` marks the end of the declaration. Only one procedure can be declared between each pair of keywords. The procedure itself is written in the C programming language. The procedure can have arguments, which are in the `argc, argv` style used in C to pass command line options to the `main()` procedure. The procedure must have the following prototype

```
return_type procedure_name(int argc, void *argv[])
```

Currently, the system only supports procedures with the integer return type, `int`.

Numbers, strings and metrics can be passed as arguments to procedures. The number of arguments in a procedure call is passed in `argc` and the arguments are passed in `argv`. The arguments are stored consecutively in `argv` from `argv[0]` to `argv[argc-1]`.

Each argument is passed to the procedure as a "generic" pointer (`void *`) to the argument. For string arguments, the pointer must be cast to `char *` to reference the string. For number arguments, whether integer or real, the pointer must be cast to a `double *` to reference the number. For metric arguments, the pointer must be cast to `struct metric_arg *`[2]. The `metric_arg` structure is defined as follows.

```
struct metric_arg {
   int substitution;
   int id;
   char *metric_name;
};
```

where `id` is a substitution identifier if `substitution` is 1 or an entity uid if `substitution` is 0. The `metric_name` field is the name of the metric.

4.10.1 Dog Library Functions

A library of functions is available for use in user supplied code. These functions are mainly for getting and setting metric values. The functions are described below.

[2]The implementation of passing metric parameters to procedures is incomplete. Currently only *implicit* metrics are passed correctly.

Name: `entity_name`
Prototype: `char *entity_name(struct metric_arg *ma);`
Description: This function returns the name of the entity from which the metric specified by `ma` is taken.

Name: `get_value`
Prototype:
```
int get_value(struct metric_arg *ma,
              struct metric_value *value);
```
Description: This function gets the value of the metric specified by `ma` and stores it in the structure pointed to by `value`. The function returns 1 if the metric value is successfully retrieved, or 0 is there is some error.

Name: `set_value`
Prototype:
```
int set_value(struct metric_arg *ma,
              struct metric_value *value);
```
Description: This function sets the value of the metric specified by `ma` to the value supplied in the structure pointed to by `value`. The function returns 1 if the value is successfully stored, or 0 if there is an error.

The metric value structure is given below

```
struct metric_value {
  int type;

  int int_val;
  // ... others as required in the future
};
```

The only type currently available is the integer type specified by setting `type` to 1. The `int_val` field holds the integer value.

4.10.2 Defining New Actions

The system administrator can define new actions with user defined procedures. If a procedure has been declared and the system administrator wants to use this procedure to define a new action, the `#action` directive is used. The directive has the following syntax

```
#action action_name procedure_name
```

where `action_name` is the name of the new action, and `procedure_name` is the name of the procedure which implements the action.

Arguments for new actions are specified and processed in the same way as for builtin actions (refer to section 4.9.1). If a new action has an argument, it is passed as a C string, that is, as a `char *`.

Redefinition of builtin actions is allowed but not advised. The Bark parser generates a warning when builtin actions are redefined.

```
RULE host_too_busy
STARTS
CONDITION
    cpu_busy > busy_threshold
ACTION
    alarm %entity,"'s cpu_busy (",%entity.cpu_busy,
         ") has exceeded the threshold of ",%entity.busy_threshold;
RESET
    cpu_busy < 65
END
```

Figure 4.11: Declaration of the `host_too_busy` rule

4.11 Configuration File Example Continued

This section presents the rules used to detect symptoms of performance problems with host entities of the system presented in section 4.6.

Assume we want to be notified when hosts exhibit the following symptoms.

1. Cpu activity is too high, indicating a high demand for the cpu.
2. The loads on hosts are significantly different, indicating that the work load on each host is unbalanced.
3. When the average packet retransmission rate is too high on any host, we want to be notified if the router is dropping too many packets.

Figure 4.11 shows the rule used to detect high cpu activity on a host. When the `cpu_busy` metric, collected from an entity, exceeds the `busy_threshold` metric collected from the same entity, the `alarm` action is executed. The entity from which the metrics are to be collected is left unspecified. This is essential so that the rule can be applied to all host entities. The value of the `busy_threshold` metrics depends on the entity from which it is taken. The hosts `bats` and `circa` use the metric value inherited from the host class, namely 95. The hosts `depot` and `downstage` have their own values which are 90 and 80 respectively.

The entity name is required for the argument of the alarm action. Because entity names are left unspecified, the substitute name `%entity` must be used. This is also used in the specification of metric value arguments.

Figure 4.12 shows the rule used to detect when the loads on hosts are unbalanced. To be precise, this rule will show when the load between *two* hosts is unbalanced, where the loads of two hosts are unbalanced if they differ by more than five.

If the entities from which the metrics are taken were left unspecified, the metrics would be taken from the *same* host which is not what we want. We cannot explicitly specify these entities without writing many rules. Instead, substitute names are used. The names `%entity1` and `%entity2` are used to refer to two different hosts, that are not known until dog is running.

These substitute names are also used in the arguments of actions. This rule also uses a user defined action called `balance_load`. The declaration of this new action is given below.

Figure 4.13 shows the rule which determines when the average number of packet retransmissions is too high. The action of this rule enables the rule called `packets_dropped_too_high` which determines when the router has dropped too many packets. Figure 4.14 shows this

```
RULE load_unbalanced
STARTS
CONDITION
    | %entity1.load - %entity2.load | > 5
ACTION
    notify %entity1," and ",%entity2," have an unbalanced load";
    balance_load %entity1," ", %entity2;
RESET
    | %entity1.load - %entity2.load | < 3
END
```

Figure 4.12: Declaration of the load_unbalanced rule

```
RULE retransmissions_too_high
STARTS
CONDITION
    avg_pkt_retransmissions > 2
ACTION
    enable "packets_dropped_too_high";
RESET
    avg_pkt_retransmissions <= 1
ACTION
    disable "packets_dropped_too_high";
END
```

Figure 4.13: Declaration of the retransmissions_too_high rule

```
RULE packets_dropped_too_high
CONDITION
    ngauranga.packets_dropped > 10
ACTION
    alarm "Router ngauranga dropped ", ngauranga.packets_dropped,
         " packets";
    highlight "ngauranga packets_dropped";
RESET
    ngauranga.packets_dropped < 5
ACTION
    unhighlight "ngauranga packets_dropped";
END
```

Figure 4.14: Declaration of the packets_dropped_too_high rule

```
#ACTION balance_load do_balance


PROCEDURE
int do_balance(int argc, void *argv[])
{
    char *host1, host2;

    if (argc != 1) {
        /* action requires an error - report error */
        return 0;
    }

    /*
     * The names of the two hosts to be balanced are given in the string
     * (char *)argv[0].  This string must be split into separate tokens
     * to get the host names.
     */

    host1 = strtok((char *)argv[0], " ");
    host2 = argv[0] + strlen(host1) + 1;

    magically_balance_hosts(host1, host2);

    return 0;
}
ENDPROCEDURE
```

Figure 4.15: Declaration of the `balance_load` action

rule. Notice that the rule doesn't have the `starts` keyword, thus it doesn't appear in the active rule set until another rule enables it. When the rule condition for the rule `packets_dropped_too_high` is true, an alarm is raised and the meter displaying the `packets_dropped` metric is highlighted.

When the original situation disappears, i.e., the average number of packet retransmissions drops below an acceptable level, the `packets_dropped_too_high` rule is disabled.

In the action section of the `load_balanced` rule is the action `balance_load`. This is a user defined action whose purpose is to balance the load of two hosts. Figure 4.15 shows the declaration of this new action and the procedure that implements it. The `#action` directive specifies that a new action called `balance_load` is being defined and is implemented by the procedure called `do_balance`. The `do_balance` procedure is given between the `procedure` and `endprocedure` keywords.

The arguments of the `balance_load` action, in the `load_unbalanced` rule, are the names of two hosts separated by a space. These arguments are concatenated at run time and passed as a C string (`char *`) to `do_balance` in argv[0]. For example, if the rule is applied to hosts named `cancer` and `capricorn` and these hosts are unbalanced, the string `"cancer capricorn"` is passed to `do_balance`. The procedure splits the string into separate host names which are given as arguments to the `magically_balance_hosts` procedure.

4.12 Preprocessing

4.12.1 Comments

The configuration file can be annotated with comments which Bark ignores. There are two flavours of comment. The first is denoted by the character sequence `//`. These characters are ignored as are all others following them until the end of the line. The second is denoted by the character pairs `/*` and `*/`. These character pairs are ignored as is all text between them including `//`. These comments may be nested, that is, comments can be placed within other comments, which can be placed within other comments and so on. Other than nesting, comments in the Bark language are the same as those in the C++ programming language.

4.12.2 File Inclusion

It is possible to include the contents of another file into the configuration file by using the `#include` directive. On the same line as the directive is the name of the file that is to be included. This name may be enclosed in quotation marks (`"filename"`) or angle brackets (`<filename>`) which indicate where the file is located. If the first form is used the file is included from the current working directory, if the second form is used the file is included from the directory that holds library files. This feature is the same as that provided in the C programming language.

4.12.3 Macro Substitution

A macro facility is provided whereby the system administrator writes some text, gives it a name and whenever this name is encountered in the configuration file it is replaced with the text. A macro is declared with either of the directives `#replace` or `#define`. Each directive takes two arguments `name` and `replacement text`.

The macro facility provided in Bark is exactly the same as that provided in the C programming language, except that `#replace` may be used in place of `#define`. In fact, the Bark implementation is derived from the C implementation. For a complete description of the macro facility, refer to any good C book such as *The C Programming Language* by Kernighan and Ritchie [KR88].

Chapter 5

The Alarm Manager

The alarm manager, called Dog, is a prototype of the component in control room that detects and reports performance problems. Dog has been designed to provide a framework in which alarm functions, such as those described in chapter 2, can be implemented and evaluated. This framework builds on the service provided by the manipulator.

Dog uses rules, written by the system administrative staff in the Bark language, to detect and report performance problems.

This chapter discusses the implementation of the alarm manager.

5.1 Alarm Manager State

The state used by dog to evaluate rule conditions is distributed throughout the control room system. Metric values used in rule condition evaluation are stored in the cooked metrics shared memory segment. Rules, entities, and other state information are maintained by dog.

Figure 5.1 shows the information stored for entities and metrics. A record for each entity is stored in a table of all entity records and contains fields including the metrics associated the entity, and the rules used to manage the entity.

Each entity has a variable number of metrics, stored in a list associated with the entity. Each metric record includes a reference to a metric body, a reference into the cooked metrics memory segment where the value of the metric is stored, and a cache for the value so that metric values are used consistently in conditions and actions. Each entity also has a variable number of rules used to manage the entity, stored in a list associated with the entity. The list contains references to rule records stored in a rule table.

Each metric body record is stored in a table containing all metric bodies and includes the metric type and update frequency.

Figure 5.2 shows the information stored for rules and actions. A record for each rule is stored in the rule table and includes a reference to its reset rule, the condition to evaluate, and the actions to execute when the condition is true. The actions to execute are stored in a list associated with each rule. Each action record includes the arguments of the action and a reference into the action table, which holds the implementations of the actions. All user defined actions are stored in the action table.

5.1.1 Rule Instances

When a rule condition is declared, it will in general be expressed in terms of metrics without explicit reference to the entities from which the metric values should be taken. Dog builds structures called *rule instances* which have a rule identifier and a list of entities whose metric values are used to evaluate the rule condition. A rule instance is represented as a two-tuple

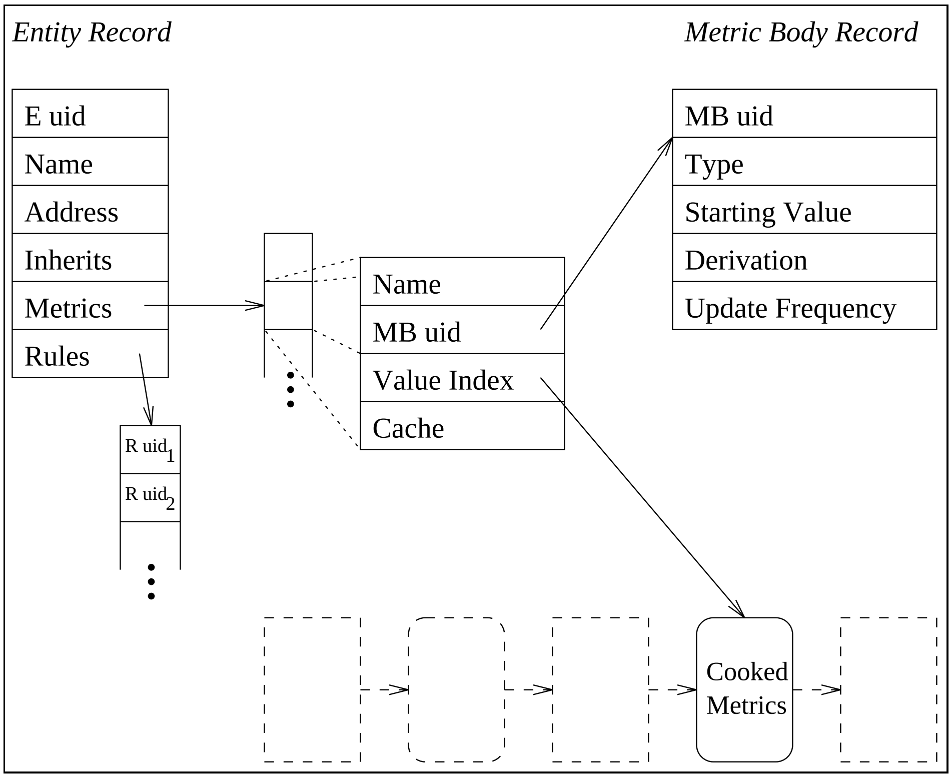

Figure 5.1: Entity and metric state

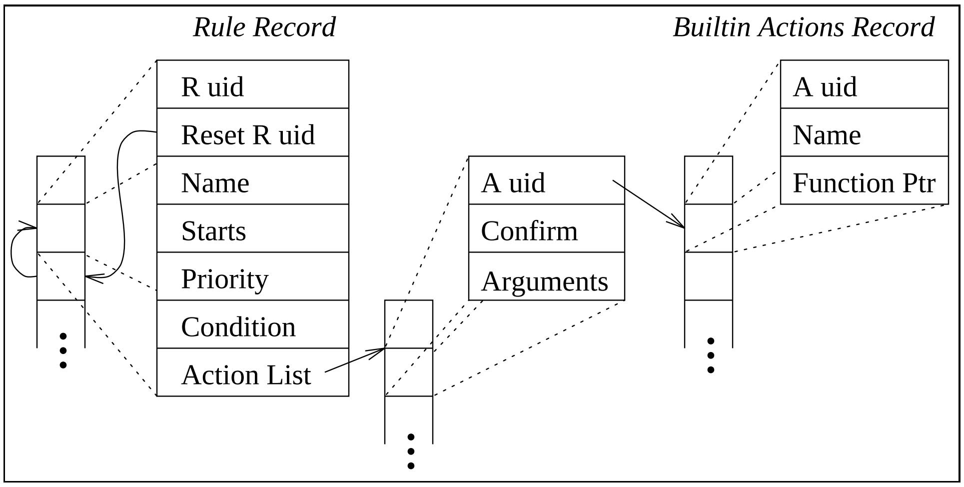

Figure 5.2: Rule and action state

<`mapping : rule id`> where the `rule id` is the uid of a rule and `mapping` is a function from substitution identifiers to entity uids. The notation $s_1, \ldots, s_p \rightarrow e_1, \ldots, e_p$, used in the examples that follow, represents a mapping where substitution identifier s_i maps to entity uid e_i.

Rule Conditions Without Substitutions

When the metrics in a condition have no substitutions, they are taken from the same entity. The parser writes these metrics as though they were taken from an entity with the substitute name `%entity0`. A rule instance is produced for each entity which has the rule name in its rules field with a mapping from substitution identifier 0 to the entity uid.

For example, suppose a rule with uid 12 has the condition

```
cpu_sys + cpu_nice + cpu_user > 90
```

and is used to manage three entities with identifiers 4, 9, and 11. The following three rule instances are created.

$$
\begin{aligned}
&<0 \rightarrow 4 \;\; : 12> \\
&<0 \rightarrow 9 \;\; : 12> \\
&<0 \rightarrow 11 : 12>
\end{aligned}
$$

Rule Conditions With Substitutions

Rule instances are created for rule conditions with substitutions in the following way.

Let n be the number of unique substitutions in the rule condition, and let $\bar{s} = s_1, \ldots, s_n$ be those unique substitution identifiers. Let p be the number of entities that have the rule name in their rules field, and let $\bar{e} = e_1, \ldots, e_p$ be the uids of those entities. There are ${}^nC_p$ rule instances generated, with mappings from $\bar{s}$ to each combination of $\bar{e}$ of size p.

For example, suppose a rule with uid 18 has the condition

```
| %entity1.load - %entity2.load | > 5
```

and is used to manage four entities with identifiers 3, 6, 8, and 13. The following six rule instances are created.

$$
\begin{aligned}
&<1,\, 2 \rightarrow 3,\, 6 \;\; : 18> \\
&<1,\, 2 \rightarrow 3,\, 8 \;\; : 18> \\
&<1,\, 2 \rightarrow 3,\, 13 : 18> \\
&<1,\, 2 \rightarrow 6,\, 8 \;\; : 18> \\
&<1,\, 2 \rightarrow 6,\, 13 : 18> \\
&<1,\, 2 \rightarrow 8,\, 13 : 18>
\end{aligned}
$$

Combinations of entity uids are used instead of permutations because for operators such as addition, multiplication and absolute difference ($| a - b |$), the order of the arguments (metric values) is unimportant. I have yet to find a rule condition where combinations of entity uids is insufficient, nevertheless, such a condition may exist, and if it is found the system can be easily extended to generate permutations of uids.

5.1.2 Expression Representation

Expressions, used for rule conditions, are represented as lists of operators and operands stored in reverse polish notation. A stack is used in the evaluation process. The stack contains operands and the results of partially evaluated expressions that result from operators being applied to operands from the stack. After the whole expression has been processed the stack contains the result of the expression.

An operand consists of a type and a value, which can be a number, a metric or the result of a call to a user supplied procedure.

Metric Operands

A metric operand is represented as a three-tuple <`flag, uid, metric name`>, where `flag` specifies whether `uid` is an entity uid or a substitution identifier, and `metric name` is the name of the metric.

A metric value is obtained in the following way. If uid is a substitution identifier, the corresponding entity uid is obtained from the mapping in the rule instance currently being processed. The entity uid is used to get an entity record from the entity table. The metric name is used to locate the metric record from the metrics stored with the entity record. The metric record contains the index of the value in shared memory.

Procedure Operands

A procedure call operand is represented as a three-tuple <`uid, argc, argv`>, where `uid` is the index of the procedure in the array of function pointers constructed by the Bark parser, `argc` is the number of arguments in the procedure call and `argv` stores the arguments.

5.2 Implementation of the Alarm Manager

This section discusses rule condition evaluation and scheduling of active rule instances. To evaluate rule conditions, dog uses the metric values stored in the cooked metrics shared memory segment. Access to the memory segment is unrestricted as neither the manipulator nor dog write to any of the same locations in shared memory. Rule condition evaluation needs to be efficient. The problem to address is when should dog read metrics from the shared memory segment?

5.2.1 Rule Condition Evaluation and Scheduling

The condition of each active rule instance, whether it is an alarm rule or a reset rule must be evaluated periodically to detect the presence or absence of symptoms of performance problems. Whenever a rule condition is false, it must be re-evaluated at some time in the future.

When the rule condition is an expression with a single metric, the rule condition should be re-evaluated whenever the value for the metric is updated. Ideally, the manipulator would send a signal to dog each time the metric value was updated. However, this reduces the degree of independence of the manipulator, which is undesirable as the plug and play potential of the tool is reduced. Instead, dog "guesses" when the metric has been updated and polls the value of the metric. After a metric value has been polled the first time, dog assumes that the metric will next be updated after the time given by the metric's update frequency. Thus for rule conditions with just one metric, the rule condition is evaluated at the period given by the metric's update frequency.

When the rule condition is an expression of several metrics, the time that the rule condition should be re-evaluated depends on the operators used in the condition. For example, consider the expression

```
cpu_busy <= 20 && ioq_length >= 50
```

where `cpu_busy` is derived every 10 seconds, and `ioq_length` is collected every 30 seconds.

If the condition is evaluated and is false because the value of `cpu_busy` is greater than twenty *and* the value of `ioq_length` is greater than or equal to fifty, it is scheduled to be re-evaluated in ten seconds, when the value of `cpu_busy` is recalculated. On the other hand, if the expression is false because the value of `ioq_length` is less than fifty regardless of the value of `cpu_busy`, the expression does not need to be re-evaluated until the value of `ioq_length` is collected, in thirty seconds.

The code to determine when a rule condition should be re-evaluated is integrated with the code to evaluate the rule condition. Remember that conditions are stored in reverse polish notation and the evaluation process uses a stack. Each item on the stack contains, as well as the value of a subexpression, the time to wait before the subexpression is re-evaluated. Figure 5.3 shows this algorithm.

Because dog guesses when a metric value is updated, it can poll a metric value twice before the metric value is updated, or miss a metric value completely. The same metric value will be polled when the manipulator is delayed, and will be missed when dog is delayed. The number of misses or re-reads depends on the update frequency of the metric and the amount of delay.

Under normal loads we would expect dog to miss or re-read at most one metric value. Dog cannot tell if the manipulator has been delayed, so doesn't worry about re-reading metric values. Also, dog cannot differentiate between the manipulator being delayed and a metric value being updated with the same value.

Dog doesn't worry about missing values either. If dog misses a value that would have indicated a performance problem, the next value may still indicate the problem, but if it doesn't, the problem was short lived.

5.2.2 The Active Rule Set

The active rule set stores rule instances whose conditions are currently being evaluated. When dog first runs, the active rule set contains rule instances for rules with the `starts` flag. Rule instances are added and removed from the active rule set by enable and disable actions, and when an alarm rule instance is replaced with its reset rule instance when its condition is true, and vice versa.

The set is implemented with a priority queue in which two priorities are used to order queue elements. The primary priority is the time the rule condition is scheduled to be evaluated, and the secondary priority is the rule priority, used to order rule instances whose conditions are scheduled for evaluation at the same time.

Figure 5.4 shows the pseudo code for the dog's rule processing loop, written in a C++ style. The rule instance and rule set methods are self-explanatory, except for the `update` method for rule instances. This method returns the time that should elapse before the condition is evaluated, assuming we know nothing about the state of the operands. This time is the minimum update frequency of the operands in the rule's condition, where the update frequency of a number is infinity, the update frequency of a metric is the metric's update frequency, and the update frequency of a procedure is infinity[1].

[1] See the footnote in figure 5.3.

Let $\bar{e} = e_1, \ldots, e_n$ be the operators and operands of an expression stored in reverse polish notation. Let (v, t) denote a node stored on the evaluation stack consisting of value v and update time t.

For each e_i from e_1 to e_n do

- if e_i is a number
  push (e_i, ∞)

- if e_i is a metric
  push (value(e_i), update_frequency(e_i))

- if e_i is a procedure call
  push (the result of the e_i, ∞^a)

- if e_i is a unary operator
  pop (v, t)
  push $(e_i(v), t)$

- if e_i is any binary operator other than AND
  pop (v_1, t_1)
  pop (v_2, t_2)
  push ($e_i(v_2, v_1)$, $\min(t_1, t_2)$)

- if e_i is the binary AND operator
  pop (v_1, t_1)
  pop (v_2, t_2)

  if $t_1 > t_2$
      swap (v_1, t_1) with (v_2, t_2)

  if v_2 is FALSE
      push (FALSE, t_2)
  else
      push (v_1, t_1)

[a] This is a known "bug" with dog. The update for a procedure should be the update of the argument with the smallest update frequency. This problem can be worked around. If the argument of a procedure with the smallest update frequency is a metric, m_i, construct a new condition as follows: $0 * m_i + \mathrm{proc}(\ldots, m_i, \ldots)$

Figure 5.3: Expression evaluation and scheduling

```
// set up the initial active rule (instance) set

create_rule_instances();
the_time = current_time();
FOR EACH rule instance r_i
    r_i.execution_time() = the_time + r_i.update();
    rule_set.add(r_i);

// process the rule instances

r = rule_set.next_rule_instance();
DO FOREVER
    the_time = current_time();
    sleep_period = r.execution_time() - the_time;
    IF sleep_period >= 0
        sleep_period.sleep();
    ELSE
        // panic, we are behind with our processing
    ENDIF

    // Set the time variable to the time we wanted to wake up instead
    // of the time we actually woke up. This is done so that rules
    // are synchronised.
    the_time = r.execution_time();

    DO    // process rule instances scheduled for the_time
        (v, t) = r.evaluate();
        IF v == FALSE
            r.execution_time() = the_time + t;
        ELSE
            r.execute_actions();
            r.uid() = r.reset_uid();
            r.execution_time() = the_time + r.update();
        ENDIF

        rule_set.add(r);
        r = rule_set.next_rule_instance();
    WHILE r.execution_time() == the_time
```

Figure 5.4: Rule instance processing

The basic function of the alarm manager is to sleep until a metric has changed value, then evaluate the conditions of all rules that depend on that metric. The only problem that can occur is dog gets behind in its processing, indicated by a negative sleep period. This can happen for two reasons. Firstly, the host on which dog runs could be heavily loaded, delaying all processes. Secondly, dog may get behind because it processes a lot of rules between sleeps. In the current implementation, whenever dog observes it is behind, it prints a message and continues processing as though it wasn't behind.

As the implementation of the alarm manager can be improved, and no real experience has been gained with it because it hasn't been used to manage the performance of a distributed system, no improved strategies for dealing with being behind with rule processing have been implemented.

Chapter 6

Conclusions

Part of my work was to explore and evaluate different methods for detecting performance problems. The other part of my work was to design and implement a framework for detecting performance problems in distributed systems. The framework was designed to be flexible so that a wide variety of methods for problem detection, such as those discussed in chapter 2, could be used. The alarm manager, dog, and the management language, bark, are the result of this work. Bark is discussed in chapter 4 and dog is discussed in chapter 5.

This thesis contributes the following.

- Various methods for detecting performance problems are presented and explored in the context of computer systems. These methods include ideas from statistical process control and time series modelling.
- The alarm manager and the management language are presented, which are part of the ongoing control room project whose eventual goal is to regulate the use of a distributed system.

  These components provide the following features.

  - Performance problems can be expressed in terms of arbitrary expressions, as opposed to just threshold alarms. The manipulator process plays an important role by producing cooked metrics.
  - Performance problems can also be expressed in terms of metrics from a group of entities as opposed to just a single entity.
  - Rule enabling and disabling gives a system administrator flexibility in rule construction. Rules can be constructed intelligently, allowing computationally inexpensive rules, that may not accurately express symptoms of a problem, to enable expensive but accurate rules. Rules that detect symptoms of performance problems in one part of the system can enable rules that look for problems in other parts of the system.

Dog and bark are different from the commercial tools used to manage computer systems. Technical information on commercial tools is scarce. However, with most commercial tools the user interacts with a graphical display that provides a view of the system and system activity. The problem detection mechanisms provided by these tools, if any, usually consist of a threshold alarm and notification when a host is not responding. It appears that users are encouraged to enter performance problems through a point and click interface rather than write expressions for system performance problems. Dog is centered around the Bark management language which allows problems to be described by arbitrary expressions of metrics from many hosts.

Bark and dog have not yet been used to detect performance problems in a distributed system so no conclusions can be made as to how good the framework is for the exploration of performance problems. It is however possible to implement the methods for detecting performance problems described in chapter 2 with bark and dog. Figures 6.1 and 6.2 show the implementation of the cusum decision interval test for a metric called `cpu_busy`.

6.1 Future Work

There is plenty of scope for other work with this project, and with the CR project in general. The most important work that needs to be done with respect to dog and bark is to use them to manage the performance of a distributed system to gain experience with them and the methods for detecting performance problems.

To effectively manage a system requires knowledge of the system so that symptoms of performance problems can be identified. In some cases problems may relate to the configuration of a system, defining who can use what, and in other cases performance problems will relate to the design of the system components. For example, MVS is a batch system designed to maximise job through-put, while UNIX is a time-sharing system designed to give performance to interactive jobs [LMKQ89]. MVS may not give good performance to interactive jobs and UNIX may not give good performance to batch jobs.

6.1.1 UNIX Performance Metrics

This section discusses two metrics found in the UNIX operating system, the cpu state counters and the load average. The properties of these metrics are discussed in relation to the design of UNIX.

Cpu State Counters

The cpu states, `cpu_sys`, `cpu_user`, `cpu_nice`, and `cpu_idle`, are counters that record the number of times the cpu was in each state. These are not good indicators of performance because of the job scheduling policy.

A job is initially given a high priority and allowed to run for a period of time. If the job doesn't relinquish the processor during this time, its priority is lowered. If the job does relinquish the processor, to do I/O say, its priority is unchanged. The cpu states on a host executing a cpu bound job will show the host is 100% busy. However, this job will be preempted by other jobs of higher priority, including interactive jobs. Users running interactive jobs, such as news readers and text editors, will not notice the load placed on the machine by cpu bound jobs, such as compile jobs and manuscript typesetting programs. Thus a high cpu busy is not a reliable symptom of a performance problem.

"Load Average"

An interesting metric to consider, because of it's importance, is the load average. This metric is used in the calculation of process priority and to determine if processes should be swapped. The program for sending mail within the Internet, `sendmail`, shuts down when the load on the machine it runs on goes above nine so that it places no further load on the machine.

Three load averages are maintained by the UNIX kernel, a 1-minute, 5-minute and 15-minute average. The 1-minute load average is calculated using the following formula,

```
CLASS cpu
METRICS:
    cpu_busy {
        update:  20 seconds
    }
/*
 * constants
 */
    h { // the decision interval
        starts:  99
        derivation:  external
        update:  once
    }
    k { // half the reject quality level
        starts:  85
        derivation:  external
        update:  once
    }
/*
 * variables maintained by the cusum procedure
 */
    cusum_score { // the cusum score
        derivation:  external
        update:  once
    }
    reset { // used to reset the cusum score
        starts:  1
        derivation:  external
        update:  once
    }
RULES: decision_interval
END

RULE decision_interval
STARTS
CONDITION
    cusum(cusum_score, cpu_busy, k, reset) > h
ACTION
    alarm %entity, "'s cpu is busy...";
RESET
    cusum(cusum_score, cpu_busy, k, reset) < (h/2)
END
```

Figure 6.1: An implementation of the decision interval

```
PROCEDURE
int cusum(int argc, void *argv[])
{
    struct metric_arg *cusum_score, *metric, *k, *reset;
    struct metric_value cusum_val, m_val, k_val, r_val;

    if (argc != 4)
        /* error */;

    get_value(cusum_score, &cusum_val);
    get_value(metric, &m_val);
    get_value(k, &k_val);
    get_value(reset, &r_val);

    if (r_val.int_val == 1) {
        r_val.int_val = 0;
        cusum_val.int_val = m_val.int_val;
    }
    else {
        cusum_val.int_val += (m_val.int_val - k_val.int_val);
        if (cusum_val.int_val < 0)
            r_val.int_val = 1;
    }

    set_value(reset, &r_val);
    set_value(cusum_score, &cusum_val);

    return cusum_val.int_val;
}
ENDPROCEDURE
```

Figure 6.2: An implementation of the decision interval (cont.)

```
void main(void)
{
   for(;;);
}
```

Figure 6.3: A cpu hogger

$$load_t = e^{-\frac{1}{12}}\, load_{t-1} + (1 - e^{-\frac{1}{12}})\, n_t \tag{6.1}$$

where n_t is the length of the run queue at time t.

The five and fifteen minute load averages use the same formula except that $e^{-\frac{1}{60}}$ and $e^{-\frac{1}{180}}$ and used respectively, instead of $e^{-\frac{1}{12}}$ in the above equation.

The processes that are counted in the run queue are as follows. Processes that have the resources to run, processes that don't have the resources to run because they have been created recently, and processes that are waiting for a resource and should not be interrupted because the resource requires atomic access are counted.

The load average is an exponentially decayed function of the run queue length. Why exponential smoothing? Probably because it is cheap in terms of computation and memory requirements. Why were those choices of e used? Note that the powers of e have the same ratios as the duration of the load average.

Each average is updated every 5 seconds, in which time there are approximately fifty time slices[1] in which tasks can run. What are the designers of the load average trying to see given such a large update period? What "typically" happens in the system during this time?

What sort of properties does this function have, and how good is the load a measure of system performance?

It has the property that $load_t$ is constant for constant n_t, and in fact equals n_t, irrespective of the type of jobs running. The load average doesn't differentiate between interactive and cpu bound jobs — but then it shouldn't, it is an average run queue length. Yet a metric that measures load should take into account the types of jobs that are running. For this reason I think that "load average" is an unfortunate name, "average run queue length" would be better.

The load being high may not be a symptom of a performance problem when the load is caused by a lot of low priority jobs. Interactive users will be relatively unaffected by these jobs. However, other cpu bound jobs will be affected.

As a little experiment I ran twenty copies of the program illustrated in figure 6.3.

The load very quickly rose above twenty. I was able to edit files and read news without noticeable delay, though a compile job that normally took one minute, took ten. This is of course what we would expect given the way cpu is scheduled in the UNIX operating system. The interactive jobs have higher priority than the hoggers. The compile job, after having read the source files, is cpu bound while it compiles the files, and ends up competing with the twenty other cpu bound jobs for the cpu.

6.1.2 Alternative Metrics

The following three metrics could be useful performance measures of a UNIX system.

[1] BSD UNIX version 4.3

- A metric that measures how much cpu time is spent running system tasks (such as processes 0-2, the `swapper`, `init`, and the `pagedaemon`) and how much time is spent running user tasks. Obviously system tasks need to run, but if they run for too long, users will notice a degradation in performance as user tasks aren't running.
- A process priority reflects how much time it has spent at the processor recently. Another "load average" might be based on the weighted sum of process priorities. For example, ten processes that have just finished doing I/O represent more load on the machine than ten cpu bound processes.
- A metric that measures the amount of time the cpu spends executing jobs with priorities higher than the default priority. As this time increases, users may notice performance problems are more high priority jobs are competing for the processor.

Exploring how jobs and job mixes perform when other resources such as memory and disk are scarce would also be useful in finding symptoms of performance problems.

6.1.3 Other work

As stated earlier, experience with this tool is required before it can be evaluated as a performance manager. However, some experience has been gained through testing and writing about the system. The following work is needed.

- Rule enabling and disabling will need to be extended. For example, an enabling mechanism should be capable of enabling a set of rules for particular entities, instead of just the entities that enabled the rule. Also, when a rule is disabled, it may be necessary to execute the reset actions of the rule if its alarm condition was true.
- Another area to look at, related to rule scheduling, is the idea of having a mechanism to turn on and off the collection of metrics from entities, depending on the rules being used to evaluate the performance of entities. This is a useful mechanism in that it reduces the load on the network by only collecting what is required. However, it reduces the independence of the processes in Control Room.
- A survey can be conducted to determine the sorts of problems that system administrators want notification of. I asked our system administrator what notification he would want from the alarm manager. He wanted notification when
  - certain processes weren't running,
  - static tables, such as the process and text table, became full,
  - the load became too high,
  - the disks were almost full, and
  - communication links failed.

  Many of these can be built from ad hoc gathers using shell scripts and the like to monitor parts of the system.

Now that the display, alarm and data gathering components are functional and integrated, the tool should be used to explore automated performance management of a distributed system.

Appendix A

User Guide

A.1 The Bark Language Grammar

This chapter presents the Bark language grammar. The syntax of the grammar consists of a set of production rules denoted by `non-terminal` :– `body` where `non-terminal` is the name of the production rule and `body` is what is produced.

The body of a production rule can contain literals, names, numbers, strings and non-terminals. Production rule bodies also have notation to represent repeated symbols, optional symbols and the choice of one from a number of symbols. The following table summarises this notation.

Notation	Meaning
x :– y	an x produces a y
{x}	x occurs 0 or more times
$(\mathrm{x})_{opt}$	x is optional
x | y	x or y
'x'	x is a keyword or a literal
x_{str}	x is a string
x_{num}	x is a number
x_{name}	x is an identifier

The production rule named Configuration_file starts the grammar.

A.1.1 The Configuration File

Configuration_File :–
  Declaration {**Declaration**}

Declaration :–
  Class_decl
  | **Entity_decl**
  | **Filename_specification**
  | **NewAction_decl**
  | **Procedure_decl**
  | **Rule_decl**

A.1.2 Classes and Entities

Class_decl :–
  'class' class_name_{name} **Class_body**

Class_body :–
  'inherits' ':' class_name_{name} {',' class_name_{name}}
  ('metrics' ':' **Metric_declaration**)$_{opt}$
  ('rules' ':' **Rule_namelist**)$_{opt}$
  $\Big($'display' ':' show_immediately[1] (label_{str})$_{opt}$ $\Big)_{opt}$
  'end'
  |
  ('inherits' ':' class_name_{name} {',' class_name_{name}})$_{opt}$
  'metrics' ':' **Metric_declaration**

'rules' ':' **Rule_namelist**
('display' ':' show_immediately[1] (label_{str})$_{opt}$)$_{opt}$
'end'

Rule_namelist :–
('!')$_{opt}$ rule_name_{name} { ',' ('!')$_{opt}$ rule_name_{name} }

Entity_decl :–
'entity' $\text{entity\_name}_{name}$ {',' $\text{entity\_name}_{name}$ }
'address' ':' $\text{entity\_address}_{str}$
Class_body

A.1.3 Metrics

Metric_declaration :–
Metric_decl { **Metric_decl** }

Metric_decl :–
Positive_metric_decl
| **Negative_metric_decl**

Positive_metric_decl :–
$\text{metric\_name}_{name}$ { ',' $\text{metric\_name}_{name}$ } '{'
('type' ':' **Type_spec**)$_{opt}$
('starts' ':' **Start_value**)$_{opt}$
('derivation' ':' **Derivation**)$_{opt}$
'update' ':' **Metric_update_frequency**
('display' ':' widget_{name} label_{str} show_immediately[2] (**Widget_options**)$_{opt}$)$_{opt}$
'}'

Negative_metric_decl :–
'!' class_name_{name} '.' $\text{metric\_name}_{name}$

Start_value :–
Expression
| '{' number_{num} { ',' number_{num} } '}'

Type_spec :–
$\text{Type\_identifier}_{name}$
| 'cycle' '(' Time_period ',' Time_period ')'

Derivation :–
'external'
| **Expression**

Metric_update_frequency :–
Time_period
| 'once' ($\text{metric\_name}_{name}$)$_{opt}$

[1] Show_immediately is specified by either keyword 'yes' or 'no'
[2] Show_immediately is specified by either keyword 'yes' or 'no'

Time_period :–

number$_{num}$ time_unit[3]

Widget_options :–

'[' ('–')$_{opt}$ **Option** $\Big\{$ ('–')$_{opt}$ **Option** $\Big\}$ ']'

Option :–

number$_{num}$

| identifier$_{name}$

| string$_{str}$

A.1.4 Rules

Rule_decl :–

'rule' rule_name$_{name}$

('starts')$_{opt}$

('priority' priority$_{num}$)$_{opt}$

'condition' **Expression**

'action' **Action_list**

'reset' **Expression**

('action' **Action_list**)$_{opt}$

'end'

Action_list :–

Action_decl ';' { **Action_decl** ';' }

Action_decl :–

('confirm')$_{opt}$ action_name$_{name}$ (**Action_argument**)$_{opt}$

Action_argument :–

string$_{str}$

| identifier$_{name}$

| entity_name$_{name}$ '.' metric_name$_{name}$

A.1.5 Expressions

The grammar for expressions is ambiguous. Ambiguity is resolved by operator associativity and precedence. The table given below shows the operators and their associativity. Precedence increases down the table.

Expression :–

number$_{num}$

| metric_name$_{name}$

| entity_name$_{name}$ '.' metric_name$_{name}$

| procedure_name$_{name}$ '(' (**Argument** {',' **Argument**})$_{opt}$ ')'

| '(' **Expression** ')'

| **Expression** '+' **Expression**

| **Expression** '–' **Expression**

[3]The time unit is specified by one of the following keywords: 'millisecond', 'second', 'minute', 'hour' and 'day', which may be abbreviated respectively to 'msec', 'sec', 'min', 'hr' and 'dy'.

<table>
<tr><th>Operator</th><th>Associativity</th></tr>
<tr><td>|| |</td><td>left</td></tr>
<tr><td>&& &</td><td>left</td></tr>
<tr><td>< <= == != >= ></td><td>non-associative</td></tr>
<tr><td>+ −</td><td>left</td></tr>
<tr><td>− (unary)</td><td>right</td></tr>
<tr><td>* /</td><td>left</td></tr>
<tr><td>!</td><td>right</td></tr>
</table>

| **Expression** ‘*’ **Expression**
| **Expression** ‘/’ **Expression**
| **Expression** ‘&’ **Expression**
| **Expression** ‘|’ **Expression**
| ‘|’ **Expression** ‘|’ ....absolute value
| ‘−’ **Expression**
| ‘!’ **Expression**
| **Expression** ‘&&’ **Expression**
| **Expression** ‘||’ **Expression**
| **Expression** ‘<’ **Expression**
| **Expression** ‘<=’ **Expression**
| **Expression** ‘==’ **Expression**
| **Expression** ‘!=’ **Expression**
| **Expression** ‘>=’ **Expression**
| **Expression** ‘>’ **Expression**

Argument :–
string$_{str}$
| identifier$_{name}$
| number$_{num}$
| entity_name$_{name}$ ‘.’ metric_name$_{name}$

A.1.6 Procedure and New Action Declarations

Procedure_decl :–
‘procedure’ c-function ‘endprocedure’

NewAction_decl :–
‘#action’ action_name$_{name}$ procedure_name$_{name}$

A.1.7 Action Summary

The table given below provides a summary of the built in actions.

A.1.8 Configuration Output Specification

Filename_specification :–
ConfigFile_specifier filename$_{str}$

ConfigFile_specifier :–
‘config_file’ | ‘action_file’ | ‘gather_file’ | ‘manip_file’ | ‘display_file’
| ‘manip_names’
| ‘dog_path’ | ‘cr_path’

A.2 The Raw Metrics

The gatherer that uses `rstat` to collect performance information from hosts provides a subset of them for use in CR. All metrics are running totals and are sampled at a fixed interval which is by default two seconds. This interval can be changed with a command line option on the gatherer. The gatherer deposits the *difference* between sampled counter values in the raw metrics shared memory segment. The available metrics are described below.

intr The number of device interrupts

page_in The number of memory pages paged in

page_out The number of memory pages paged out

cpu_user The number of times the cpu is in user state

cpu_nice The number of times the cpu is in nice state

cpu_sys The number of times the cpu is in system state

cpu_idle The number of times the cpu is in idle state

pg_swap_out The number of memory pages swapped out

pg_swap_in The number of memory pages swapped in

in_pkt The number of packets received on all network interfaces

out_pkt The number of packets sent on all network interfaces

in_berr The number of input errors on all network interfaces

out_berr The number of output errors on all network interfaces

collision The number of collisions on the CSMA interfaces

disk0 The number of transfers made by disk0

disk1 The number of transfers made by disk1

disk2 The number of transfers made by disk2

disk3 The number of transfers made by disk3

A.3 Bark Manual Page

NAME

bark - Bark language parser

SYNOPSIS

bark <u>filename</u>

DESCRIPTION

The Bark parser is part of the Control Room (CR) Distributed System Manager. The parser translates <u>filename</u>, a file written in the Bark language into a new set of configuration files for the gather, manipulator, display and alarm manager components of CR.

DIAGNOSTICS

Unfortunately diagnostics don't really exist. When a parse error is detected the parser aborts with the message "parse error" and the line number where the error was detected. Other errors are detected when the new configuration files are created. These messages are verbose and a few are cryptic but they describe the error and its location.

FILES

The following files list respectively, the actions built into dog, and the types supported by dog, and are produced by the dog(8C) -install option.

/a/projects/cr/KENNEL/dog/lib/builtin_actions
/a/projects/cr/KENNEL/dog/lib/builtin_types

SEE ALSO

dog(8C)

BUGS

When a metric is specified with an explicit reference to the entity from which the metric value is taken, no checking is done to see if the metric exists. Dog will dump core if it encounters metrics undefined in this way.

The implementation of passing metric parameters to procedures is incomplete. Currently only implicit metrics are passed correctly.

The confirm keyword for actions is present but not supported yet.

The cycle type is present in the language but not supported yet.

A.4 Dog Manual Page

NAME

dog - The alarm manager for Control Room

SYNOPSIS

dog [-log logname] [-nolog] [-alertport portnumber] -config configfilename -action actionfilename

dog -install

DESCRIPTION

Dog is part of the Control Room (CR) Distributed System Manager. Dog is given a set of rules which comprise a condition for detecting symptoms of performance problems and a set of actions to execute when the condition is true. Rules are provided in configfilename and the actions are given in actionfilename. Both these files are produced by the bark(8C) parser.

Dog will only run when the gatherer, manipulator and display processes of the CR system are running. All processes must be run and on the same host under SunOS as they communicate through shared memory segments.

Problems are indicted in two ways. The first is by a textual message written to the standard output, that results from an alarm or notify action. The terminal bell is also sounded for the alarm action. The second is by changing the colour scheme of display meters, that results from a highlight action.

OPTIONS

Options may appear in any order.

-log logname

The default log used for alarm and notify actions is called dog.log. The log is named logname when this option is given.

-nolog

No log is produced when this option is given, even when the -log option is given.

-alertport portnumber

Dog and the display communicate over TCP socket 8008. Should this socket be unavailable socket portnumber will be used instead when this option is given. The display also has a -alertport option. The interface between dog and the display is clumsy and may disappear from user view in the future.

-config configfilename

This option specifies the name of the configuration file, produced by bark(8C), that contains all configuration information except action information.

-action actionfilename

This option specifies that name of the configuration file, produced by bark(8C), that contains the actions to execute.

-install
: This option is used whenever a new version of dog is installed. The builtin action and builtin type tables are written to the bark library directory.

FILES

The following files list respectively, the actions built into dog, and the types supported by dog, and are produced by the dog(8C) -install option.

/a/projects/cr/KENNEL/dog/lib/builtin_actions
/a/projects/cr/KENNEL/dog/lib/builtin_types

SEE ALSO

bark(8C)

Action	Argument	Description
`alarm`	message	prints the message and rings the keyboard bell
`disable`	rule_name	removes a rule instance from the active rule set
`enable`	rule_name	adds a rule instance to the active rule set
`execute`	command	executes a shell command
`hidecolumn`	metric_name	hide a column of meters
`hiderow`	entity_name	hide a row of meters
`highlight`	meter	uses a vivid colour scheme for the meter
`nil`		does nothing
`notify`	message	prints the message
`showcolumn`	metric_name	display a column of meters
`showrow`	entity_name	display a row of meters
`unhighlight`	meter	revert the meter to its normal colour scheme

Bibliography

[AL83] Bovas Abraham and Johannes Ledolter. *Statistical Methods for Forecasting.* Wiley series in probability and mathematical statistics. John Wiley and Sons, 1983.

[Amm91] Leendert Ammeraal. *C++ for programmers.* John Wiley and Sons, 1991.

[AR89] Layth C. Alwan and Harry V. Roberts. Time Series Modelling for Statistical Process Control. In J. Bert Keats and Norma Faris Hubele, editors, *Statistical Process Control in Automated Manufacturing*, volume 15 of *Quality and Reliability*, pages 45–65. Marcel Dekker, 1989.

[BACW90] Amatzia Ben-Artzi, Asheem Chandna, and Unni Warrier. Network Management of TCP/IP Networks: Present and Future. *IEEE Network*, 4(4):35–43, July 1990.

[BCJZ88] Lawrence Bernstein, Robert N. Cronk, Alan R. Johnston, and Douglas N. Zuckerman. Expert Systems in Network Operations and Management. *IEEE Network*, 2(5):5–6, September 1988.

[BCW88] Richard A. Becker, John M. Chambers, and Allan R. Wilks. *The New S Language: A Programming Environment for Data Analysis and Graphics.* Wadsworth and Brooks/Cole, 1988.

[BH86] Roger W. Berger and Thomas H. Hart. *Statistical Process Control: A guide for implementation*, volume 8 of *Quality and Reliability.* Marcel Dekker, 1986.

[BJ70] George E. P. Box and Gwilym M. Jenkins. *Time Series Analysis: Forecasting and Control.* Holden-Day, 1970.

[BO87] Bruce L. Bowerman and Richard T. O'Connell. *Time Series Forecasting: Unified Concepts and Computer Implementation.* Duxbury Press, second edition, 1987.

[CBH+92] J. H. Chou, C. R. Buckman, T. Hemp, A. Himwich, and F. Niemi. AIX NetView/6000. *IBM Systems Journal*, 31(2):270–285, 1992.

[CC87] C. K. Chui and G. Chen. *Kalman Filtering: with Real-Time Applications*, volume 17 of *Springer Series in Information Sciences.* Springer-Verlag, 1987.

[CCB88] Robert N. Cronk, Paul H. Callahan, and Lawrence Bernstein. Rule Based Expert Systems for Network Management and Operations : An Introduction. *IEEE Network*, 2(5):7–21, September 1988.

[CD91] H. Elston Carter and Janario P. Dia. Evaluating Network Management Systems: Criteria and Observations. In I. Krishnan and W. Zimmer, editors, *Integrated Network Management, II*, pages 213–223. Elsevier Science Publishers B.V., 1991.

[Cha84] C. Chatfield. *The analysis of time series: an introduction.* Chapman and Hall, third edition, 1984.

[CM90] Teresa Cochran and Joseph M. Mellichamp. AUTOREC: An Automated Error Recovery System for Network Management. *IEEE Network*, 4(2):14–18, March 1990.

[Dei90] H. M. Deitel. *Operating Systems.* Addison Wesley, second edition, 1990.

[DI89] Murthy V. Devarakonda and Ravishankar K. Iyer. Predictability of Process Resource Usage: A Measurement-Based Study on UNIX. *IEEE Transactions on Software Engineering*, 15(12):1579–1586, December 1989.

[DSWY91] Alexander Dupuy, Soumitra Sengupta, Ouri Wolfson, and Yechiam Yemini. NETMATE: A Network Management Environment. *IEEE Network*, 5(2):35–43, March 1991.

[DSY+89] A. Dupuy, J. Schwartz, Y. Yemini, G. Barzilai, and A. Cahana. Network Fault Management: A Users View. In B. Meandzija and J. Westcott, editors, *Integrated Network Management, I*, pages 101–107. Elsevier Science Publishers B.V. (North-Holland), 1989.

[EGH+86] R. L. Ennis, J. H. Griesmer, S. J. Hong, M. Karnough, J. K. Kastner, D. A. Klein, K. R. Milliken, M. I. Schor, and H. M. Van Woerkom. A continuous real-time expert system for computer operations. *IBM Journal of Research and Development*, 30(1):14–28, January 1986.

[FLNO88] M. Feridum, M. Leib, M. Nodine, and J. Ong. ANM : Automated Network Management System. *IEEE Network*, 2(2):13–19, March 1988.

[FQ87] I. Fassina and P. Quaroni. A Statistical Model for the Capacity Planning of an Interactive System. In S. Fdida and G. Pujolle, editors, *Modelling Techniques and Performance Evaluation*, pages 103–119. Elsevier Science Publishers B.V., 1987.

[GHJ+89] J. Goldman, P. Hong, C. Jeromnimon, G. Louit, J. Min, and P. Sen. Integrated Fault Management in Interconnected Networks. In B. Meandzija and J. Westcott, editors, *Integrated Network Management, I*, pages 333–344. Elsevier Science Publishers B.V. (North-Holland), 1989.

[GMS89] Rodney M. Goodman, John Miller, and Padraic Smyth. Real Time Autonomous Expert Systems in Network Management. In B. Meandzija and J. Westcott, editors, *Integrated Network Management, I*, pages 599–624. Elsevier Science Publishers B.V. (North-Holland), 1989.

[Har81] A. C. Harvey. *Time Series Models.* Philip Allen Publishers, 1981.

[Har89] Andrew C. Harvey. *Forecasting, structural time, series models and the Kalman filter.* Cambridge University Press, 1989.

[Hel93] Joseph L. Hellerstein. An Introduction to Modelling Dynamic Behaviour With Time Series Analysis, May 1993. IBM Research Report *RC 18893.*

[Hol89] David Holden. Predictive Languages for Management. In B. Meandzija and J. Westcott, editors, *Integrated Network Management, I*, pages 585–596. Elsevier Science Publishers B.V. (North-Holland), 1989.

[IBM81] IBM. Contingency Planning for Catastrophic Events in Data Processing Centers, September 1981. IBM publication *G320-6729-0*.

[JKM+89] C. Joseph, J. Kindrick, K. Muralidhar, C. So, and T. Toth-Fejel. MAP Fault Management Expert System. In B. Meandzija and J. Westcott, editors, *Integrated Network Management, I*, pages 627–636. Elsevier Science Publishers B.V. (North-Holland), 1989.

[JS91] Sedrak Jidarian and Dawn M. Shapiro. Traffic Administration and Engineering of a Corporate Wide Area Network. *IEEE Network*, 5(2):24–29, March 1991.

[Kle88] S. Mark Klerer. The OSI Management Architecture: an Overview. *IEEE Network*, 2(2):20–29, March 1988.

[Kle91] William Klein. Operations and Management of a Campus Packet Network. *IEEE Network*, 5(2):16–22, March 1991.

[Kob87] Hisashi Kobayashi. *Modeling and Analysis: An Introduction to System Performance Evaluation Methodology*. The Systems Programming Series. Addison-Wesley Publishing Company, 1987.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[Lav83] Stephen S. Lavenberg. *Computer Performance Modelling Handbook*. Academic Press, 1983.

[LMKQ89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley Publishing Company, 1989.

[MCE+86] K. R. Milliken, A. V. Cruise, R. L. Ennis, A. J. Finkel, J. L. Hellerstein, D. J. Leob, D. A. Klein, M. J. Massullo, H. M. Van Woerkom, and N. B. Waite. YES/MVS and the automation of operations for large computer complexes. *IBM Systems Journal*, 25(2):159–180, 1986.

[MFGL89] Akain Mouttham, Monica Frontini, Jonathan Griffin, and Simon Lewis. LAN Management Using Expert Systems. In B. Meandzija and J. Westcott, editors, *Integrated Network Management, I*, pages 491–502. Elsevier Science Publishers B.V. (North-Holland), 1989.

[Mur79] J. Murdoch. *Control Charts*. The Macmillan Press, 1979.

[New85] J. Newton. Strategies for problem prevention. *IBM Systems Journal*, 24(3/4):248–263, 1985.

[Oak86] John S. Oakland. *Statistical Process Control: A practical guide*. Heinmann Professional Publishing, 1986.

[Pot87] Dominique Potier. Computer Performance Evaluation and Expert Systems: A Survey. In S. Fdida and G. Pujolle, editors, *Modelling Techniques and Performance Evaluation*, pages 205–219. Elsevier Science Publishers B.V., 1987.

[PT89] Craig Partridge and Glenn Trewitt. The High Level Entity Management System (HEMS). *IEEE Network*, 2(2):37–42, March 1989.

[Shu86] R. H. Shumway. Autobox (Version 1.02). *The American Statistician*, 40(4):299–301, November 1986.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, second edition, 1991.

[Svo76] Liba Svobodova. *Computer Performance Measurement and Evaluation Methods: Analysis and Applications*. Computer Design and Architecture. Elsevier Science Publishers B.V., 1976.

[TF93] Tihamer T. Toth-Fejel. Personal communication, August 1993.

[TFJS90] Tihamer Toth-Fejel, Celia Joseph, and Allen Sherser. Knowledge-based Systems for Managing Manufacturing Local Area Networks. In *Industrial Process Control*, April 1990.

[TWGJ91] Gary Tjaden, Mark Wall, Jerry Goldman, and Constantine N. Jeromnimon. Integrated Network Management for Real-Time Operations. *IEEE Network*, 5(2):10–15, March 1991.

[VST91] Tania Volochine, G. S. Subramanian, and Ronald W. Toth. Network Management and Traffic Analysis for CICNet. *IEEE Network*, 5(5):41–50, September 1991.

[WM88] Steve Wilbur and Madjid Merabti. Towards Network and Distributed Systems Performance Management. In I. N. Dallas and E. B. Spratt, editors, *Issues in LAN Management*, pages 147–159. Elsevier Science Publishers B.V., 1988.

[Zha93] Zhiqing Zhang. CR: A Monitor for Distributed Systems. Master's thesis, Victoria University of Wellington, 1993.

[ZK89] William A. Zazaret and William Klingler. Tuning Computer Systems for Maximum Performance: A Statistical Approach. In Khosrow Dehnad, editor, *Quality Control, Robust Design, and the Taguchi Method*, pages 175–185. Wadsworth and Brooks/Cole, 1989.