Eidgenössische Technische Hochschule Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Department of Information Technology and Electrical Engineering

Computer Engineering and Networks Laboratory

Master's Thesis

Event Correlation Engine

Andreas Müller
Spring Term 2009

Tutors:
Christoph Göldi
Bernhard Tellenbach

Supervisors:
Prof. Bernhard Plattner
Stefan Lampart

open systems

Institut für
Technische Informatik und
Kommunikationsnetze

Preface

Abstract

As modern IT systems running on distributed platforms tend to become more and more complex, the required management effort grows as well, and it is no longer economic, to manage a complete system manually.

In the search for a way to handle status and incident messages from various subsystems in an automated way, the use of event correlation techniques has become widespread in recent years. Although many products for event correlation are readily available today, there is no universal solution for all applications, and event correlation remains an open research topic.

This thesis investigates the use of a correlation engine in the context of a global network offering various services, as a means to facilitate the monitoring of the network and of the individual services. After the analysis of the occurring event patterns, and the specification of the resulting requirements, a suitable correlation engine is presented, and its aptitude is investigated.

Outline

This master's thesis report is split into the following chapters:

Chapter Introduction presents the project and some background.

Chapter Event Pattern Analysis examines event statistics and discusses the most frequent patterns.

Chapter Survey of Existing Event Correlation Approaches reviews existing approaches as well as some of the available event correlation applications.

Chapter Event Correlation Engine Specification discusses the requirements and a specification for a suitable correlation engine.

Chapter Implementation explains the implementation details and concepts.

Chapter Evaluation investigates the aptitude of the developed correlation engine for real-world events.

Chapter Conclusions and Outlook reviews the project and draws some conclusions.

Author:	Andreas Müller	andrmuel@ee.ethz.ch
Tutors:	Christoph Göldi	chg@open.ch
	Bernhard Tellenbach	tellenbach@tik.ee.ethz.ch
Supervisors:	Prof. Bernhard Plattner	plattner@tik.ee.ethz.ch
	Stefan Lampart	stl@open.ch

Acknowledgements

This thesis would not have been possible without the support of various people, to whom I would like to express my gratitude.

I would like to thank Christoph Göldi from Open Systems for his continued support with explanations, ideas and feedback throughout this thesis, as well as Bernhard Tellenbach from the TIK, for supporting me with interesting ideas and a lot of feedback concerning both technical and formal aspects of this thesis.

Furthermore, I would like to thank Open Systems for providing the possibility to realize this thesis in a real-world environment, which made the task much more interesting. The people at Open Systems were very friendly and supportive; among others, I would like to thank Renato, Patrick, Thomas, Roel, Goetz, David and Stefan, for helping me identify problematic event patterns, and for providing ideas on how to handle them.

At the ETH, I would further like to thank Professor Bernhard Plattner from the TIK, for supervising this thesis, and for allowing me to do this master thesis in a collaboration between the TIK and Open Systems.

Last but not least, I also feel grateful towards the developers of Python, Vim, LaTeX, and various other free software applications, toolkits and libraries, which made the writing of this thesis and the development of the accompanying tools both more productive and more fun.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter discusses the task and its background, and gives a quick introduction to event correlation. Furthermore, it explains some of the terms used throughout this thesis.

1.1 Background

This master thesis was realized at Open Systems AG, in collaboration with the Computer Engineering and Networks Laboratory (TIK), which is part of the Information Technology and Electrical Engineering Department (D-ITET) at ETH Zurich.

1.2 Problem Statement and Setup

Open Systems operates more than 1500 hosts all around the world, which run services for Open Systems customers, such as spam protection, firewalls or internet proxies.[1]

In order to notice problems as early as possible, log messages generated by the services are automatically matched against a large set of signatures for known messages. If a log message matches a signature, which identifies it as sufficiently significant, a new event with the log message and a description of the matching signature is automatically created. Furthermore, an event is also generated, if the log message matches no known signature at all. The generated events then end up in the ticketing system of Open Systems (either as a new ticket, or appended to an existing ticket, if a ticket for the sending host is already open), and are handled by a human operator, who is responsible for investigating and resolving the problem.

In practice, a single problem often results in many generated events, which leads to complex and large tickets, making it difficult for the operator to identify the root of the problem. In some cases, events are also generated, even though there is no problem at all (false positives), which leads to unnecessary tickets.

As each ticket has to be handled manually, unnecessary or overly complex tickets waste valuable time. On the other hand, overseeing an actual problem creates inconveniences for the customer and might lead to a violation of the Service Level Agreement (SLA).

[1] A full list of services can be found at `http://open.ch/en/services/`.

1.3 Task Description

This thesis investigates the use of event correlation as a possible means to mitigate this problem. As specified in the assignment (cf. Appendix D), a correlation engine will be presented, which can process and correlate incoming events automatically and in quasi real-time, according to rules specified in a suitable configuration language and dependent on the internal state as well as on external information sources.

The goal for the correlation engine is to identify known event patterns, group, filter and prioritize the events to make the tickets more readable, and ultimately, to take some of the load off the operators.

1.4 Introduction to Event Correlation

Generally speaking, an event is simply anything, which happened at some moment in time. This could be the ringing of a phone, the arrival of a train, or anything else, which *happened*. In the context of computing, the term event is also used for the message, which conveys, what has happened, and when it has happened. Examples here could be a message indicating, that a web page was requested from a server, which is sent to the system log, a message, that a network link is down, or a message, that the user pressed a mouse button, sent to the User Interface (UI).

The meaning of the term correlation becomes clearer, by inserting a hyphen at the right place: we are looking for co-relation, i.e. for relations between different events. Event correlation is usually done to gain higher level knowledge from the information in the events, e.g.

- to identify extraordinary situations,
- to identify the root-cause of a problem,
- or to make predictions about the future and to discover trends.

Various approaches for doing event correlation exist (the most important ones are discussed in Chapter 3), and applications can be found in various areas, such as

- market and business data analysis (e.g. detecting market trends),
- algorithmic trading (e.g. making predictions about the development of a stock price),
- fraud detection (e.g. detecting uncommon use patterns of a credit card),
- system log analysis (e.g. grouping similar messages, and escalating important events)
- or network management and fault analysis (e.g. detecting the root-cause of a network problem).

This short (and definitely incomplete) list already shows, that event correlation is a broad topic, with numerous applications. In order to limit the volume of this thesis, we will discuss event correlation mainly from the perspective of system log and network event analysis, largely ignoring other applications.

1.5 Terminology

Throughout this thesis, the term **service** usually designates one or more server processes, that perform some task for a client. Although on Transmission Control Protocol (TCP) level, a service is usually associated with a port, served by some daemon, a more generic view is assumed here, and a service may even be distributed among several hosts in some cases.

A **daemon** is a background process on a computer, that provides a service to local or remote users, or to another program.

An **incident** is a temporarily abnormal situation, which requires attention. One or more events may be generated for an incident, and an incident may stretch over multiple hosts.

1.5.1 False Positive and False Negative

A **false positive** is an event or alarm indicating an extraordinary situation, even though there is no problem, e.g.

- a service is reported to be unavailable even though it isn't,
- an email is classified as spam even though it is not spam.

A **false negative** is an event indicating, that the situation is normal, even though there is a problem, e.g.

- a service is reported to be available even though it isn't,
- an email is classified as ham, even though it is spam.

If the healthiness of a service is checked and no event is generated, or a generated event is later ignored, even though the service has a problem, this can also be considered as a false negative, as no message is conventionally considered to signify, that there is no problem[1].

On the other hand, if a problem is not reported, because there is no check covering that problem at all, this should not be called a false negative, because there was no decision, that could have resulted in a positive or negative answer (e.g. if a service does not produce any log messages *at all*, then there are no false negatives). Reducing the number of false negatives does not help in this case; the solution is simply to introduce new checks to cover the overlooked problem.[2]

1.5.2 Event Correlation Terminology

As event correlation is a rather young research area (and an even younger area for marketing), the terms are sometimes used ambiguously in research papers, and even more so in product marketing. This thesis tries to follow the most widely used terminology.

An **event source** generates events. For this thesis, the syslog is often the event source; but in a more abstracted view, an event source can be anything from a hardware device (e.g. a sensor) to a software process, or even a user.

An **event sink** receives events. In practice, an event sink could be a database to store the events, a ticketing system, or some process, that takes further action, such as sending an email.

[1] This is generally the case under Unix; for instance, many command-line tools, such as `rm`, `mount`, `mkdir`, etc. give no feedback, if the command was executed correctly

[2] This is however not a task for the correlation engine. This thesis is therefore more focused on genuine false negatives, assuming that there is at least *some* evidence for a problem, and the task is to interpret it correctly.

A **raw event** (primitive event) is an event, that is generated by an event source, from outside the correlation engine, for instance an event generated directly from a log message.

An **input event** is an event at the input of a specific correlation operation or the correlation engine. Conversely, an **output event** is generated at the output of a correlation operation or the whole correlation engine.

A **compressed** event is an event representing multiple identical events, but which does not contain the individual events it represents.

A **derived event** (synthetic event) is an event, that was generated by the correlation engine, e.g. because an event was generated during a given state of the correlation engine, because a given set of events occurred, or because a given event was not generated for a specified time (timeout event).

A **composite event** is an event generated by the correlation engine, which contains multiple other events, e.g. raw, derived or composite events. A composite event usually represents some information on a higher level.

The terms **alarm** and **alert** are used interchangeably in this thesis, and indicate a notification about a situation, that requires attention.

For an explanation of event correlation operations, please refer to Section 3.2.

1.5.3 CEP and ESP

The two terms Complex Event Processing (CEP) and Event Stream Processing (ESP) can often be found in the context of event processing. Although the two concepts are related, and partially overlapping, they are not the same. In [50], CEP is defined as “Computing that performs operations on complex events, including reading, creating, transforming, or abstracting them”, whereas ESP is defined as “Computing on inputs that are event streams”.

As a coarse rule of thumb, it can be argued, that ESP is focused more towards events streams, with regularly generated events, whereas CEP is more concerned with irregularly generated events.

A typical example for a ESP operation is the calculation of a moving average for the value of a share from a stock ticker. For CEP on the other hand, an example is the identification of the root cause for problems in a network, from multiple events indicating different symptoms.

1.5.4 SIM

The term Security Information Management (SIM) encompasses the collection, processing, storing and reporting of security information and events. Depending on the background, various variants of the term SIM can be found, with varying definitions. In [40], the author explains:

> There has been a vendor-fuelled explosion in acronyms around SEM, and you will see them referred to variously as SEM, SIM, CSEM, CIEM, and ESM systems. All of these perform broadly similar functions with differing scalability, utility, user-friendliness, and price.

In this thesis, only the term SIM will be used.

1.5.5 Acronyms

A list of all acronyms used throughout this thesis can be found in Appendix G.

1.6 Typography

A `teletype` font is generally used to designate script names, shell interaction, host names, variables and program code. Shell interaction is further designated by the $ sign (normal commands) or by the # sign (root commands) at the start of the line. Values for an option or a parameter are *emphasized*.

1.6.1 Host Names

The host names `host-a`, `host-b`, . . . `host-z` will be used for different, unrelated hosts, whereas names like `host-a-1` and `host-a-2` will be used for different hosts belonging to the same cluster (such as two hosts in a hot standby configuration).

Chapter 2

Event Pattern Analysis

In order to specify a correlation engine, it is necessary to know, what kind of event patterns can be expected. To learn about possible event patterns, we therefore examined existing syslog[1] messages and corresponding tickets from hosts operated by Open Systems.

To get a more representative overview, we examined a database dump of all log messages generated during one month (a total of 58386 messages from 896 host with 223 different event types), rather than looking at incoming messages in real-time, as the finished correlation engine is supposed to.

Since the examined messages are exclusively messages that ended up in a (new or existing) ticket, each log message is also an event. The opposite is for the moment generally also true, as composite events are not considered yet. The two terms are therefore used almost interchangeably, although the term event is preferred in some cases, to suggest generality.

We did the analysis in this chapter with Python, using SciPy and IPython.[2] The mentioned scripts can be found on the CD-ROM under `/log_analysis/`.

In the first two sections of this chapter, the logs will be analyzed with a data-mining approach, in the hope to learn more about event statistics, event patterns and dependencies between event types and hosts. The goal of this automated analysis is to get an overview of the data and to find event patterns. It should be noted, that this is a different goal than the one for the correlation engine (which will analyze events in real-time, rather than using collected data), and it is unlikely, that the approach taken here will be suitable later.

2.1 Statistics

As the number of the examined events is quite large, it is reasonable to do some automated analysis, before looking at specific events in detail and identifying possible event patterns manually. In this section, the statistics of the log messages are analyzed, and a look from a high-level perspective is taken.

[1] Syslog is a protocol used to convey system log messages on a network. Although many different implementations exists, a specification can be found in Request For Comment (RFC) 3164 [49].

[2] SciPy (available at `http://scipy.org`) and IPython (available at `http://ipython.scipy.org`) are libraries and an advanced shell for Python, which allow the use of Python in a similar fashion as how one might otherwise use products like Matlab.

2.1.1 Database Dump Format and Event Type Names

Conveniently, the database dump not only contains log messages, but also additional information, as well as information extracted from the log messages. The most interesting parameters of a given event are:

- Short name (name of the rule that matched the log message)
- Host name (name of the host that generated the log message)
- Message time (date and time, when the event was generated)

The short name is in general a name for the regular expression that matched the log message. Short names have the format `[A-Z][A-Z0-9-]+(:[A-Z0-9-]+)+` and they usually specify the service or protocol layers and the problem or notification carried in the matched log message. As an example, the short name `NIC:ETHERNET:LINKUP` indicates that an Ethernet Network Interface Controller (NIC) link was activated.

The meaning of the short names is however not strictly specified. In some rare cases, a short name stands for a set of log messages, that haven't been broken up more finely, and in a few cases, a simple form of log processing is already done, and a short name may stand for more than one message. Specifically, this concerns the two event types `GUMA:MULTI` and `GUMA:LIMIT`, which indicate, that event compression, respectively rate limiting has been applied. These mechanisms are however used very defensively at the moment, and no events of these two types were generated in the month under scrutiny.

As the short names already designate classes of events, they can be used as event types. This also saves a lot of work, as the classification done by the Open Systems engineers can be used, rather than trying to classify thousands of log messages in an own scheme. Although the final event correlation engine may well use a different classification, the existing classification can be very helpful to identify event patterns.

A lot more information is available in the database dump, and also in the log message itself, such as the time the message arrived at the database, the syslog severity and facility, etc. As the goal is to get a high-level overview, this information is however ignored for the moment.

2.1.2 Most Frequent Event Types and Most Active Hosts

As already mentioned, there was a total of 58386 message that arrived in the month under scrutiny, or in other words an average of no more than 1-2 messages per minute. While this is quite a low value, the short-term load can be significantly higher, as the messages arrive in bursts.

First, it is interesting to see, which types of events are generated most often, and how many different hosts generate events of a given type. For this task, we wrote a small Python script called `logstats.py`.

Table 2.1 shows the five event types that occurred most often, along with the number of events and the number of different source hosts for each type. Although this is of minor concern at the moment, a short description of the event types is also given. The number of events in the top five sums up to 29831, which is slightly more than half of all the generated events. This suggests, that even with only a few correlation rules, the number of events could be significantly reduced. Reducing the number of events does not necessarily mean, that the number of tickets is reduced, as a lot of events might have ended up in the same ticket anyways[1]. Simple event

[1] This is especially true in the case of the `HSP:MESSAGE:UNKNOWN:CRITICAL` event, where more than 6000 of the 8612 events were generated by the two most active hosts. As there were tickets with up to around 1000 events, there were actually only rather few tickets with `HSP:MESSAGE:UNKNOWN:CRITICAL` events.

Event type	Messages	Hosts	Message description
WINBIND:CONF:ADCONN	12082	42	There was a problem when trying to connect to an Active Directory server.
HSP:MESSAGE:-UNKNOWN:CRITICAL	8612	10	One of various HTTPS proxy messages.
SES:RPROXLOGD:LOCK	3558	14	There was a problem starting a log daemon for the reverse proxy, because of an existing lock file.
MAIL:FRESHCLAM:ERROR	3261	37	An error occurred when trying to update the AntiVirus signatures.
NIC:ETHERNET:LINKUP	2318	220	An Ethernet NIC link was activated.

Table 2.1: Top five event types.

aggregation may therefore not be too helpful for human operators, as their load depends much more on the number of tickets, than on the number of events. Still, a few simple aggregation rules at the input can certainly reduce the processing load for the rest of the engine.

A look at the corresponding tickets suggests, that a lot of the events were generated because of simple problems, such as problems with the connectivity, ISP outages, or in some cases power failures resulting in reboots. Since a certain degree of instability of internet and power providers has to be expected in some countries, and short outages may be acceptable on a non-critical server, a lot of these events could be ignored as long as they do not exceed a certain measure.

A similar analysis as for the event types can also be done for hosts. Table 2.2 shows the most active hosts (the host names have been anonymized).

Host	Number of Events	Number of Event Types
host-a	3863	5
host-b-1	3676	9
host-b-2	2635	8
host-c	1996	23
host-d	1869	9

Table 2.2: The five most active hosts.

The five most active hosts generated 14039 events, which is about 24% of the total. What is noteworthy is that even when a host generates many events, there are often only very few different event types.

2.1.3 A Graphical Look at the Events and Event Bursts

In a next step, a graphical look at some plots of the events is taken, to get a better feeling for the message patterns.

To be able to treat the event streams as signals, the number of events of each type during a given time is counted and used as a measure of “signal strength”. Although the log messages have time stamps with a resolution of one second, the number of messages arriving in one minute has commonly been counted, because the log time is not usually precise down to one second, and because too few messages arrive (which would also make a plot rather boring, as in any given

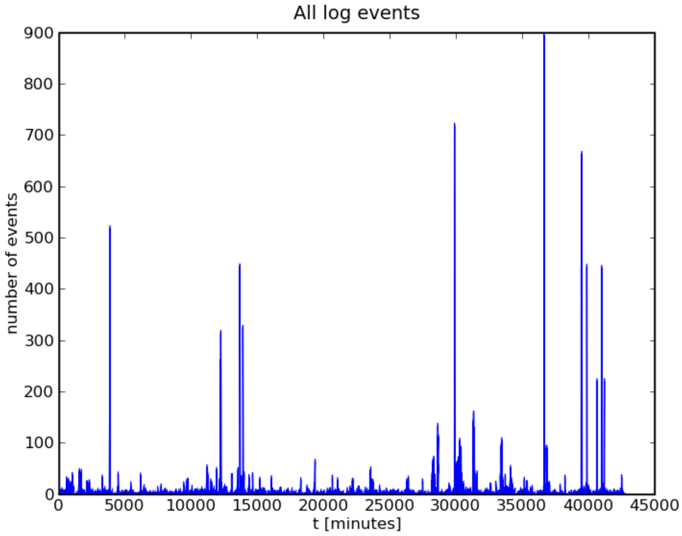


Figure 2.1: Plot of all events during a month.

second, usually zero or one messages arrive). Choosing a lower resolution will also considerably lower the processing effort, when the correlation is calculated later.

Figure 2.1 shows a plot of all events from any host during the observed month. What should be noted is that the events arrive in bursts, rather than continuously. This is not unexpected, as one problem often generates a multitude of events, but it has several implications for the implementation of a suitable correlation engine, such as that the engine must be able to cope with event rates much higher than the average event rate. The plot shows, that in some cases, several hundred events per minute were generated (whereas the average is less than two events per minute).

When looking at all events generated by a single host, it can be seen even better, that the events are generated in bursts. Figure 2.2 shows all events generated by one host. This particular host generated 909 events during the whole month. It is not uncommon, that there are only a few bursts of events and a large part of all events by a given host is generated in a short time.

To get an idea of the “burstiness” of an event stream, we can ask, what percentage of all events arrived in the largest burst of a given time. Using the script `analyze_bursts.py` with a sliding window of one hour, we answered this question. Table 2.3 shows the results for the five most active hosts. In average (across all hosts), a host generated 63.5% of all messages of a month during its most active hour (the standard deviation is ±31.7%). This result may be slightly biased because of hosts that generated only few messages, as for instance for a host that generated only one message, it is inevitable that 100% of all messages were generated within a window of one hour. Considering only hosts that generated at least 100 events yields the slightly lower number of an average of 58.0%, with a standard deviation of ±31.5%. For a more detailed insight, Figure 2.3 shows a histogram of the burst ratios of all hosts, as well as the corresponding cumulative distribution.

Doing the same analysis for different event types instead of hosts results in more varied results, i.e. the “burstiness” depends strongly on the event type. The most frequent event, which signifies

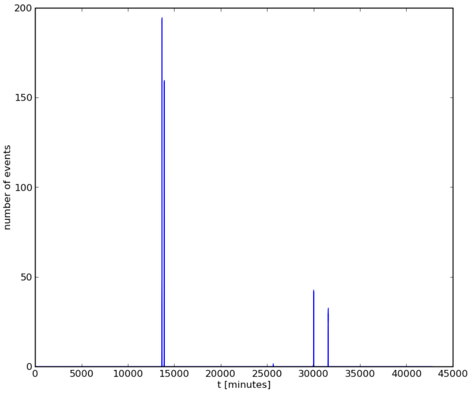


Figure 2.2: Plot of all events generated by a typical host throughout a month.

Host	Events in total	Events in largest burst	Ratio
`host-a`	3863	3686	≈ 95%
`host-b-1`	3676	1544	≈ 42%
`host-b-2`	2635	1109	≈ 42%
`host-c`	1996	1410	≈ 71%
`host-d`	1869	1798	≈ 96%

Table 2.3: Largest one-hour bursts on the most active hosts.

problems with the connection to an active directory server, clearly occurs in bursts, as shown in Figure 2.4. Other events, such as reboots, usually occur one at a time, as can be seen in Figure 2.5 (however, some bursts occurred for reboots as well – a possible reason could be a power outage at a site with many hosts).

Looking at the percentage of messages sent during the largest one-hour bursts reveals an average of 46.4% with a standard deviation of 35.6%. Considering only event types with 100 or more events again leads to a smaller number, 24.1% with a standard deviation of 27.4%. This makes sense, as events of one type are usually generated on many independent hosts. Even though events may still be generated in bursts, there may be many independent bursts from different hosts for a given event type. Figure 2.6 shows the cumulative distribution of the burst ratios of different event types, i.e. each bar shows on the y-Axis, how many event types there are, which a had largest burst (relative to the total number of events of that type), that is smaller or equal to the percentage value on the x-Axis. It is obvious that all event types had at most 100% of their events in the largest bursts (i.e. the y-value for $x = 100\%$ is the total number of events), and (since only event types with at least one event were counted) that no event type had zero events in the largest burst ($x = 0\% \Rightarrow y = 0$).

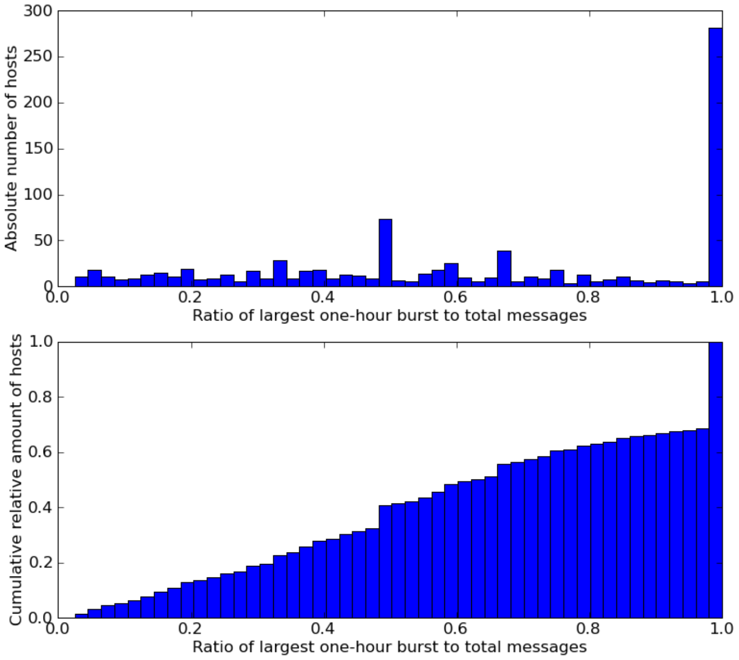


Figure 2.3: Histogram of the burst ratio of different hosts: Absolute numbers (top) and cumulative distribution (bottom).

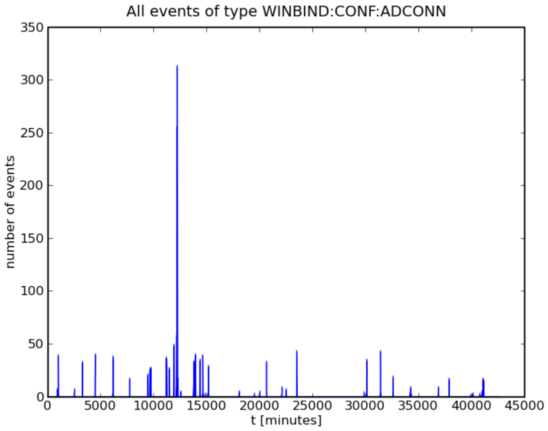


Figure 2.4: Plot of all active directory connection problem events.

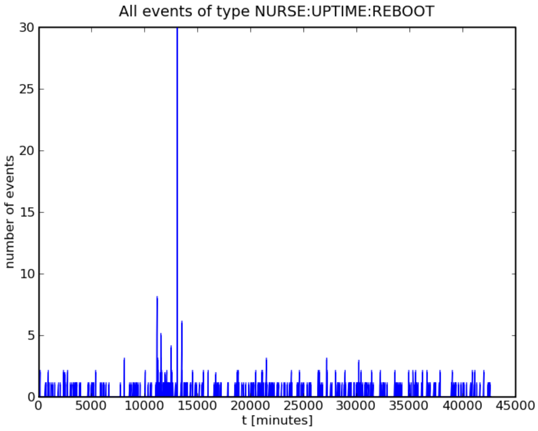


Figure 2.5: Plot of all reboot events.

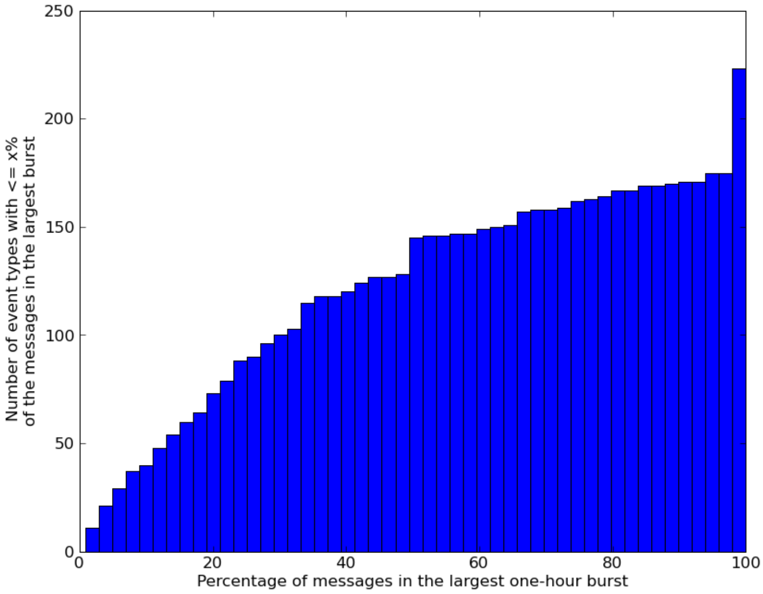


Figure 2.6: Cumulative distribution of the burst size of different event types.

2.2 A First Look at Correlation

2.2.1 The Naive Correlation Approach

The first naive correlation approach is to simply treat each event stream as a signal in time and look at the correlation between pairs of event streams with different time shifts.

A signal $x[t]$ indicates, how many log messages for event type x were generated at the time t. The correlation between two time discrete signals $x[t]$ and $y[t]$ is defined as

$$c[t'] = \sum_{t=0}^{N} x[t]y[t+t'], \qquad t' \in [-W, W]$$

where t' is the time shift, W is the correlation window and N is the length of the signal (values outside the signal are defined as zero). As the time shift is added to $y[t]$, a correlation with a positive shift indicates, that $y[t]$ is delayed with respect to $x[t]$.

For the moment, the analysis is done without distinguishing between different source hosts, although this is certainly a refinement that should be implemented later.

Since some log messages are generated at a much higher rate than others (even if the problem exists for the same amount of time), it is useful to normalize each stream, such that the signal energy

$$E = \sum_{t=0}^{N} |x[t]|^2$$

is one.

This experiment is implemented in `correlate_events.py`. Figure 2.7 shows the correlation between the two event types `WINBIND:CONF:ADCONN` and `SES:RPROXLOGD:LOCK`. From the plot, it looks like there is a strong correlation at a time shift of -16392 minutes. Unfortunately, this makes absolutely no sense – it is rather unlikely that events of the types `SES:RPROXLOGD:LOCK` (lock file problems) are somehow related to events of the type `WINBIND:CONF:ADCONN` (Active Directory connectivity problems) eleven days later, and the correlation between these two event types (which have been deliberately chosen to be unrelated) should be more or less zero.

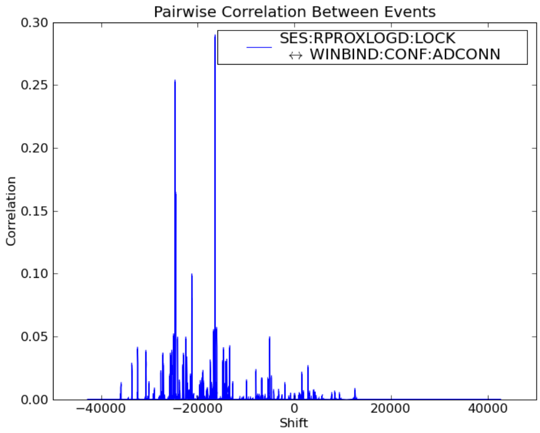


Figure 2.7: Correlation between two event streams.

The problem is that when all hosts are treated as a combined event stream, large bursts of events in the two streams will always look like correlation. It is therefore necessary to treat each host separately. This will of course only allow us to detect correlation between events on the same host. Correlation of events between events on different hosts will be examined separately, later.

2.2.2 Per-Host Correlation Across Event Types

In order to correlate each host separately, the event streams have to be considered as a two-dimensional discrete signal $x[t, h]$, which indicates, how many events of type x were generated by host h at time t. The correlation between $x[t, h]$ and $y[t, h]$ is then

$$c[t'] = \sum_{t=0}^{N} \sum_{h=0}^{H} x[t, h] y[t + t', h], \qquad t' \in [-W, W]$$

where t' is the time shift, W is the correlation window, N is the length of the signal and H is the number of hosts. This means, the correlation is still done only in time and across event types, but for each host separately, and the result is summed up across all hosts (which reflects the assumption that hosts are independent, but that the correlation between two given event types is similar on each host).

The script `correlate_events.py` was extended to implement this new approach. As a result, correlation between `SES:RPROXLOGD:LOCK` and `WINBIND:CONF:ADCONN` is now zero everywhere, as one would expect[1]. Figure 2.8 shows the correlation between two other event types, `HOTD:SYNC:DUPLICATEMASTER` (which indicates that there are two master hosts in a hot standby configuration) and `NETWORK:IP:ADDRESS:THEFT` (which indicates a duplicate Internet Protocol (IP) address in the network) with a time shift window of 10 hours. The plot clearly shows that there is a strong correlation between the two event types, for a time shift close to

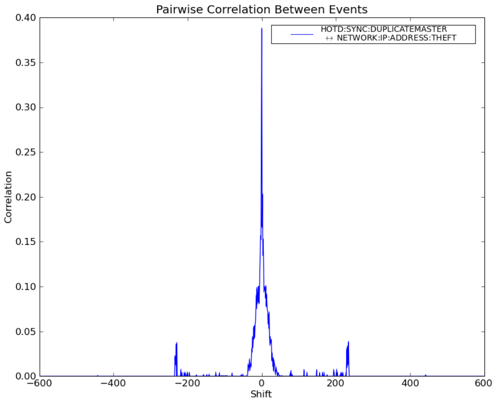


Figure 2.8: Correlation between two dependent event types.

[1] For performance reasons, hosts that did not generate at least one event of both correlated event types are ignored. As the set of hosts which created `SES:RPROXLOGD:LOCK` events and the set of hosts which created `WINBIND:CONF:ADCONN` events are disjoint, a calculation is not even done by the script in this case.

zero. This is of course not surprising – if a hot standby slave becomes master, while the master is still active, both use the same IP address, and the IP address theft event is generated.

The most interesting results of the correlation analysis are the maximum value for the correlation and the corresponding time shift, i.e.

$$\max c(t') \quad \text{and} \quad \underset{t'}{\operatorname{argmax}}\, c(t')$$

If we focus on these two results only, rather than looking at each event type pair separately, all event types can be plotted in one matrix. This mode has been implemented in `correlate_events.py`. Figure 2.9 shows the corresponding matrix with the correlation between the 13 most frequent events types (all event types with at least 1000 events) within a time window of 60 minutes. The size of the dots indicates the correlation, and the color is used as an additional dimension to visualize the time shift. As the correlation is symmetric, only one direction has been calculated. The correlation of an event type to itself is of course always one at shift zero; these calculations are also omitted in the plot. A larger version with a matrix for all event types

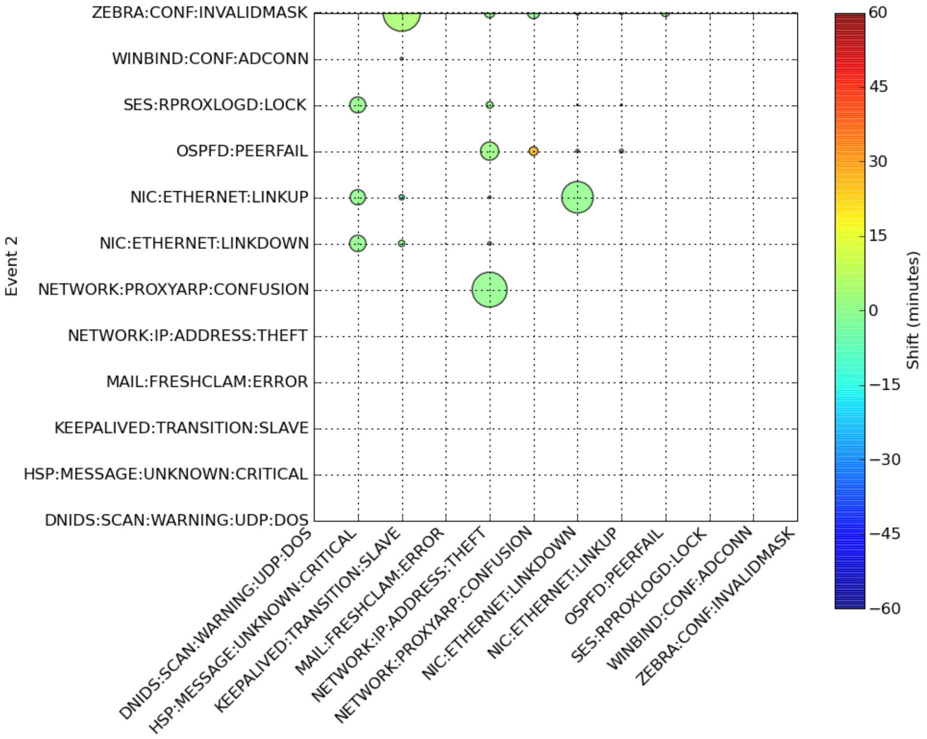


Figure 2.9: Maximum correlation and offset among top 13 events.

with more then 100 generated events can be found on the CD under `/log_analysis/results/`.

Although the results clearly show correlation between some event types, it should be noted that correlation does not imply causality. As seen at the beginning of this chapter, the correlation may simply be random, e.g. because of large bursts, or it is possible that two event types are correlated because of an unknown third factor, which induces events of both types. Even if there is causality between two event types, automated analysis can not tell us why there is causality, and in which direction. For this reasons, event patterns cannot be derived directly from the correlation results. However, the results will certainly be helpful in identifying event patterns manually.

2.2.3 Correlation Across Event Types and Hosts

While most hosts are independent, there are some cases where it is interesting to see the correlation between (different as well as the same) event types across two hosts. For instance, if two hosts are used in a hot standby configuration, some events are expected to be correlated, as when one host has a problem, the other is supposed to become active.

Figure 2.10 shows the correlation across event types and hosts on two mail serves in a cluster (all event types with at least four events on the two host are shown). For comparison, the analysis was also done on two other mail servers. Figure 2.11 shows the results.

The main problem, that can be seen on these four hosts, is an occasional failure when updating the anti-virus signature patterns. The finding, that the problem usually occurs on both hosts in a cluster simultaneously, is not surprising, because the problem is often caused by a common external factor, such as a temporary unavailability of the update hosts.

2.2.4 Comparison to Another Month

To assess the stability of the results, they can be compared to the results with data of the following month, March.

In March, 39210 events were generated by 680 hosts, with 201 different event types. The statistics for the top five event types look quite similar to the previous month; again, the top five event types (listed in Table 2.4) contain more than half of all messages.

Event type	Number of messages	Number of hosts
`HSP:MESSAGE:UNKNOWN:CRITICAL`	6079	7
`WINBIND:CONF:ADCONN`	5996	30
`TMON:RRD:ERROR`	4269	26
`SES:RPROXLOGD:LOCK`	3344	12
`MAIL:FRESHCLAM:ERROR`	2870	41

Table 2.4: Top five event types in March.

Table 2.5 shows the five most active hosts. Although the hosts are different ones than in the previous month (except for the two hosts `host-b-1` and `host-b-2`), the number of events per host is comparable.

Again, the “burstiness” can be analyzed, by asking for the relative size of the largest one hour burst. This yields the results shown in Table 2.6. While the numbers are generally smaller than for the previous month, the relative statements still hold, namely that the events on a specific host are generated in bursts, whereas the events of a given type are generated more continuously. Again, the results without a threshold of at least 100 generated events have a limited validity, and should be taken with a grain of salt (for the same reasons as explained in Section 2.1.3).

Looking at the correlation between different event types yields the picture shown in Figure 2.12 (the same event types as in Figure 2.9 were chosen, to allow a comparison). The main

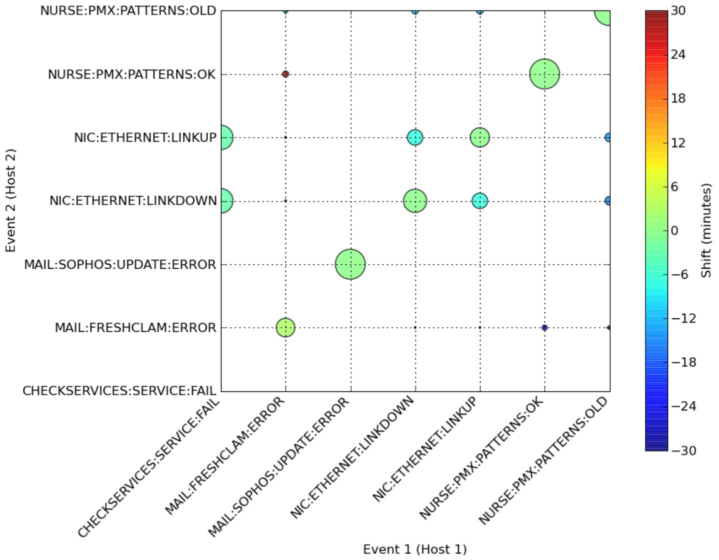


Figure 2.10: Correlation across hosts and events on two mail servers.

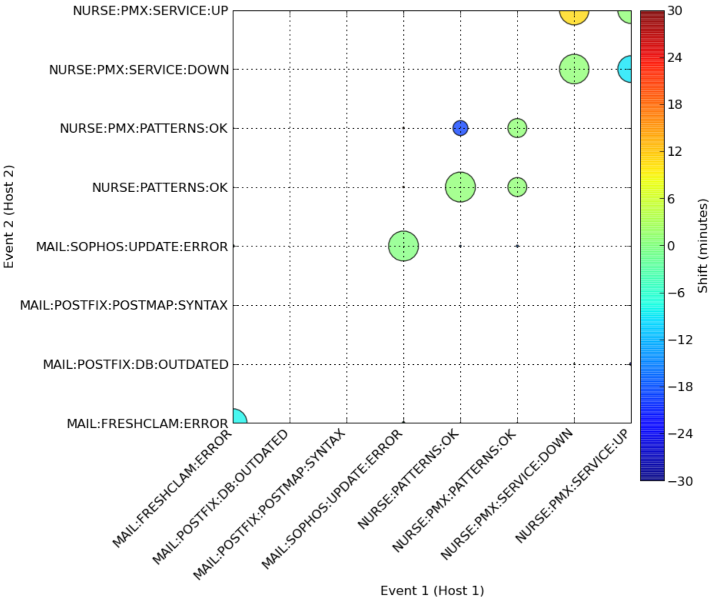


Figure 2.11: Correlation across hosts and events on two other mail servers (please note that the event types are *not* all the same ones as is Figure 2.10).

Host	Number of Events	Number of Event Types
`host-b-1`	3827	10
`host-b-2`	2658	11
`host-e`	2234	7
`host-f`	1749	9
`host-g`	976	6

Table 2.5: Top five hosts in March.

Description	Average relative size	Standard deviation
Largest one hour burst on a host	57.0%	33.2%
(hosts with at least 100 events only)	39.9%	31.6%
Largest one hour burst of a given event type	48.3%	36.0%
(event types with at least 100 events only)	19.5%	27.3%

Table 2.6: Burst analysis with the data of March.

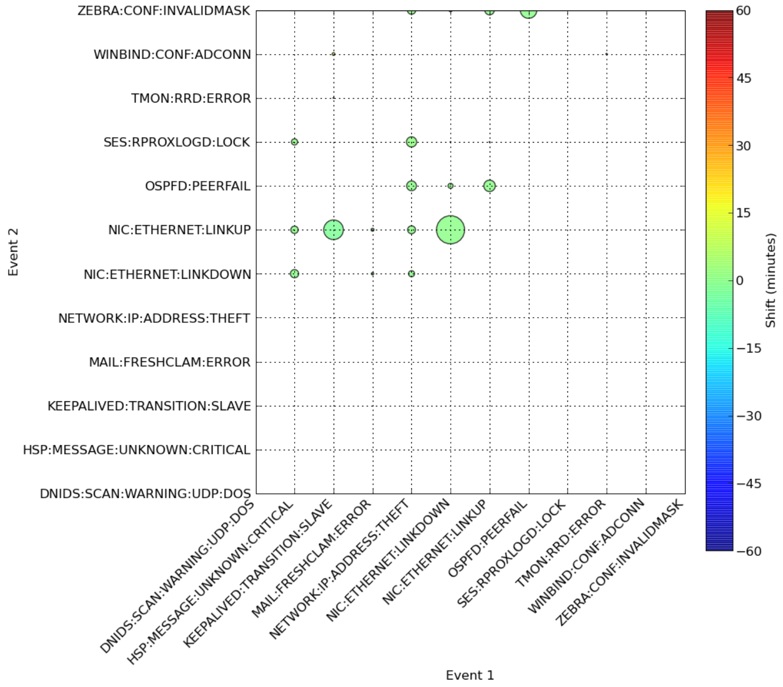


Figure 2.12: Correlation between event types in March.

difference is that there were no `NETWORK:PROXYARP:CONFUSION` events in March. Otherwise, the picture looks similar to the one from February, even though the values for the correlation coefficients (indicated by the size of the dots) differ in some cases.

2.3 Identification and Classification of Event Patterns

With the results of the analysis from the previous sections in mind, event patterns can now be identified. In the following sections, the most common patterns will be discussed. The discussion starts with the simpler cases involving events from a single host, and later moves on to more complex cases, including event patterns involving events from multiple hosts. The focus lies on patterns that are relevant for the correlation engine, i.e. patterns with automatically generated events, that need to be (pre-)processed by the correlation engine.

Some strategies to deal with each event pattern will be suggested throughout this chapter; they are for the moment however (intentionally) rather generic and fuzzy. A more formal specification of the possible correlation operations will be presented in Chapter 4.

Although the patterns are split into groups, this does not mean, that each pattern belongs to exactly one group. Most of the patterns can be assigned to more than one group, and it is likely that even in the finished correlation engine, there will be more than one way to deal with a given pattern.

The effort to identify the patterns presented in this section was greatly supported by feedback from various Mission Control engineers. Their well-thought-out suggestions and comments provided valuable input, for which I would like to express my gratitude.

2.3.1 Preliminary Remarks

2.3.1.1 Event Flow

Although the design of the correlation engine is not yet specified, a general idea already exists, and should be kept in mind throughout the rest of this chapter.

As already specified in the task description (which can be found in Section D), the correlation engine should work in a distributed fashion. The events are generated at hosts distributed all around the world. At the source host, a first correlation step can already be made, with the information that is available there. The preprocessed events are then forwarded to a central host, for further processing and correlation. Finally, the correlated events end up in tickets. Although two correlation steps are assumed here, a correlation in more than two steps is imaginable. Generally, it will be assumed throughout the rest of this chapter, that the correlation process at least forms a directed graph, i.e. each host forwards events towards a central host.

Whether a correlation step should be made at the source or at a central host is not always clear. Obviously, the central host has to handle a lot more events than the source hosts, as each source host is responsible only for its own events. The source host should therefore compress events if possible, and pre-process them as much as possible. On the other hand, accessing external information (such as the log of ISP outages[1]) is easier from a central host.

2.3.1.2 Low-level Signaling Schemes

On a very basic level, two signalling schemes can be distinguished:

- An event is sent when a problem first occurs, and another one when the problem is resolved (up/down events).

[1] At Open Systems, the connectivity to the hosts is monitored from a central location, and all connection problems are logged. The connection statistics are therefore available independently of the events generated at the individual hosts.

- As soon as a problem occurs, events are sent continuously, until the problem is solved (repeated problem events). The event rate can range from a few events per hour to multiple events per second. Often, the event rate does not contain any information, i.e. events are sent at a defined, fixed rate.

Obviously, both methods have its advantages and drawbacks. The most obvious advantage of up/down events is that they generate less traffic. Indeed, four of the five most frequent events listed in Table 2.2 are events, which are repeatedly sent (only `NIC:ETHERNET:LINKUP` is of the up/down type). On the other hand if events are sent as long as the problem exists, the chance of overlooking the problem is smaller.

While it is disputable, which signaling scheme is preferable in which case, it should be stressed, that it is important that the correlation engine is able to tell, when a problem begins and ends, in both cases. This means, that the following information is crucial:

- In the case of up/down events, the correlation engine must have the information, which events form a pair, and which event in a pair indicates the everything-fine state[1].
- In the case of repeated problem events, the correlation engine should know, how long after the last event, the situation can be considered normal again.

If this information is available, the two signaling schemes are for most purposes exchangeable (some differences of course remain, e.g., rate measuring is not possible, when up/down signaling is used).

2.3.2 Multiple Identical Events for a Persistent Problem

One of the most common sources for large quantities of events is a process, which continuously sends log messages, as long as the problem persists, i.e. a process using the repeated problem event signaling scheme explained in the previous section.

2.3.2.1 Practical Examples

As an example, if a host detects that another host is trying to use to same IP address, the `NETWORK:IP:ADDRESS:THEFT` event is in some cases generated at a rate of up to two events per second (in this case, the event rate does not contain any information).

2.3.2.2 Possible Solutions

One solution would be to compress the events, i.e. rather than forwarding 100 messages saying that event `X` occurred, the correlation engine (on the source host) should forward only one message, saying that event `X` occurred 100 times. This would of course require, that the first event is delayed for a defined time, to see whether more events of the same type are generated. If this is not acceptable, an alternative solution would be to forward the first event and then suppress or compress the following events for a given time.

Another solution would be to forward an event only if a defined threshold for the event rate (or for the event count) is exceeded, e.g. at least 20 events in 5 minutes[2]. Of course, this only

[1] With the naming scheme used at Open Systems, this information is partially contained in the semantics of the names, as most event pairs consist of some `X:Y:DOWN` and a corresponding `X:Y:UP` event (which is usually the default state). This is however currently *not* formally specified, and there are some exceptions, such as the event pair `NURSE:PROXYTEST:ALERT` and `NURSE:PROXYTEST:OK`.

[2] The term event rate is used as a generic term throughout this section. Appendix A discusses the different approaches to measuring it.

makes sense, if the event rate actually contains some information (in many cases, events are sent at a fixed rate, rather than a rate which indicates the frequency or magnitude of a problem).

Finally, the signalling could also be changed to the up/down scheme, by forwarding a corresponding up event, when the first input event occurs, and a down event when no more input events occurred for a given time.

2.3.3 Old Events

If a host loses its connection to the internet, events can not be forwarded, and events may be accumulated over a long period.

2.3.3.1 Possible Solutions

In order to avoid the creation of multiple tickets for old problems, old events could be forwarded as one composite event, if the accumulation time exceeds a certain value (e.g. if an event older than a day arrives, wait 10 minutes and if more old events from the same host arrive, create a composite event).

2.3.4 Late Events for Closed Tickets

If a new event occurs for a host with a ticket, that is open or has been closed a short time ago, the event is appended to the existing ticket, and the ticket is reopened if it was closed.

Although this behaviour generally makes sense, it can be annoying in some cases, for instance if a late event, for a problem that was already solved, arrives.

2.3.4.1 Practical Examples

An example is the `NURSE:SERVICE:DOWN` event, which indicates a problem with a service. In the ideal case, an operator looks at the corresponding ticket, solves the problem and closes the ticket. However, sometime later, a corresponding `NURSE:SERVICE:UP` event for the fixed service will be generated. Although this only indicates, that the situation has gone back to normal, the ticket is reopened, and has to be closed again manually.

2.3.4.2 Possible Solutions

Rather then reopening the ticket if the situation goes back to normal, it would make sense to generate an alert if the "return-to-normal event" is not generated. To achieve this, the correlation engine should have some way to affect the presentation in the ticketing system, and to receive a notification, when a ticket is closed. One possibility would be to forward events as either "active" or "inactive", with only "active" events having the possibility to reopen tickets. Additionally, a timeout operation is needed, to allow the reopening of the ticket, if the situation does not go back to normal in due time. Ideally, the timeout should start when the ticket is closed (and of course only if the event, that indicates the return to normal, has not arrived yet).

2.3.5 Irrelevant Unique Events

Some events can be ignored, if they are generated only once[1].

[1] Ignored should be read as "ignored by the operator" in this context. It may still make sense to forward the event for logging, and possibly the creation of an already resolved ticket.

2.3.5.1 Practical Examples

As seen in Section 2.2.3, it sometimes happens, that the anti-virus signature patterns can't be updated, for instance because of an ISP outage, or because the update servers are unavailable. Whenever this happens, an event is generated. If the problem occurs only once, this can be ignored, as the patterns will be updated the next time (e.g. one hour later).

2.3.5.2 Possible Solutions

The obvious solution is to introduce a rule to forward an event only if it occurs at least N times in a row (the separation between events in a row and isolated events requires either a timeout or an event specifying that the situation is normal again after an error). Additionally (or alternatively), a rule to specify a rate threshold might be used.

2.3.6 Flickering Services

Sometimes, a problem is only temporary, and it disappears by itself, after a short time. Often, this repeats itself later, leading to continuous short service outages, similar to the flickering of a light bulb, that is connected to an unreliable power source. This flickering may last for some minutes, hours or even days.

Such a behaviour is often caused by external factors, that can not be fixed. As each short outage leads to new events and to a new, or newly reopened ticket, this pattern is particularly annoying.

2.3.6.1 Practical Examples

A common example are problems caused by temporary internet outages. Unstable ISPs sometimes have multiple subsequent outages within a short time frame. Such outages can result in multiple event pairs, such as `VRRP:MONITOR:VPN:DOWN` events followed `VRRP:MONITOR:VPN:UP`, or temporary event bursts, such as the `WINBIND:CONF:ADCONN` error.

As another example, a servers CPU may sometimes be temporarily overloaded (resulting for instance in `ZEBRA:OVERLOAD` events, which indicate slow Zebra threads). In order to permanently fix the problem, more resources would have to be added to the system, but this may not always be possible or desirable (as an example, a weekly backup may simply transfer data as fast as it can be handled, and adding more hardware would only speed up the backup process, but not prevent events indicating that the CPU is used to full capacity).

2.3.6.2 Possible Solutions

For events, that can be (but not necessarily are) caused by ISP outages, the answer is simple: Since the connection statistics are accessible to the correlation engine, the engine can simply check, if there was an ISP outage at the time, when the event was generated. If there was an ISP outage, and the service in question is up again (since the service cannot send any events when there is no internet connection, the up event often arrives shortly after the down event, so a long delay is not necessary for the correlation), the events could be appended to the ticket for the ISP outage (if there is one – whether a ticket is generated for an internet outage is created depends on how long it lasted), or a newly ticket that is already resolved, could be created. To facilitate the handling of ISP outages, it would be interesting to allow the specification of generic acceptable outage durations and rates for different scopes, e.g. on country and host level. Specifying both a limit for the duration and the rate (e.g. at most 3 ISP outages per day, each one not longer than 5 minutes) is advisable, as a serious problem might otherwise be overlooked.

More generally, event patterns of flickering services can usually be handled with a set of conditions (which may be country, company or even host specific), which describe the required circumstances to resolve tickets automatically. For transparency, the correlation engine should clearly state in the ticket log, which conditions were required to resolve the ticket, and how they were fulfilled.

In the case of the `ZEBRA:OVERLOAD` events, the conditions could be that the events occurred in a specified time window, and that the situation was back to normal after a given time. Besides backups, the regular load caused by users can also lead to `ZEBRA:OVERLOAD` events. These events are therefore often generated during workdays. As can be seen in Figure 2.13, the CPU load is indeed quite high during workdays, with peaks on a more or less fixed time. In the case of a permanently high load caused by users, ignoring the resulting events is of course no solution. In this case, an upgrade of the responsible hardware may be necessary.

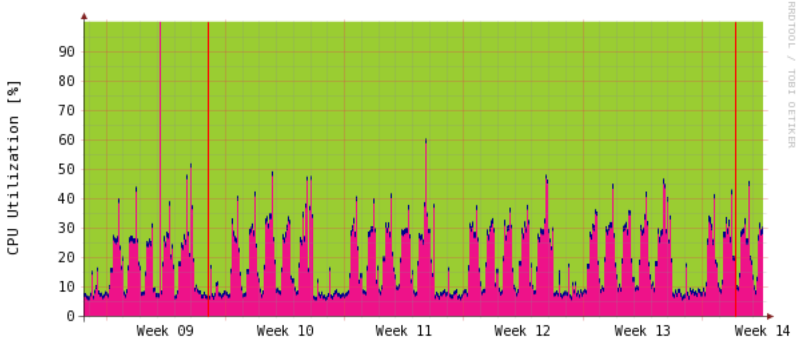


Figure 2.13: CPU utilization over the time of six weeks, on a firewall that generated `ZEBRA:OVERLOAD` events.

2.3.7 Dependencies Between Services on a Host

A single failure of a component (soft- or hardware), can result in problems with other components of the same host. This is the case, whenever there are dependencies between components and/or services. An incident can then result in problems not only with the directly affected service, but also with dependent services, leading to events from many services. This is often called an event storm.

In a simple case, there may be only two event types. The pattern would then look like this:

1. Event `X:DOWN` occurs
2. Event `Y:DOWN` occurs
3. The problem causing event `X:DOWN` is solved
4. Events `X:UP` *and* `Y:UP` occur

A more complicated case might look something like this:

1. Event `A:DOWN` occurs
2. Events `B:DOWN` and `C:PROBLEM` occur (caused by `A:DOWN`)

3. Event D:DOWN (caused by B:DOWN) and more C:PROBLEM events occur
4. The problem causing event A:DOWN is solved
5. *All* problems are solved: A:UP, B:UP, D:UP; no more C:PROBLEM events

In this example, there is more than one level of dependencies, and the problem propagates through all levels. An even worse scenario is the case, where the A:DOWN event is not even generated, i.e. the root problem is not noticed. Another complication is that it is not at all guaranteed, that all dependent services will come back automatically, once the root problem is solved.

2.3.7.1 Practical Examples

The previously mentioned HOTD:SYNC:DUPLICATEMASTER event, followed by NETWORK:IP:ADDRESS:THEFT events is a practical example for the simple pattern described above.

Another simple example is the update of anti-virus signature patterns and the accessibility of Domain Name Service (DNS) servers. If domain name resolution is not possible, the host of the signature pattern provider can not be found, and the patterns can not be updated (even if the internet connection otherwise works).

Obviously, the ISP outage itself is once again a common example as well, as most services depend on a working internet connection.

A last example is the failure of any hardware component, as the correct function of software usually depends on correctly working hardware (unless there is redundant hardware). A possible problem is a failing hard disk. Hard disks generally have a limited lifetime and occasional failures are hard to avoid, especially in an unfriendly (e.g. hot) environment[1]. A hard disk failure may be reported (e.g. with a KERNEL:IO:FAILURE event), or it may simply result in unpredictable behaviour and failure of services, without any direct notification.

2.3.7.2 Possible Solutions

In this case, the goal for the correlation engine is to identify the root cause, and forward the events in a structured fashion, e.g. as a composite event. Additionally, the correlation engine should track events from dependent services, and generate an alarm if the dependent services do not come back automatically, once the root problem is solved.

Unfortunately, simply assuming, that the first event is the root-cause, is unlikely to work, as some events may be delayed (e.g. because the events are generated by periodic checks, which are executed at different times). As an additional complication, the root-cause is not always noticed, and an event for the actual problem may not be generated (e.g. in the case of a hardware problem).

A possible solution is to specify the dependencies for each service. If a service is not working, and a dependency is not fulfilled (e.g. anti-virus signature patterns can not be updated, and no DNS server is reachable), the source of the problem is found (more exactly, one problem source – it is of course possible, that there was more than one cause). In some cases, it might also be useful, if the correlation engine actively checks, whether the dependencies are fulfilled (even though strictly seen, this is outside the scope of a correlation engine).

Hardware problems are a more difficult case. Even for a human operator, it is rather difficult to infer a hardware problem from high-level service failures. While it would be possible to specify dependencies on hardware, there is no easy way for the correlation engine to verify whether the

[1] Even in a Redundant Array of Inexpensive Disks (RAID), a simultaneous failure is possible, e.g. due to an intensive power spike. Additionally, using a RAID may simply not be cost-effective for some hosts.

hardware is working. The correlation engine might still be used to at least give hints about *possible* hardware failures (e.g. if multiple service failures, which are often related to hard disk problems, occur), but it is doubtable, whether the small probability of helpful hints justifies the effort – particularly as the correct functioning of the correlation engine itself can not be assumed in the presence of a hardware problem.

2.3.8 Events Caused by Problems on Another Host

In some cases, an event can be caused by a problem on another host.

2.3.8.1 Practical Examples

An example frequently mentioned by the interviewed Open Systems engineers is the `TMON:VPN:UP` event, which indicates that a Virtual Private Network (VPN) tunnel was established, which was previously down. On two hosts, this event is quite frequent, because these two specific hosts are VPN termination points for backup satellite links, i.e. whenever a remote host with a backup satellite link loses its primary connection, a VPN with one of these hosts is created.

Another type of event, that often falls into this category, is the `MAIL:SYNC-MAILDIR:ERROR` event. A `MAIL:SYNC-MAILDIR:ERROR` event indicates, that a host responsible for spam protection was unable to fetch email addresses from a protected mail server. This is often caused by a problem (e.g. ISP outage) at the protected mail server. As there are in some cases a lot of hosts, from which the email addresses have to be fetched, it is not surprising, that once in a while, one of them is unreachable. A single error can usually be ignored.

2.3.8.2 Possible Solutions

In the first example, the solution would be the correlation of `TMON:VPN:UP` events with the ISP outages at the remote end of the VPN connection.

More generally, it should be possible to specify a set of conditions to resolve events, similarly to handling flickering services, but including conditions (especially ISP outages) on remote hosts.

Concerning the second example: As the connectivity of the protected mail server is usually not monitored, a connectivity test would have to be made actively. As the `MAIL:SYNC-MAILDIR:-ERROR` event usually already implies connectivity problems, an alternative solution might be the same as in Section 2.3.5 – the event is ignored, if it happens only once in a row.

2.3.9 Mutual Dependencies Between Hosts

Whenever there are two hosts in a cluster, e.g. two hosts in a hot standby configuration, there are dependencies between the two hosts.

2.3.9.1 Practical Examples

As an example, consider two redundant routers and an ISP outage on the primary router. In this case, the following events will be generated (in three different tickets):

- A notification about the ISP outage on the primary router.
- A `VRRP:MONITOR:VPN:DOWN` event on the primary host, indicating that the VPN tunnel is down.
- A `KEEPALIVED:TRANSITION:SLAVE` event on the primary router, indicating that this host can no longer provide the service.

- A `KEEPALIVED:TRANSITION:MASTER` event on the secondary router, indicating that it has taken over the service.

Later, when the situation goes back to normal, the following events should be seen:

- A `VRRP:MONITOR:VPN:UP` and a `KEEPALIVED:TRANSITION:MASTER` event on the primary router.
- A `KEEPALIVED:TRANSITION:SLAVE` event on the secondary router.

2.3.9.2 Possible Solutions

Firstly, related events should end up in the same ticket. This can be done simply by checking, whether a ticket for a related host is already open, before opening a new ticket. This requires of course, that the information about relations between hosts is available to the correlation engine.

Secondly, the correlation engine should check, whether everything is fine again in the end, specifically, whether there is exactly one master and one slave after a transition. This essentially requires, that the logic of the interaction of the two hosts can be represented in the correlation engine, e.g. as a simple Finite State Machine (FSM).

Finally, if the cause for the transition is known (e.g. there was a transition and an ISP outage was detected on the primary host at the same time), the correlation engine could resolve the ticket.

2.3.10 Location Dependent Relations

As some problems are caused by environmental factors, there can be relations between events from different hosts at the same location.

2.3.10.1 Practical Examples

An `IPMI:TEMP:HIGH` event indicates that the temperature of a server is too high. If this happens on multiple hosts at the same location, there is likely a common cause (e.g. the cooling system in the server room is failing).

As another example, reboots of multiple hosts at the same location may indicate a power outage, e.g. in a whole building.

2.3.10.2 Possible Solutions

A simple solution would be to introduce a property for each event type, which specifies, whether an event can be caused by an environmental factor. If identical events are generated from different hosts at a common location, and this property is set for the corresponding event type, the events can be correlated.

2.3.11 Gathering of Additional Information

In some cases, it is useful if additional information can be gathered (both for the correlation engine and as a service to the operator, who has to handle the ticket later). Since a part of the correlation engine runs directly on the host, which generates the events, this can often be done by running an external script or by communicating with an other process.

2.3.11.1 Practical Examples

An example mentioned often by the Open Systems engineers is the `checkservices` script, which returns detailed information about the status of each service on the host it is executed on.

Another example is the `wbinfo` command, which can be run to check the connection to the Active Directory (AD) server after `WINBIND:CONF:ADCONN` events.

2.3.11.2 Caveats

It is important, not to generate new problems. External scripts should not be blindly executed whenever a given event arrives. Some events are generated at a rate of multiple events per second. Running an external script each time such an event is generated would be possibly disastrous.

In the case of `checkservices`, the same information can be gathered from the nurse process. In other cases, a rate limit for the execution of external scripts might be a good idea.

2.3.12 Correlation with Information from External Sources

Occasionally, it may be helpful to be able to take information from other sources into account.

2.3.12.1 Practical Examples

As an example, service windows can be considered. When maintenance work on a server is planned, a service window is usually scheduled. As this information is available to the correlation engine, it would be useful to correlate events with this information, such that all events generated during a service window end up in a ticket opened for this specific service window.

2.3.13 Summary

It can be seen, that a large part of the events are caused by connection problems. Especially in countries with unstable internet connections, a large number of such events is generated, often unnecessarily. A good correlation of the events with information about connection problems and ISP outages is therefore essential. Additionally, it might be interesting to correlate ISP outages also on a higher level, e.g. to detect a simultaneous ISP outage on multiple sites.

A look at the events reveals many frequent event patterns, which are suitable for automatic correlation. From a high-level perspective, the following patterns can be seen, amongst others:

- Event burst caused by a service sending repeated messages for the same problem
- Irrelevant late events, which reopen old tickets
- Irrelevant unique events, which would not require an open ticket
- “Flickering” of a service, caused by repeated temporary problems
- Event storms, if a root-problem provokes multiple issues with dependent services
- Patterns involving events from multiple hosts

Handling these patterns often requires the correlation of events with other events from the same host, as well as with information about ISP outages. In more complex cases, a correlation with events from other hosts may be required, e.g. correlation with events from hosts in the same cluster, hosts at the same physical location or hosts with the same internet link. Additionally, it is in some cases useful, if more information can be gathered on the source host, or if an external information source, such as a database, can be consulted.

Table 2.7 shows the operations, that can be used to correlate events with the identified patterns (the operations will be explained in Section 3.2). The stateless filtering operation is not in the table, because it is of no use for any of the observed patterns (which is not surprising, as we observed the patterns, after filtering had already been done). Obviously, filtering is still an important operation for any correlation engine.

Pattern	Compression	Logical Operations	Aggregation	Suppression	Masking	Thresholding	Rate Limiting	(De-)Escalation	Temporal Operations	Generalization	Specialization	Clustering
Multiple identical events	x			(x)		(x)	(x)		(x)			
Old events		x	x	(x)					x			
Late events for closed tickets		x		(x)				x	x			
Irrelevant unique events				(x)		x			(x)			
Flickering services		x	x			x	(x)		x			x
Dependencies between services		x	x	x						(x)	(x)	x
Events caused by remote problem		x	x	x	x	(x)						x
Mutual dependencies betw. hosts		x	x									x
Location dependent relations		x	x	(x)	(x)					(x)	(x)	x

Table 2.7: Correlation operations useful for correlation of the observed event patterns.

Chapter 3

Survey of Existing Event Correlation Approaches

This chapter gives an overview over existing event correlation approaches, and presents some of the available tools. As event correlation is an active research as well as a growing market, both a large number of event correlation approaches and a huge amount of products exist.

As the topic of this thesis is log alert and network event correlation, the selection of the presented approaches and products is also biased in that direction. Event correlation products for market data analysis are beyond the scope of this thesis and will not be presented.

3.1 Properties of Event Correlation Engines

Before looking at different event correlation methods, the different properties will be discussed. Please note that not all properties apply to all techniques and many techniques can be used with different properties (e.g. many techniques can be used both with expert knowledge as well as with automatically acquired knowledge). Additionally, a statement about a property does not say anything about the quality of an approach. While an approach with a given set of properties may be better suited for some application, another application might as well require the opposite properties.

3.1.1 Domain Awareness

A correlation engine can either be built for a specific domain (i.e. it "knows", what kind of information it processes), or as a general purpose correlation engine.

The advantage of a domain aware correlation engine is that it may provide special operations and data structures for that domain (e.g. a correlation engine for network events might provide a function to evaluate, whether two hosts belong to the same network). On the other hand, the separation between processing logic and domain specific information may be clearer with a general purpose correlation engine.

As with most properties, the borders are of course blurry, and even general purpose correlation engines are usually designed with a specific purpose in mind (as an example, SEC is a general purpose tool that can deal with any line based input, but was designed with log monitoring in mind [63]).

3.1.2 Self-Learning vs. External Knowledge

In order to be able to correlate events triggered by service- and network problems, a correlation engine requires knowledge, such as information about the network structure, information about the triggers for the events, or information about service dependencies.

Such information can either be gathered automatically, or manually from experts. Obviously, the second option requires a lot of work from experienced operators. This is economic only if the majority of the supplied external knowledge is static. If it is mostly dynamic, a self-learning correlation engine may be more suitable. On the other hand, automatic learning is in some cases difficult and may lead to incorrect information, if it is not done very carefully.

A compromise is to do automatic information gathering, but leave the final decision, which information to use, to an operator.

3.1.3 Real-time vs. Stored Data

Event correlation can be done either in real-time with the incoming data, or offline with stored data.

In the case of event correlation for log monitoring, real-time event correlation is needed. It should be noted however, that this only means that the events must be processed in real-time, but not necessarily, that they have to be forwarded immediately. In some cases, it may be necessary to delay events for a short time, such that events can be modified or forwarded conditionally, depending on whether another event arrives later.

Offline event analysis on the other hand may be useful to find event patterns in large amounts of data. Such pattern mining techniques are discussed in [65].

3.1.4 Stateless vs. Stateful

A real-time correlation engine can be stateful (i.e. it has a memory of the event history), or stateless (without any memory).

Obviously, a purely stateless correlation engine is very limited, as an incoming event can not be put into relation to older events. A completely stateless correlation engine is limited to filtering the events according to predefined rules (it is arguable, whether this can even be called correlation; some methods however assume several events to occur quasi-simultaneously, for instance the coding approach presented in [73] can thus be considered a form of stateless event correlation [55]). On the other hand, stateless operations are usually very simple and fast, and may be useful to handle events at the input. A typical correlation engine is therefore usually stateful, but allows both stateless and stateful operations.

3.1.5 Purely Passive vs. Active

The term event correlation implies, that there are incoming events, which are correlated depending on previous events and the internal state (cf. the FSM model presented in Section 3.3.1). Such an event correlation engine would be purely passive, i.e. it would not interact with its environment other than by receiving and generating events. In practice, it may however be desirable to gather additional information, e.g. by running an external script (a possibility provided e.g. by SEC [66]), thus allowing active behaviour to a certain degree.

3.1.6 Centralized vs. Distributed

As the events that need to be correlated are usually generated by distributed sources, it suggests itself that the correlation is also done in a distributed fashion. The obvious advantage is a better performance and scalability, and easy access to additional information from the source. On the other hand, a centralized approach is better suited to find correlation between events from different sources. Additionally, a central solution is easier to manage and requires less Operating System (OS) independency.

As a compromise, a possible solution is a system, where the events are pre-processed (e.g. filtered and compressed) on the sources, and then correlated centrally in a second step.

For any operation done on the source, it is important to avoid unnecessary feedback effects (cf. Section 3.3.1).

3.1.7 Default Policy

Similar to packet filters (such as `iptables` under Linux), a correlation engine can have different default policies for events with no matching rule, such as dropping all events, forwarding all events, or simply logging all events without further action.

As a practical example, a part of a distributed correlation engine in a network might be responsible only for low-priority devices, such as printers, and thus, a default policy to log events only might be used, with rules specifying exceptions for the few interesting cases.

3.1.8 Loss of Information

An event correlation operation is lossless, if no event or information from an event is lost during the operation. An event correlation engine is lossless, if all correlation operations are lossless.

In some cases, it may be a requirement, that a correlation engine is lossless, to allow the logging of all events, and the creation of an audit trail. On the other hand, it may sometimes be desirable to filter some events, to preserve system resources, i.e. information loss may be desirable.

From the perspective of an operator, a good trade-off is to keep all information available, but to show only the most relevant part. The details should be hidden by default, to avoid confusion, but revealed on demand ("drill-down"), to allow a closer look at the problem.

3.1.9 Transparency

Another criterion is the transparency of the correlation decisions for a human operator. If a decision can not be reproduced, the only option is to blindly trust the correlation engine, which is usually undesirable.

Transparency requires, that all operations of a correlation engine are deterministic, and the internal state as well as the input events are known. Additionally, the behaviour of the correlation engine (which may be represented by correlation rules) must be known. The correlation operations are usually deterministic and the inputs known. The internal state and the behaviour depends the chosen approach.

In the case of the rule based approached, the rules are always known, and ideally, the internal state only depends on a limited set of past input events (for example, input events older than a week should usually be irrelevant and the internal state should depend only on recent events). In this case, it is easy to reproduce the correlation decisions (provided the rules are deterministic).

In other cases, the decisions are often less obvious. Especially in the case of a self-learning correlation engine, the behaviour may depend on input events from long ago. Although the complete internal state and all other required information could always be made available to the operator, and the results are then theoretically reproducible, in practice, the question is not so much, whether it is *possible* to reproduce the correlation decisions, but rather, *how difficult* it is.

Ideally, the correlation engine should be able to explain each decision with a short message, which is added to the generated output events.

3.1.10 Robustness

If a system handles new and unknown situations well, it is said to be robust. An unknown situation may arise for instance due to noise (unknown, missing or irrelevant events), a changed network structure or incomplete information about the network.

3.1.11 Maintainability

As one of the central goals of event correlation is to make the work of the system operators easier, it is important that the event correlation itself does not require a lot of maintenance. Poor maintainability is a problem especially in cases, where a lot of expert knowledge is required, and the environment changes frequently.

3.1.12 Deep vs. Surface Knowledge

Correlation engines can further be discerned by whether they rely on knowledge gained from observation and experience only (surface knowledge), or on knowledge based on understanding the structure and functioning of a system (deep knowledge).

A classical language, that can be (and has been) used to represent deep knowledge, is Prolog (e.g. in IBM Tivoli [7], which will be briefly presented in Section 3.5.1).

3.2 Event Correlation Operations

In this section, the basic operations used for event correlation will be presented. These operations can seen as building blocks for the construction of more complex event correlation patterns.

The operations required from a correlation engine depend on the field of application. A number of different event correlation operation exists, as well as a number of terms. Unfortunately, the same term is not always used for the same operation; specifically, the terms *aggregation*, *compression*, *counting* and *duplicates removal* are often used for similar operations, with slightly varying definitions. This thesis tries to follow the most widely used definition.

3.2.1 Compression

Compression is the operation of replacing multiple identical events (i.e. identical, except for the time they were generated) by a single event. The single event should ideally contain the number of replaced events and the time of the first and last event, or alternatively the time of each event.

The term compression is used for this operation e.g. in [42, 52, 55, 65]. In some sources, the requirement, that the compressed events need to be identical, is also weakened to “similar” events.

3.2.1.1 Other terms

The term *compaction* can be used for the same operation [55], but will not be used in this thesis. The term *duplicate removal* is often used for the same or a similar operation as well (cf. e.g. [7]).

Although the compression operation is sometimes also called *aggregation* (e.g. in [72]), the term aggregation will be used in this thesis with a different meaning.

3.2.2 Logical Operations

A logical operation is a connection of events with Boolean logic (in [42], the correlation operation itself is simply called "Boolean"), such as "events `A` and `B`, but not `C`".

3.2.3 Aggregation

Aggregation is the operation of collecting (aggregating) multiple events in a single event. It can therefore be seen as a form of lossless compression [55]. As in [7], we do not require that the aggregated events are identical. Unlike compression, aggregation thus creates a new event, with a new meaning, which contains the aggregated events (rather than replacing them, as compression does).

3.2.4 Filtering (Stateless Filtering)

Filtering is the operation of removing events with given properties from an event stream. Alternatively, the default policy could also be to drop all events, and a filter would then decide which events to forward. The filtering operation does not take into account other events and is therefore stateless.

Most texts agree on the definition of filtering as a stateless operation, taking only event parameters into account [52, 55, 65]. In practice, the term is however often used in a broader sense, for the filtering of events based on any criteria, including the occurrence of other events. In [52], the term "intelligent filtering" is suggested for this use.

3.2.5 Suppression (Stateful Filtering)

Suppression (or selective suppression) is the operation of suppressing certain events depending on the context of the event correlation engine [42, 52, 65], i.e. it is the stateful counterpart to filtering. The goal of this operation is not to detect event patterns, but rather to hide or remove certain events, if a specific event pattern occurs. Suppression is thus more of a filtering operation, than a correlation operation.

3.2.5.1 Event Masking

Event masking (or topological masking) is a special case of suppression, based on topography. The goal is to hide events from nodes that are (from the view of the event sink) behind a node that already reported a problem (e.g. events from nodes behind a router). As pointed out in [51], this operation is more complex, if there are multiple paths to the event sink, or even multiple event sinks.

3.2.5.2 Examples

As an example, if a host is unreachable, and its default router failed, the `host-unreachable` event can be masked in the presence of the `router-failed` event.

3.2.6 Thresholding

Thresholding is the operation of generating an event if a certain event rate[1] threshold is exceeded (or possibly also if the event rate falls below a threshold).

3.2.6.1 Other Terms

Thresholding is sometimes also referred to as counting [65] or throttling [7]. The term throttling is however misleading, as it is often used as a synonym for rate-limiting.

3.2.7 Rate Limiting

Rate limiting is the operation of forwarding events at no more than a given event rate. This operation can be combined with compression.

3.2.8 Escalation

Escalation is the operation of increasing a parameter of an event (e.g. priority) if some given conditions apply [7, 65].

3.2.8.1 Other Terms

In [52], the term *scaling* is used for this operation. In [55], a more generic operation called *modification* is introduced, which can modify any event parameter.

3.2.9 Temporal Relationship

Temporal relationship operations correlate events based on the time and order they arrive [52,65].

3.2.9.1 Examples

Some examples are:

- Event `A` arrived at least 5 minutes, but at most 10 minutes after `B`.
- Events `A` and `B` arrived within a window of 5 minutes.
- After `A`, no event `B` arrived for at least 20 minutes.

3.2.10 Generalization

Generalization is explained in [42] as “reference to an alarm by its superclass”. In other words, the idea is to forward a more general event, rather than the specific event. Although no new information is generated, generalization can be useful to detect a common root-cause for multiple events.

3.2.10.1 Examples

As an example, `host-unreachable` events indicating that a specific host is not reachable, could be replaced by more general `isp-host-unreachable` events, indicating that an unspecified host, which uses a specific ISP, is unreachable. The `isp-host-unreachable` events can later be compressed, and a rate threshold may be used to detect an ISP outage.

[1] Please refer to Appendix A for a more detailed description of event rate measuring

3.2.11 Specialization

Specialization is the opposite of generalization. This operation replaces an event by an event of it's subclass [65].

3.2.11.1 Examples

As an example, if we receive an event indicating that a given host is down, a `service-down` event could be automatically created for each service running on that host. Although this would not generate any additional information, it might be useful to trigger specific rules to handle failing services.

3.2.12 Clustering

Clustering is defined, e.g. in [52, 65] as an operation, that generates a new event from a complex pattern, possibly combining other operations defined so far.

3.2.12.1 Example

Some examples are:

- Generate event `C`, if the rate of event `A` exceeds a threshold of five events per minute, and no event `B` has occurred for at least one hour.
- Aggregate all events that arrived within the last ten minutes into event `B`, if event `A` arrives and context `C` exists.

3.3 Event Correlation Techniques

3.3.1 Finite State Machine Based

The FSM approach to event correlation is introduced in [9]. The authors argue, that the fault identification process can be split into two steps, fault detection (i.e. noticing, that there is a problem) and fault localization (i.e. finding out, what the problem is), and that a FSM based correlation engine can help in the first step, by modeling the monitored system. The model proposed in [9] is an FSM based on the observable events generated by the monitored process, (which is assumed to be an FSM as well). If an event arrives, which leads to an invalid state, an error is reported. In [9], it is assumed, that the filter, which decides, which events are observable, is a design parameter. The authors show, that the construction of a maximum filter (i.e. a minimal set of events that allow the detection of a given problem) is an Nondeterministic Polynomial (NP)-complete problem, and thus propose a heuristic algorithm.

The selection of a filter is however of secondary interest for this thesis, as the FSM is introduced mainly as a generic model for a correlation engine. In order to allow the generation of different output events, rather than just the reporting of an error, a slightly extended model of a FSM, a Finite State Transducer (FST), will be used. In the context of event correlation, a FST can be defined as a quintuple, consisting of

- A set of possible input events I (input alphabet)
- A set of possible output events O (output alphabet)
- A set of possible states S

- An initial state $s_0 \in S$
- A state transition function: $f : I \times S \to S \times (O \cup \varepsilon)$, which defines the next state and the (possibly) generated output event for each state and input event[1]

The finite-state transducer is a nice basic model for a correlation engine, because it highlights the important factors, but ignores irrelevant aspects. For instance, in a practical setting, it is an important feature (and for commercial products a frequent sales argument), that many types of input (syslog messages, Extensible Markup Language (XML) events, . . .) and output events (such as email notifications, pager messages, syslog messages, . . .) can be handled, respectively generated – but for a formal discussion of correlation techniques, this is completely irrelevant.[2]

Furthermore, many of the seemingly more complex correlation operations can be represented in this model, simply by adding *external* event sources and sinks, and thus do not make the formal model more complex. As an example, a rate threshold can be realized with a normal finite-state transducer, if we assume, that there is an external event source, which generates timer events at the necessary resolution (e.g. one event per second)[3].

Another aspect illustrated nicely by the FSM model is the assumption, that output events do not have any effect on input effects. Although this not always strictly true in practice, it is an important assumption. While there may be indirect relations, e.g. an operator that takes actions based on output events, which has consequences on future input events, direct effects from the output on the input should generally be avoided, as it might provoke many new problems.[4]

Finally, with the FST model, it is also easy to answer the question, how to handle more than one event pattern. A FST for each pattern can be created individually, and the union of all FSTs can then be used to model the complete system (if we consider non-deterministic FSTs, which are formally equal to deterministic FSTs, building the union simply means that the new starting state is the set of all starting states of the individual FSTs).

3.3.2 Rule Based Event Correlation

One of the earliest approaches to event correlation is Rule-based Reasoning (RBR). As explained in [22], a rule based event correlation engine is organized in three levels:

- **Data level** – working memory or global database, which contains information about the problem at hand.
- **Knowledge level** – a knowledge base (rule repository), which contains domain-specific expert knowledge.
- **Control level** – an inference engine, which determines, how to apply the rules from the knowledge base to solve a given problem.

[1] ε represents the empty event, i.e. no output.

[2] In a practical program, the separation of input data acquisition, correlation and output action (e.g. by having an independent process for each task) might actually be a good idea as well, for several reasons (modularity can help to lower the complexity, and thus increase stability and security).

[3] On the other hand, such a realization of an event threshold is rather impractical, as all *possible* states would have to be included. For instance, for a sliding window of one hour, with a timing resolution of one second and a threshold of 100 events, 101^{3600} states would be needed to represent every possible event history (as every second of the past hour, between zero and 100 events could have arrived).

[4] In control theory, a positive feedback loop, which often leads to unstable behaviour, represents this problem. A nice example is a signal from microphone, that is fed through an amplifier on a speaker. If the microphone is too close to the speaker, a positive feedback loop is created, and the signal becomes unstable.

Unlike in a traditional program, control and knowledge are thus separated, and the knowledge can be extended without changing the program code of the inference engine.

The rules usually specify condition-action relations, i.e. each rule specifies a condition (e.g. "event A occurs at least ten times within five minutes, then event B occurs within no more than one minute") and a corresponding action (e.g."send an email with a warning to the operator"). As the evaluation of a rule is triggered by corresponding input events; such rules are often called Event Condition Action (ECA) rules.

The language used to represent correlation rules is a topic of active research. Approaches used in current products range from simple proprietary languages (e.g. in Simple Event Correlator (SEC); cf. Section 3.4.3), to XML based languages (e.g. in OSSIM; cf. Section 3.4.7), general purpose languages (such as Lua, which is used in Prelude; cf. Section 3.4.6) to Structured Query Language (SQL) based rule languages (e.g. in Esper; cf. Section 3.4.9).

As stated above, the evaluation of the rules requires some control logic. In a simple case, the control logic may simply evaluate one rule after the other, until the first matching rule is found (similarly to how the Linux network filter executes `iptables` rules). In more complex cases, forward chaining may be used to derive new knowledge by applying the rules from the rule repository to the existing knowledge (deductive reasoning). Practical implementations often make use of some variant of the RETE algorithm [26]. In contrary to the linear approach, the RETE algorithm is asymptotically independent of the number of rules [55].

A benefit of rule-based systems, especially when used with simple if-then style languages is the similarity to the natural language. A statement, such as "**if** event `user-login-failed` occurs 10 times within 5 minutes, **then** send an email to the operator" is perfectly understandable even to someone without computer programming experience. This also makes the decisions of a rule-based correlation engine comparatively easy to reproduce.

Unfortunately, rule-based approaches also have several drawbacks. Traditionally, the rule repository relies on the knowledge of an expert, who has domain-specific experience with the problems that are to be solved by the system, and a knowledge engineer, who knows how to represent that knowledge in the rule-system [22]. Even if the rule creation is simple enough to be done directly by the expert, the knowledge still has to be entered into the system manually, which is time-consuming. While a lot of initial work may be acceptable, frequent changes in the network also make tedious maintenance of the rule-repository necessary, which is counterproductive, as one of the stated goals for the correlation engine is to lessen the workload of operators. Another problem is the inability of rule-based systems to automatically learn from experience, meaning that the same calculations have to be made over and over again, whenever the same set of events occurs [52]. Similarly, rule-based systems also tend to fail when they are presented with new or unexpected situations [47].

3.3.3 Case Based Reasoning

In Case-based Reasoning (CBR), each problem and the corresponding solution is considered as a case. The approach of CBR to solve a given problem is to find past problems from a case library, that are similar to the problem at hand, and to try to apply a similar solution. In the end, the gained experience is stored as a new case in the library. This approach is not unlike human behaviour – if we are confronted with an unfamiliar situation, we often try to adapt a working solution from a similar problem in the past.

The CBR process can be separated into different cycles. The following separation is adapted from [1] and [58]:

1. Identify the most similar case from the case library and retrieve the solution.

2. If necessary, adapt the old solution to the current problem, to propose a new solution.
3. Apply the new solution to the problem, and verify the outcome.
4. If the solution was successful, store the new case in memory.
5. Otherwise, explain the failure and propose a better solution.

The advantage of this approach is that knowledge from past cases can be reused automatically, and that the knowledge base grows with each solved problem. Additionally, in contrast to strictly rule based systems, a CBR system can also propose solutions for previously unknown problems. As argued in [46], the fact that proposed solutions are derived from working solutions for past problems (which can be presented as evidence) may also increase the user acceptance, i.e. the users trust in the systems decisions.

On the other hand, each of the above steps can be difficult, and usually, general domain-specific knowledge (as opposed to knowledge about specific cases) is required in the cycle [1]. For instance, the retrieval of similar cases must be done carefully. As an example, in a medical diagnosis system, proposing a solution based on the fact that it worked in another case, for a patient with a similar name, height and weight is usually not a good bet, although e.g. the weight may be relevant in some cases. In [47], the proposed solution is the use of a determinator for each case in the case library, which points out the relevant attributes to determine similarity.

An even more difficult task is the adaptation of the old solution. While the manual specification of adaptation rules is a possible solution, such an approach partially defeats the advantage of CBR systems, as manual knowledge engineering is again needed. According to [46], "difficulties with case adaptation have led many CBR systems to simply dispense with adaptation, replacing the *retrieve-evaluate-adapt* cycle with *retrieve and propose* systems."

Obviously, there is some similarity between a case library and a ticketing system, which usually contains problems and corresponding solutions as well. Therefore, it suggests itself to use the ticketing system as a case library. This idea has been investigated in [47], where a CBR trouble ticketing system called CRITTER is presented. An interesting (and important) feature of CRITTER is the possibility to rate solutions, i.e. give feedback, whether they were fruitful.

Application areas for CBR systems can be found, wherever knowledge is based an past cases, such as in medical diagnosis (where a physicians diagnosis is usually based on her experience with past patients), or in law (where a judgment may be based on past cases, i.e. precedence).

3.3.3.1 Examples

Some examples from research are presented in [58], for instance the CHEF program, which is designed to develop plans for preparing dishes. This process is described as follows:

> For example, when presented with the task of creating a strawberry soufflé, CHEF resorts to modifying a vanilla soufflé recipe. However, simply adding strawberries to the standard recipe keeps the soufflé from rising properly. CHEF discovers the source of the problem in the excess liquid from the berries and decides that the best remedy is to add more whipped egg whites. This solution fixes the recipe. CHEF never repeats this mistake and can use this experience in other recipes, such as a raspberry soufflé.

A more practical example is Compaq's Support Management Automated Reasoning Technology (SMART) [2], where a CBR system is used to support the customer service.

3.3.4 Model Based Reasoning

The basic idea of Model-based Reasoning (MBR) is to represent the structure and the behaviour of the system under observation in a model, to allow the reasoning about fault causes. The task therefore is "a process of reasoning from behaviour to structure, or more precisely, from misbehavior to structural defect" [23]. As explained in [23], this requires

- a description of the structure,
- a description of the behaviour,
- and a set of guidelines to investigate misbehaviour based on these two descriptions.

As pointed out in [37], the MBR approach itself does not suggest a detailed technique. MBR is thus more of a paradigm, rather than a specific approach. According to [23], "a variety of techniques have been explored in describing behavior, including simple rules for mapping inputs to outputs, Petri nets, and unrestricted chunks of code."

Although a practical implementation might use a rule-based model, the MBR approach differs fundamentally from rule-based event correlation. In contrary to a rule-based correlation engine, which specifies event patterns as *conditions* for certain actions, an MBR system specifies a system model, with events as *consequences* of certain model states or transitions. Even though rules may be used to specify the model, MBR systems are thus closer to the FSM approach.

In a computer network, the application of MBR methods may be unsuitable, as the description of the network structure and the behaviour of each service would likely be difficult and time consuming. A more practical use case for MBR is fault diagnosis in an electrical circuit, where the network structure is specified already in the design phase as a circuit diagram, and the behaviour is – at least for a suitably limited subclass of electrical components – clearly defined by a few rules (such as Ohm's law, Kirchhoff's laws, etc.). In [23], the use of MBR for fault diagnosis in logic circuits is examined.

3.3.5 Codebook Based Event Correlation

The codebook based event correlation approach is explained in [73]. The authors propose the use of coding techniques. To allow the localization of problems, the dependencies between observable symptoms and underlying problems are examined and a suitable subset of the symptom events is selected (the codebook). The codebook must be sufficiently large to identify the problems (a codebook that is too large provides unnecessary redundancy, but a small codebook may omit information needed to distinguish between problems). For each possible problem, a binary vector is then created, which indicates, whether each symptom in the codebook can be caused by that specific problem.[1] To identify problems, the events in the codebook are monitored in real time, and whenever some events occur, the event vector is compared to the vector for each problem. The most similar vector (e.g. the problem vector with the smallest Hamming distance to the observed vector) is selected to identify the observed problem.

To illustrate the explanation, consider the following example: We have problem `A`, which causes symptoms `X`, `Y` and `Z`, problem `B`, which causes `W` and `Y`, and problem `C`, which causes `Y` and `Z`. Additionally, events can be generated or lost randomly. As `Y` is generated by all problems, it provides no information, and we select `W`, `X` and `Z` as the codebook. The correlation matrix, which consists of all problem vectors, is shown in Table 3.1.

[1] I.e. each element is either a one, if there is a relation between the given symptom and problem, or a zero if there is no relation. [55] points out, that other values could be used as well, e.g. a value indicating the probability of a relation.

	A	B	C
W	0	1	0
X	1	0	0
Z	1	0	1

Table 3.1: Correlation matrix for the codebook example – problem vectors for A, B and C.

If events W, X and Z are now observed, the problem vector with the smallest Hamming distance is A, and the problem is thus identified. In other cases, e.g. if only W and Z are observed, there is no clear decision.

As the measurement of the Hamming distance is straightforward, and only a subset of the input events has to be processed, codebook based correlation is a comparatively fast approach. Additionally, there is a good tolerance to lost events or noise. Furthermore, the problem vectors can be generated automatically from a set of training data, as explained in [73].

A significant shortcoming of the codebook approach (at least when trying to apply it to the patterns identified in Chapter 2) is the missing notion of time. When we say, a set of events occurred together, this statement contains no information about the time window applied to group the events. Additionally, there is no notion of an event order – all events in a group are assumed to occur simultaneously. Many of the patterns identified in Chapter 2, would be difficult to handle with a pure codebook approach, e.g. event bursts, or related events, which occur at different points in time. The fact that events do not have any properties is another significant problem in our case. To correlate the patterns applied in Chapter 2, a distinction between different source hosts would for instance be needed (since there are relationships between events from some hosts, we can not correlate the events from each host independently).

3.3.6 Voting Approaches

As explained in [55], the idea of the voting approach is that "each element must express its opinion on a specific topic. Then, a majority rule (absolute majority or k-majority for instance) is applied on this set of opinions (i.e. votes)."

Correlation by voting can be used to localize a fault. Usually, the votes (expressed by events from different nodes) can not give exact information about the location of a fault, but they can indicate a direction. As pointed out in [52], in this case it is necessary, that the correlation engine knows the topology of the managed network, such that the correlation engine can calculate the number of votes for each element.

An example for the use of a voting approach is given in [27]. The authors describe a scenario, where it is necessary to identify a poison message (which propagates through the network by triggering a software bug in vulnerable devices) from a number of possible message types, in a distributed network. The examined solution employs a neural network in each subnetwork. Each neural network decides, which message type is most probably the poison message and votes for that message. The message type with the most votes is then chosen as the most likely poison message type.

3.3.7 Explicit Fault-localization

In [10], the author proposes to include the information about *all* possible fault localizations with each alarm. As the authors explain, the process of fault localization is then simple:

> In the case that alarms are reliable and there is only a single fault in the network, then fault localization is straight forward: The fault lies in the intersection of the set of locations indicated by each alarm. Thus, intuitively, alarms that share a common intersection should be correlated.

The required prerequisite of having only a single fault in the whole network is of course not very practical (although it is an assumption frequently made); the authors therefore propose an extension to cover multiple faults, which is explained in [10].

Additionally, this method depends heavily on a-priory information, as the author himself explains in [10], and a method to gather this information automatically would be vital for a practical implementation.

3.3.8 Dependency Graphs

In [35], the use of dependency graphs for event correlation is examined. A dependency graph is a directed graph, which models dependencies between the managed objects. In the case of a network, the nodes represent the network elements (e.g. hosts), and an edge from node A to node B indicates, that failures in node A can cause failures in node B.

Assuming that some fault events were generated in a given time window, the goal is to find the likely root cause. As explained in [35], the basic idea is to start at the nodes that generated the initial events, and find nodes, on which many (or ideally all) of the initial nodes depend. This nodes are then interpreted as possibly responsible nodes (root-cause). As explained in [35], the length of the path between an identified responsible node and the initial nodes can be used as a metric for the quality of the correlation, as an operator has to reproduce this path to localize the fault.

As explained in [35], we (rather optimistically) assume that only one problem occurs at a time (i.e. only one root-cause; multiple events from dependent services may of course be generated). If multiple problems occur at approximately the same time, an identification of all root-causes may not be possible.

A similar approach, which takes probabilities into account, is the Bayesian network based approach, which will be discussed next. For a more elaborate discussion of dependency graph based event correlation, the interested reader is referred to [35].

3.3.9 Bayesian Network Based Event Correlation

A Bayesian network (sometimes also called a belief network) is a directed acyclic graph [45], which models the probabilistic relations between network elements, represented by random variables.

Rather than an elaborate description, Bayesian networks will be explained with an example. A more thorough explanation can be found in [4].

3.3.9.1 Example

Let's consider a case, similar to the example discussed in Section 2.3.7.1, where we receive an event from a mail server, indicating that the anti-virus signature patterns can not be updated (this event will be represented by the random variable U, which has two possible values; $U = 1$ indicates, we received this event, and $U = 0$ indicates that we did not receive such an event). This event could be caused by a problem with the infrastructure of the vendor (random variable V; $V = 1$ indicates that a problem exists), or because of an ISP outage (random variable I; $I = 1$ if there was an ISP outage). However, an ISP outage is likely to cause a event from a central monitoring process (M; $M = 1$ if such an event arrived) as well. Assuming that we know (from analysis of historical data or expert knowledge) the probability distributions of V and I (which are assumed to be independent) and the conditional probability distributions of U and M (which are assumed to be conditionally independent), the situation can be modeled as shown in Figure 3.1 with the probability distributions given in Tables 3.2 and 3.3.

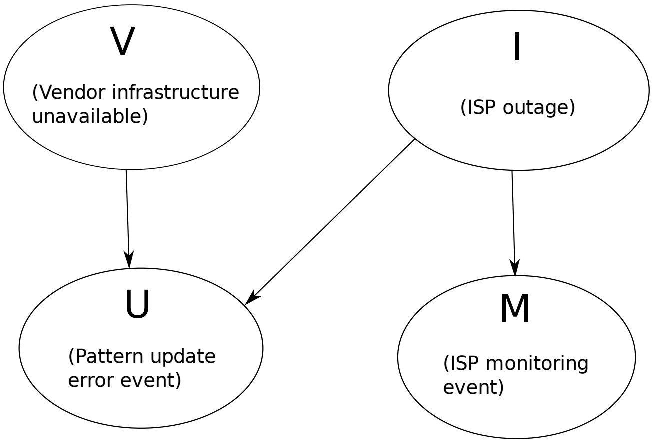

V
(Vendor infrastructure unavailable)
I
(ISP outage)
U
(Pattern update error event)
M
(ISP monitoring event)

Figure 3.1: Simple Bayes network example.

$P(V = 0)$	$P(V = 1)$
0.98	0.02

$P(I = 0)$	$P(I = 1)$
0.8	0.2

Table 3.2: Probabilities for vendor problems and an ISP outage.

I	$P(M = 0\|I)$	$(M = 1\|I)$
0	0.95	0.05
1	0.1	0.9

V	I	$P(U = 0\|V, I)$	$(U = 1\|V, I)$
0	0	0.99	0.01
0	1	0.04	0.96
1	0	0.08	0.92
1	1	0.03	0.97

Table 3.3: Conditional probability distributions for a monitoring event and a pattern update error event.

The probability distributions are completely fictional, but could be justified as follows:

- $P(M = 1|I = 0) > 0$ (i.e. there is a chance for false positives from ISP monitoring), because probe packets may get lost.
- $P(M = 0|I > 1) > 0$, because an event may get lost.
- $P(U = 1|V = 0, I = 0) > 0$, because there are reasons for a pattern update error other than an ISP outage or problems with the update servers.
- etc.

This model can now answer our probabilistic questions, e.g. what is the chance, that an ISP outage has happened, if we observe an event from ISP monitoring and a pattern update error event. The answer to this specific question is the following one:

$$
\begin{aligned}
&P(I=1|U=1,M=1)= \\
&\quad = \frac{P(U=1,M=1|I=1)P(I=1)}{P(U=1,M=1)} \qquad \text{(Bayes' theorem)} \\
&\quad = \frac{\sum_{x\in\{0,1\}} P(I=1)P(V=x)P(U=1|I=1,V=x)P(M=1|I=1)}{\sum_{x,y\in\{0,1\}} P(U=1|V=x,I=y)P(M=1|I=y)P(V=x)P(I=y)} \\
&\quad = \frac{0.2\cdot 0.9\cdot(0.98\cdot 0.96+0.02\cdot 0.97)}{\underbrace{0.98\cdot 0.8\cdot 0.05\cdot 0.01}_{(x=0,y=0)}+\underbrace{0.98\cdot 0.2\cdot 0.9\cdot 0.96}_{(x=0,y=1)}+\underbrace{0.02\cdot 0.8\cdot 0.05\cdot 0.92}_{(x=1,y=0)}+\underbrace{0.02\cdot 0.2\cdot 0.9\cdot 0.97}_{(x=1,y=1)}} \\
&\quad \approx 0.9935
\end{aligned}
$$

Unfortunately, in the general case, probabilistic inference in a Bayesian network is NP-hard, as shown in [20]. Efficient solutions for large networks are valid either only for restricted network classes, or they use approximation algorithms [4].

3.3.10 Neural Network Approaches

The idea behind Artificial Neural Networks (ANNs) is to reproduce the function of a human brain in an artificial model. As the human brain is particularly efficient at pattern recognition, the use of ANNs for tasks, such as speech and image recognition, or event correlation, suggests itself.

Although the early research with ANNs dates back to the 40's, the initially great expectations could at that time not be fulfilled. ANN methods however experienced a revival in the 80's, when faster computers and Very Large Scale Integration (VLSI) methods made practical implementations possible [48, 52].

Generally speaking, an ANN is a network of processing nodes (the equivalent of biological neurons), which perform operations on the weighted inputs to generate an output (which is again used as an input for other nodes). The computation can be a simple mathematical operation, such as a summation of all inputs, but also something more complex, such as a temporal operation [48], a threshold, or an operation involving the memory of a node [52]. To implement automatic learning, the input weights are usually dynamically adapted [48]. The procedures for adapting the weights and the selection of operations for the nodes depend on the specific application for a neural network, and many possible approaches exist.

As stated in [55], "Neural Networks seem not to be frequently applied in Alert Correlation tools." As a reason, a lack of transparency is given – "it is difficult to 'know' what is exactly done inside the ANNs." An example for a practical application is presented in [69], where a neural network is used to correlate alarms in a cellular phone network. As benefits, the author lists the simple adaptability to a changing network configuration and the resistance to noise.

3.3.11 Even More Approaches

Although the most widely used approaches have been presented in the previous sections, the list is by no means complete. Other approaches include:

- **Genetic algorithms**, which employ artificial evolution by repeatedly selecting an optimal subset from a mutating population. An example is GASSATA (Genetic Algorithm for Simplified Security Audit Trail Analysis), presented in [53]. In this case, the "population" is a set of vectors indicating different combinations of attacks, and the goal is to find the most likely combination of attacks, given a set of generated events.[1]
- **Fuzzy logic approaches**, which employ concepts from fuzzy logic, such as fuzzy sets.[2] As argued in [55], fuzzy logic concepts and non-probabilistic approaches are not necessarily competitive. As an example, a rule-based system might use fuzzy rules.
- **Constraint-based approaches**, which apply Constraint Satisfaction Problem (CSP) paradigms[3] to fault diagnosis. This approach is somewhat similar to MBR, but constraints are used to represent the system behaviour. Further explanations and an example of a practical application can be found in [56].
- **Blackboard systems**, where a central, global data repository (the blackboard) containing input data and partial solutions is iteratively updated with information from different independent knowledge sources containing the expertise to solve the problem. The knowledge sources communicate only via the blackboard. Such a system is called a blackboard system, because it resembles a setup, where different human experts work together on a problem, by adding parts of the solution to a physical blackboard. More information can be found in [21].
- **Context Free Grammars (CFGs)**, that are used to represent fault propagation patterns in a network. As argued in [59], this can be seen as an extension of the dependency graph approach.
- **Model traversing techniques**, which make use of object models to determine fault propagation. As explained in [44], "this approach uses a generic resource model in order to represent a network's heterogeneous components and their relationships with each other in a homogeneous fashion." An example for such a model is the Open Systems Interconnection (OSI) network model, which might be used to determine dependencies between objects on different network layers. Unlike fault propagation models, such as dependency graphs, model traversing techniques do not specify fault dependencies directly, but rather derive them from the model during run-time. The model is thus independent from a specific network, which makes model traversing techniques attractive for networks with frequent changes. On the other hand, this approach lacks the flexibility to deal with more complex fault propagation scenarios [44, 59].

Even more event correlation approaches can be found in [52], [55] and [59].

3.3.12 Hybrid Approaches

Obviously, many solutions exist, which combine several approaches. An interesting example is the hybrid RBR and CBR system proposed in [43]. This system makes use of both a correlation

[1] While the case of only one attack at a time could be solved in linear time, by simply evaluating the likelihood for each attack, the author is pessimistic and assumes, that more than one attack at a time may be possible, which makes the problem NP-complete [53].

[2] In a fuzzy set, each element has an assigned value ($0 \leq p \leq 1$) for the probability of its membership in the set. This is a generalisation of classical sets, where elements are either contained ($p = 1$) or not contained ($p = 0$) in the set.

[3] A CSP is a problem, where a configuration for a set of variables must be found, which fulfills a given set of constraints. A popular example of a CSP is the Sudoku puzzle.

engine with a rule base, as well as a CBR engine with a case memory. The idea is to allow an information flow in both directions, to let the correlation engine aid in the selection of cases from the case memory, and use the additional information from past cases for further correlation. For more information about this approach, the interested reader is referred to [43] and [36].

3.3.13 Summary

A lot of different approaches, and combinations of approaches exist today. Obviously, trying to argue, that one approach is generally better than an other one is futile, as this depends highly on the problem at hand.

3.3.13.1 Classification

Different schemes have been proposed to classify the approaches. A common approach is the separation between probabilistic and non-probabilistic approaches, such as in [45], where the authors separate between FSM based and probabilistic approaches. A different classification is used in [59], where the author separates fault localization techniques into Artificial Intelligence (AI) techniques (RBR, CBR, MBR, neural networks, decision trees), model traversing techniques and fault propagation models (codebook approach, Bayesian networks, dependency graphs).

3.3.13.2 Advantages and Drawbacks of Different Approaches

Table 3.4 lists some of the strengths and weaknesses of different event correlation approaches.

3.4 Existing Open Source Event Correlation Software

.

In this section, a selection of open source event correlation applications is presented. The focus lies on products targeted mainly at log and network event correlation, but even in this area, the list of presented products is by no means complete. The presented applications were selected, based on the following criteria:

- Significant use in real-world scenarios
- Maturity
- Active maintenance and development

The discussion is focused mainly on the event correlation capabilities of each product.

3.4.1 Swatch

Swatch[1] is an open source log monitoring tool written in Perl and licensed under the General Public License (GPL). Swatch is a rule based log monitoring system, which can be configured with simple rules. Each rule contains a regular expression pattern to either ignore a matching log message, or take a specified action, like printing the message on the screen, sending an email, or executing an external program. Although the authors do not describe Swatch as event correlation software, Swatch also supports simple event correlation operations, such as the specification of a rate threshold, or of a time window for rules. Swatch can be used both offline, by reading

[1] Swatch is available on SourceForge at http://sourceforge.net/projects/swatch/.

Approach	Strengths and Weaknesses	
FSMs	**Pro:**	Simple, good as a basic model, easy to understand
	Contra:	Too simple for practical applications, no tolerance to noise [73]
RBR	**Pro:**	Transparent behaviour, close to natural language, modularity [6]
	Contra:	Time-consuming maintenance, not robust [6, 73], does not learn from experience [52]
CBR	**Pro:**	Automatic learning from experience, reasoning from past experience is natural [1], can be combined with ticketing system [2]
	Contra:	Automatic solution adaptation and reuse is difficult
MBR	**Pro:**	Relies on deep knowledge
	Contra:	Description of behaviour and structure may be difficult in practice
Codebook	**Pro:**	Fast, robust, adapts to topology changes [73]
	Contra:	Reproducing the behaviour manually is tedious; no notion of time
Voting	**Pro:**	Great for use in a distributed fashion
	Contra:	Requires knowledge about topology
Explicit fault localization	**Pro:**	More efficient and extendable than rule-based approach [10]
	Contra:	Depends heavily on a-priori information [10]
Dependency graphs	**Pro:**	Good for dealing with dynamic, complex managed systems [35]
	Contra:	Assumption, that there is only one problem at a time
Bayesian networks	**Pro:**	Good theoretical foundation
	Contra:	Probabilistic inference is NP-hard [20]
ANNs	**Pro:**	Powerful for problems, that are suitable to be solved by the human brain [52]
	Contra:	Behaviour difficult to understand; requires a lot of processing power

Table 3.4: Advantages and drawbacks of the presented event correlation approaches.

log messages from a file, or in real-time, by reading log messages directly from the output of a program or by following the syslog messages.

The following listing shows a simple swatch rule, which prints the log message in bold red, and sends an email to root, if there are at least three log messages containing the string "failed login" within one minute:

```
watchfor /login failed/
threshold track_by=failed_logins, type=both, count=3, seconds=60
echo=bold,red
mail=root@localhost
```

More information about Swatch can be found in [38], as well as in the manual page of Swatch [3].

3.4.2 LogSurfer

LogSurfer[1] is a log monitoring tool based on Swatch, but written in C (which makes it more suitable for large volumes of messages). LogSurfer operates the same way as Swatch, by matching on log lines with regular expressions and executing corresponding actions, but introduces some new features. An interesting possibility is the dynamic creation (and deletion) of rules, which

[1] Available at http://www.crypt.gen.nz/logsurfer/.

allows, for instance, the grouping (aggregation) of log messages (something, which is not possible with swatch).

LogSurfer is distributed under an own open source license. More information can be found in [62].

3.4.3 SEC

Simple Event Correlator (SEC)[1] is an event correlation tool written in Perl. Similar to Swatch and LogSurfer, SEC allows the specification of rules to match line based input events (such as log messages) and execute corresponding actions. Besides regular expressions, custom Perl functions can also be used to match the input lines, or to evaluate conditions. An action can be the creation of a log message, writing the event to a file, executing an external program, etc. Additionally, SEC also allows the creation of synthetic events and dynamic contexts (internal state). A context can be used as an additional condition for rules. Together with the basic correlation operations provided by SEC, this allows the detection of composite events. The basic correlation operations are listed and described in the man page of SEC [66] as follows:

- Single – match input event and execute an action immediately.
- SingleWithScript – match input event and depending on the exit value of an external script, execute an action.
- SingleWithSuppress – match input event and execute an action immediately, but ignore following matching events for the next t seconds.
- Pair – match input event, execute an action immediately, and ignore following matching events until some other input event arrives. On the arrival of the second event execute another action.
- PairWithWindow – match input event and wait for t seconds for other input event to arrive. If that event is not observed within a given time window, execute an action. If the event arrives on time, execute another action.
- SingleWithThreshold – count matching input events during t seconds and if a given threshold is exceeded, execute an action and ignore all matching events during the rest of the time window.
- SingleWith2Thresholds – count matching input events during t_1 seconds and if a given threshold is exceeded, execute an action. Then start the counting of matching events again and if their number per t_2 seconds drops below the second threshold, execute another action.
- Suppress – suppress matching input events (used to keep the events from being matched by later rules).
- Calendar – execute an action at specific times.

Although the individual operations are rather simple, the possibility to combine rules with other rules, dynamic contexts, Perl expressions and external scripts[2] allows for complex correlation and makes SEC very versatile.

[1] SEC can be found at `http://kodu.neti.ee/~risto/sec/`.

[2] Although Swatch and LogSurfer also allow the execution of external scripts as *actions*, SEC allows the use of the return value as *input* for further correlation.

The following rules from the SEC rule set repository [67] provide an example for the application of SEC to correlate portscan events:

```
##################################################################
# Sample SEC ruleset for "PORTSCAN FROM ip1 TO ip2:port" events
##################################################################

# process "PORTSCAN FROM ip1 TO ip2:port" events, and if a certain
# source host has scanned the same destination port on more than
# 10 distinct destination hosts during 60 seconds, raise an alarm

type=Single
ptype=RegExp
pattern=PORTSCAN FROM (\S+) TO \S+:(\d+)
context=!HORIZONTAL_PORTSWEEP_FROM_SOURCE_IP_$1_TO_TARGET_PORT_$2
continue=TakeNext
desc=Horizontal port sweep started from source $1 to target port $2
action=eval %o ( $portscans{"$1:$2"} = {} ); \
       create HORIZONTAL_PORTSWEEP_FROM_SOURCE_IP_$1_TO_TARGET_PORT_$2 60 \
       eval %o ( delete $portscans{"$1:$2"} )

type=Single
ptype=RegExp
pattern=PORTSCAN FROM (\S+) TO (\S+):(\d+)
context=HORIZONTAL_PORTSWEEP_FROM_SOURCE_IP_$1_TO_TARGET_PORT_$3
continue=TakeNext
desc=Scanned destination IP: $2
action=eval %o ( $portscans{"$1:$3"}->{$2} = 1 ); \
       add HORIZONTAL_PORTSWEEP_FROM_SOURCE_IP_$1_TO_TARGET_PORT_$3 %t: %s;\
       set HORIZONTAL_PORTSWEEP_FROM_SOURCE_IP_$1_TO_TARGET_PORT_$3 60 \
       eval %o ( delete $portscans{"$1:$3"} )

type=Single
ptype=RegExp
pattern=PORTSCAN FROM (\S+) TO (\S+):(\d+)
context=HORIZONTAL_PORTSWEEP_FROM_SOURCE_IP_$1_TO_TARGET_PORT_$3 \
        && =( scalar(keys(%{$portscans{"$1:$3"}})) > 10 )
continue=DontCont
desc=$1 has scanned more than 10 destinations
action=report HORIZONTAL_PORTSWEEP_FROM_SOURCE_IP_$1_TO_TARGET_PORT_$3 \
       mail -s 'Horizontal port sweep from $1 target port $3' root@localhost; \
       delete HORIZONTAL_PORTSWEEP_FROM_SOURCE_IP_$1_TO_TARGET_PORT_$3; \
       eval %o ( delete $portscans{"$1:$3"} )
```

The goal is to detect portscans from a single host to the same port on more than 10 different target hosts. To achieve this, a context for each new combination of source host and destination port is created, to which further events for the same combination are added. If the threshold of 10 target hosts is exceeded, a report is sent via mail. An interesting feature of SEC, which can also be seen in this example, is the possibility to use parts of the match in a human readable description (i.e. the use of variables in the value for `desc`).

Despite – or because – its simplicity, SEC has a respectable user base and is used successfully even in large networks (an overview of SEC users can be found in [65]). More information about SEC can be found in [64], or in the SEC man page [66], which is very well written.

3.4.4 OSSEC

OSSEC[1] is an open source Host-based Intrusion Detection System (HIDS), consisting of a core application, an agent for Windows systems, and a web based UI. According to the OSSEC website [14], the key features are file integrity checking, log monitoring, rootkit detection and active response.[2] OSSEC supports a large number of operating systems and can analyze logs from various devices and applications, such as Cisco routers, Microsoft exchange servers, OpenSSH or NMAP[3]. Among other options, possibilities for output include logging to syslog, storing events in a database, sending email, generating reports, and of course the access via the web UI.

Although much more could be said about OSSEC, the further discussion will focus on the part responsible for log analysis and correlation, and the interested reader is referred to the OSSEC website [14] for more generic information.

Correlation and analysis is implemented in the analysis daemon. The analysis daemon (`analysisd`) is part of the core application, which is written in C. Correlation in OSSEC is rule-based, using XML rules (although version 2.0, released in February 2009, also allows rules to be written directly in C).

In a first step, log messages are decoded, and information, such as IP addresses, user names, timestamps, etc. is extracted and stored semantically (this is done according to decoders specified in XML – please refer to [11] for more information). Next, XML rules allow the correlation based on this information, as well as additional parameters, such as the frequency of a certain event, the occurrence of other events or additional pattern matches. Each rule has a unique id, a level (a number between 0 and 15, which indicates the priority of a match) and conditions for matching. The action depends on the level — on level 0, no alerting is done at all, and on higher levels, the message is logged, or event sent to an administrator via email (depending on the global configuration).

The following listing shows an abbreviated version of the Pluggable Authentication Modules (PAM) rule group, which is included with the OSSEC source code:[4]

```
<group name="pam,syslog,">
  <rule id="5500" level="0" noalert="1">
    <decoded_as>pam</decoded_as>
    <description>Grouping of the pam_unix rules.</description>
  </rule>

  <!-- ... -->

  <rule id="5503" level="5">
    <if_sid>5500</if_sid>
    <match>authentication failure; logname=</match>
    <description>User login failed.</description>
    <group>authentication_failed,</group>
  </rule>

  <!-- ... -->

  <rule id="5551" level="10" frequency="6" timeframe="180">
    <if_matched_sid>5503</if_matched_sid>
    <same_source_ip />
    <description>Multiple failed logins in a small period of time.</description>
    <group>authentication_failures,</group>
```

[1] Available at http://ossec.net.

[2] I.e., OSSEC also has some Intrusion Prevention System (IPS) capabilities.

[3] A comprehensive list is available at http://www.ossec.net/main/supported-systems.

[4] The full rule group can be found in the file `etc/rules/pam_rules.xml` in the OSSEC sources [13] and is licensed under the GPL.

```
  </rule>
</group>
```

As can be seen, the first rule, 5500, matches any PAM log messages, but since it has level 0, no alerting is done. The next rule 5503 is conditional on rule 5500, i.e. it can only match, if rule 5500 has matched. The advantage of such a nesting is that only the first rule has to be executed for non-PAM messages. This concept is somewhat similar to the Rete algorithm, albeit the rule tree is created manually. According to [11], with this approach, only about $\frac{7}{8}$ of the rules have to be executed for an average log message. The third rule, 5551, performs an escalation of events that matched rule 5503:[1] If there are at least 6 failed authentications within 180 seconds, triggered by the same source IP address, then more aggressive alerting is done.

Other correlation operations include matching on messages that were generated during a specified time interval (e.g. to detect logins outside business hours), or the suppression of duplicate events to avoid floods. Although OSSEC as a whole is an impressive tool, the correlation operations possible with XML rules are however rather basic. More intricate correlation is possible with rules written directly in C. While C allows for rules of arbitrary complexity, the drawback is however, that the creation of a new rule is more difficult and time-consuming, and a recompilation is required for each new rule.

More information about the correlation operations of OSSEC can be found in Chapter 11 ("General configuration options") of the OSSEC manual[2], as well as in [11].

3.4.5 OpenNMS

OpenNMS[3] is an open source network monitoring platform written in Java. According to the OpenNMS website [34], the main focuses of OpenNMS are service polling, data collection and event and notification management.

Data collection can be done via various protocols, such as HTTP, Simple Network Management Protocol (SNMP) or Java Management Extensions (JMX), and event sources include SNMP traps, syslog messages or internal events. Notifications can be done via email, Short Message Service (SMS), Extensible Messaging and Presence Protocol (XMPP), or any external program. Additionally, the data can be viewed in the web based Graphical User Interface (GUI).

OpenNMS aims to implement different correlation strategies. In the current version of OpenNMS (version 1.6.4), correlation is based on the Drools engine, which will be discussed in Section 3.4.8, and is therefore not explained further here.

3.4.6 Prelude

PreludeIDS[4] is an universal SIM application written in C. The components of Prelude provide, among other things, log analysis, a correlation engine, event management, database access and a web based UI written in Python. Prelude does not provide an agent, but rather gathers input data from other security tools. Native compatibility is provided for a number of programs (such as OSSEC, Snort, PAM, Nepenthes), and log messages from various other devices can be handled by plugins of the Prelude Log Monitoring Lackey (LML).[5] Collected messages are normalized

[1] The conditions `if_sid` and `if_matched_sid` are equivalent, according to the website of OSSEC [14].

[2] Available on the website of OSSEC [14], under `http://www.ossec.net/main/manual`.

[3] OpenNMS is available at `http://opennms.org`. Commercial support is available from `http://opennms.com`.

[4] The website of Prelude can be found under `http://www.prelude-ids.com/`. The source and documentation is available from `https://trac.prelude-ids.org/`.

[5] More information about supported log formats can be found under `http://www.prelude-ids.com/en/development/documentation/compatibility/index.html`. The website claims, that Prelude is able to interoperate with *all* systems available on the market.

and represented in the standardized Intrusion Detection Message Exchange Format (IDMEF).[1] In a next step, Prelude filters, classifies and correlates the data. The results can be visualized, and reports or alerts can be generated in various formats.

Correlation in Prelude is rule based, with correlation rules written in Lua.[2] The following listing from the Prelude Correlator sources [18] shows a Prelude rule to detect events outside business hours:[3]

```
function business_hour(INPUT)

local t = INPUT:get("alert.create_time")
local is_succeeded = INPUT:match("alert.assessment.impact.completion","succeeded")

-- Run this code only on saturday (1) and sunday (6), or from 6:00pm to 9:00am.
if is_succeeded and (t.wday == 1 or t.wday == 6 or t.hour < 9 or t.hour > 18) then
        local ca = IDMEF.new()

        ca:set("alert.source", INPUT:getraw("alert.source"))
        ca:set("alert.target", INPUT:getraw("alert.target"))
        ca:set("alert.classification", INPUT:getraw("alert.classification"))
        ca:set("alert.correlation_alert.alertident(>>).alertident",
                        INPUT:getraw("alert.messageid"))
        ca:set("alert.correlation_alert.alertident(-1).analyzerid",
                        INPUT:getAnalyzerid())
        ca:set("alert.correlation_alert.name",
                        "Critical system activity on day off")
        ca:alert()
end

end
```

The function parameter `INPUT` contains an IDMEF event. In this example, the rule generates an alert (the `ca` object, which is also an IDMEF event), if an event is detected, which indicates a successful action outside business hours. For more complex rules, a context class allows the creation of dynamic contexts, and makes operations, such as thresholds and timeouts possible.[4]

For the sake of brevity, many features of Prelude have been omitted. More information about Prelude is available on it's website [61] and in the documentation [19].

3.4.7 OSSIM

OSSIM[5] is, as the name implies, open source SIM software, aiming at the integration of various open source software components (such as Nmap, Nessus, Snort, Nagios, OSSEC and others) into a comprehensive security software suite. As explained on the OSSIM website, the developers rely on existing open source products, and add a number of additional tools, "the most important being a generic correlation engine with logical directive support" [16]. The core of OSSIM, which is responsible for event collection, management and correlation, as well as for risk assessment and alerting, is written in C. Other tools provided by OSSIM include an agent (written in Python), which can be used to collect information from hosts, a PHP based web GUI, and a Python

[1] IDMEF is specified in RFC 4765 [24].

[2] Lua is a general purpose scripting language, designed to be embeddable into other programs. More information can be found at `http://www.lua.org/`.

[3] The listing can be found in the file `plugins/lua/ruleset/business-hour.lua` in the Prelude Correlator sources [18], and is licensed under the GPL.

[4] Explanations and an example for a rule, which makes use of this class, can be found under `https://dev.prelude-ids.com/wiki/prelude/PreludeCorrelator`.

[5] OSSIM is available at `http://www.ossim.net`. Commercial support is available from `http://www.ossim.com`.

daemon, `frameworkd`, which is responsible for maintenance work and for controlling other parts of OSSIM. For data storage, OSSIM relies on a SQL database.

One of the central goals of OSSIM is to reduce the number of false positives. To this end, OSSIM introduces a concept of risk, based on the three values priority, reliability and asset. Priority is a value between 0 and 5, which specifies the seriousness of an event, i.e. how harmful a successful attack would be. Reliability (a value between 0 and 10) on the other hand indicates the probability of a successful attack. Correlation can be used to modify the reliability of matching events. Together with an asset value, a number between 0 and 5, which indicates the (e.g. financial) value of the assets associated with the event, two risk values for an event can be calculated, a compromise and an attack risk value.[1] The risk value is the scaled product of reliability, priority and asset, and an alarm is generated if the risk is larger or equal to one, as can be seen in the following code fragment from the file `sim-organizer.c` in the OSSIM source code [17]:

```
//check if the source could be an alarm. This is our (errr) "famous" formula!
event->risk_c =
      ((double) (event->priority * event->asset_src * event->reliability)) / 25;
if (event->risk_c < 0)
  event->risk_c = 0;
else
if (event->risk_c > 10)
  event->risk_c = 10;

if (event->risk_c >= 1)
  event->alarm = TRUE;
```

An analogous calculation is done for the destination host.

From a high-level perspective, OSSIM provides three different types of correlation, which the OSSIM documentation [15] lists as follows:

- **Cross Correlation**, between events and destination vulnerabilities
- **Inventory Correlation**, between events and destination characteristics
- **Logical Correlation**, between events from different sources

Cross correlation relies on information about vulnerabilities on the destination host, which were gathered with Nessus. If an attack is detected (e.g. by Snort) against a host, which is known to be vulnerable to that specific attack, the reliability is changed to 10 (i.e. it is assumed that the attack was successful with 100% certainty).

Inventory correlation on the other hand relies on generic host information, such as OS, port, application, protocol and version information. For instance, if the attacked port is closed on the destination host, the reliability is changed to zero.[2] As another example, if the OS on the destination host is known to be a possible target for a given attack, the reliability is increased.

The last type is logical correlation, which relies on the backlog[3] to correlate different events. Logical correlation is generic, rule-based correlation, and unlike the name might seem to imply, also includes non-boolean operations, such as time-outs. For logical correlation, OSSIM makes use of XML directives, which are similar to rule groups in OSSEC, and can contain multiple,

[1] In short, the compromise value indicates the risk of a compromised destination host, and the attack value indicates the risk of an attack by the source host. Please refer to the OSSIM documentation [15] for more information.

[2] An application can not be exploited without successfully making a connection. As explained in [15], it might still be possible to attack the network stack.

[3] The backlog contains memory about matched directives and corresponding events, i.e. it can be seen as the internal state of the correlation engine.

possibly nested rules. Each new event is matched against all directives, and can thus also generate multiple alarms. As explained in the documentation of OSSIM [15], directives can create new events of a special type (“directive events”), which will, depending on priority and reliability, later result in alerts.

The following listing from the OSSIM sources [17] shows a directive to detect SSH brute force login attempts, with escalation based on the rate of failed logins:[1]

```
<directive id="20" name="Possible SSH brute force login attempt against DST_IP"
  priority="5">
  <rule type="detector" name="SSH Authentication failure" reliability="3"
    occurrence="1" from="ANY" to="ANY" port_from="ANY" port_to="ANY"
    time_out="10" plugin_id="4003" plugin_sid="1,2,3,4,5,6">
      <rules>
        <rule type="detector" name="SSH Authentication failure (3 times)"
          reliability="+1" occurrence="3" from="1:SRC_IP" to="ANY"
          port_from="ANY" time_out="15" port_to="ANY"
          plugin_id="4003" plugin_sid="1,2,3,4,5,6" sticky="true">
          <rules>
            <rule type="detector" name="SSH Authentication failure (5 times)"
              reliability="+2" occurrence="5" from="1:SRC_IP" to="ANY"
              port_from="ANY" time_out="20" port_to="ANY"
              plugin_id="4003" plugin_sid="1,2,3,4,5,6" sticky="true">
              <rules>
                <rule type="detector" name="SSH Authentication failure (10 times)"
                  reliability="+2" occurrence="10" from="1:SRC_IP" to="ANY"
                  port_from="ANY" time_out="30" port_to="ANY"
                  plugin_id="4003" plugin_sid="1,2,3,4,5,6" sticky="true">
                </rule>
              </rules>
            </rule>
          </rules>
        </rule>
      </rules>
    </rule>
</directive>
```

With the information, that `plugin_id="4003"` specifies that SSHd messages should be matched, and `plugin_sid="1,2,3,4,5,6"` indicates messages, that belong to unsuccessful logins, the directive should be mostly self-explanatory.

More information can be found on the OSSIM website [16], which also provides comprehensive documentation.

3.4.8 Drools

Drools[2] is an open source rules engine and management system, written in Java. The Drools website describes the project as follows [12]:

> Drools is a Business Rule Management System (BRMS) and an enhanced Rules Engine implementation, ReteOO, based on Charles Forgy’s Rete algorithm tailored for the JVM. More importantly, Drools provides for Declarative Programming and is flexible enough to match the semantics of your problem domain with Domain Specific Languages, graphical editing tools, web based tools and developer productivity tools.

[1] The listing can be found in the file `etc/server/generic.xml` in the OSSIM sources [17].

[2] Available at `http://www.jboss.org/drools/`.

As explained in [54], the use of a rules engine is suitable for an application, which involves complex Boolean logic. In such cases, a rules engine can make the application much more maintainable, as it allows the separation of logic from the source code.

Rules in Drools are specified in the Drools Rule Language (DRL), which uses the following format:

```
rule ”<unique name>”
        <attributes>
when
        <conditions>
then
        <actions>
end
```

The rules thus specify condition-action relations, as discussed in Section 3.3.2. With optional attributes, the behaviour of the rule can be influenced. An important attribute is the *salience*, a number which allows to influence the order, in which rules are executed (rules with the highest *salience* are executed first). An example for the use of business rules with Drools can be found in [54].

Besides this “native” language, Drools also allows the use of Domain Specific Languages (DSLs) to make rules more understandable. Furthermore, Drools comes with an Eclipse based rule IDE, which simplifies the textual or graphical specification of DRL and DSL rules.

More information about Drools is available on it’s website [12]. The website also provides a detailed documentation of Drools, including some examples.

3.4.9 Esper

Esper[1] is an open source component for building real-time ESP and CEP applications in Java (additionally, NEsper, written in C#, can be used with .NET). Although Esper is not primarily targeted at network event correlation, it is a CEP and ESP toolkit certainly worth mentioning.

Esper is not domain-specific and supports a wide variety of correlation operations, both for CEP and ESP. Examples include logical and temporal operations, filtering, averaging, aggregation, rate limiting, thresholding, sorting or merging. According to its website [28], “typical application areas are business process management and automation, finance, network and application monitoring and sensor network applications.”

Event data can be processed with statements in an SQL-like Event Processing Language (EPL)[2], with events represented by Java Beans, Plain Old Java Objects (POJOs), Maps[3] or XML objects. Custom actions can be written as POJOs, which are triggered, when a corresponding condition matches [5]. The FAQ of Esper, available on it’s website [28], describes the function of Esper as follows:

> The Esper engine works a bit like a database turned upside-down. Instead of storing the data and running queries against stored data, the Esper engine allows applications to store queries and run the data through.

In [5], the following example is given for the use of such a query:

> Assume a trader wants to buy Google stock as soon as the price goes below some floor value – not when looking at each tick but when the computation is done over a sliding time window – say of 30 seconds. Given a StockTick event bean with a price

[1] Available at `http://esper.codehaus.org`.

[2] In the context of Esper, the EPL is sometimes also called Event Query Language (EQL).

[3] I.e. `java.util.Map` objects, which map keys to values.

> and symbol property and the EQL *"select avg(price) from StockTick.win:time(30 sec) where symbol='GOOG'"*, a listener POJO would get notified as ticks come in to trigger the buy order.

Given a set of EQL statements (which can also be added and removed dynamically, while the engine is running), Esper can decide itself, which events need to be kept in memory, and will only keep the minimum number of events that is required (e.g. for a sliding window) [28].

More information can be found on the website of Esper [28], which also provides comprehensive documentation.

3.4.10 Many Other Applications

As mentioned before, the list of presented applications is by no means complete. A search on SourceForge or similar sites reveals many more products, such as (the product descriptions are taken from the respective websites):

- RuleCore[1] – "RuleCore is an event-driven reactive (ECA style) rule engine with GUI tools, all written in Python. RuleCore triggers actions by detecting complex patterns of events." (Although RuleCore is an ambitious and interesting project, the open source version is no longer under active development, and the project is now commercial [65].)
- OpenSIMS[2] – "We've integrated Nmap, Snort, Nagios, and Nessus into a common event correlation framework. This means you can take events from your existing open source network tools."

Even more tools, both commercial and open source, are described in [55]. Furthermore, many ambitious projects for rules engines can be found, e.g. many rules engines written in Java.[3]

3.5 Commercial Event Correlation Products

Just like in the open source world, there is also an abundance of commercial event correlation products available. For the sake of brevity, only a few selected products will be presented here.

3.5.1 IBM Tivoli Enterprise Console

Tivoli Enterprise Console (TEC)[4] is the part of IBM's comprehensive Tivoli Management Framework (TMF), which is responsible for event management and correlation. TEC is a general purpose application, which can be used to correlate business, system or network data. While TEC is commercial software, some information is freely available from the product manuals [41], which are openly accessibly at IBM's website.

For network management, TEC can be integrated with IBM NetView[5] (additionally, NetView also provides some event filtering and correlation capabilities itself). The following discussion will however focus on TEC.

In TEC, correlation is rule based, using a high-level language, which is later translated to Prolog, as explained in the manual [41]:

[1] http://sourceforge.net/projects/rulecore/

[2] http://opensims.sourceforge.net/

[3] Some of them are listed at http://java-source.net/open-source/rule-engines.

[4] The product web site can be found at http://www-01.ibm.com/software/tivoli/products/enterprise-console/.

[5] NetView can be found at http://www-01.ibm.com/software/tivoli/products/netview/.

> Rules are written in a high-level language called the rule language. The rule language provides a simplified interface to the Prolog programming language, which is the language actually used internally by the rule engine. Your rules in the rule language are precompiled into Prolog source code, which is then compiled into Prolog executable files.

The generic structure of a rule is as follows [41]:

```
rule_type: rule_name:
(
    description: 'rule description',
    event: event filter,
    action: action1,
    action: action2,
    ...
).
```

This ECA-format allows filtering or duplicate removal, but also more complex correlation operations, such as temporal and context dependent operations. Additionally, Prolog can also be used directly in TEC rules [41].

According to [7,8], TEC was "one of the first systems that introduced 'deep' event correlation technologies", and is further noteworthy for

- allowing self-modifiable events,
- using a distributed filtering approach,
- using a standard programming language (Prolog), rather than an ad-hoc solution.

On the other hand, [8] also argues, that the complexity introduced by Prolog was part of the reason, why TEC was essentially too difficult to use. According to [8], TEC will be discontinued in 2012, in favor of IBM Tivoli Netcool/OMNIbus, which uses a different correlation approach.

As TEC is a very comprehensive product, a detailed discussion would be beyond the scope of this thesis. More information can however be found in [41], [6] and [8].

3.5.2 HP Event Correlation Services

HP Event Correlation Services (ECS)[1] is an event correlation product, which is part of HP's OpenView software suite. For network monitoring, HP ECS is integrated with HP's Network Node Manager (NNM) (which has, on the other hand, also been used with SEC [65]). The main part of HP ECS is the ECS Engine, which, according to the product's data sheet [39], "transforms and processes event streams according to the installed correlations".

Even though HP ECS can be seen as a rule-based system [52, 65], it does not rely on ECA rules, but instead uses rules to control the event flow. HP provides a graphical rule editor, called ECS Designer, which allows the visual design of rules, so called "correlation circuits". These circuits represent the event flow in a flow graph, as a network of processing nodes [7]. Nodes can be used for simple operations, such as filtering the event stream; more complex nodes can be created as a combination of multiple primitive nodes [57].

Internally, an EPL, which is simply called Event Correlation Description Language (ECDL), is used for rule specification. This language can however not be used directly, as explained in an article in the HP Journal [57]:

[1] The product website can be found under http://www.openview.hp.com/products/ecs/.

> The ECS Designer ensures that the circuit designer does not need to understand this language in great detail. The circuit is specified by the visual interconnection of selected nodes. Node parameters are specified wherever possible using simple ECDL constructs and supplied library functions written using ECDL or actually built into ECDL. Advanced users are able to create specialized reusable functions. The ECDL code produced by the ECS Designer is encrypted in source form and compiled for downloading to the correlation engine. Direct coding using ECDL is not supported and cannot be compiled.

More information about HP ECS can be found in [57] and in [52].

3.5.3 Many Other Applications

Many more products are available, some also as appliances. Information is however often difficult to obtain. The following list is intended as a collection of pointers, for further investigation (the product descriptions are taken from the respective websites).

- TriGeo Security Information Management (SIM)[1] – "an award-winning product that combines real-time log management, event correlation and endpoint security with a unique active response technology."
- Tenable Log Correlation Engine[2] – "aggregates, normalizes, correlates and analyzes event log data from the myriad of devices within your infrastructure."
- ArcSight SIEM Platform[3] – "an integrated set of products for collecting, analyzing, and managing enterprise event information."
- Symantec Security Information Manager[4] – "can collect and normalize a broad scope of event data and correlate the impact of incidents based on the criticality to business operations or level of compliance to various mandates."

More applications are discussed in [52] and [55].

3.6 Comparison of Existing Event Correlation Software

Tables 3.5 and 3.6 provide an overview of the features of the discussed event correlation applications and toolkits. Rather than trying to list all capabilities, Table 3.6 is however meant to point out the most noteworthy features only.

[1] http://www.trigeo.com/products/

[2] http://www.tenablesecurity.com/products/lce/

[3] http://www.arcsight.com/products/

[4] http://www.symantec.com/business/security-information-manager

Software	Domain	Language	UI	License	Homepage
Swatch	Log monitoring	Perl	CLI	GPL	`http://swatch.sourceforge.net`
LogSurfer	Log monitoring	C	CLI	(Open Source)[1]	`http://www.crypt.gen.nz/logsurfer/`
SEC	General purpose	Perl	CLI	GPL	`http://kodu.neti.ee/~risto/sec/`
OSSEC	Log based IDS	C, PHP	WebUI	GPL	`http://ossec.net`
OpenNMS	Network Monitoring	Java	WebUI	GPL	`http://opennms.org`
Prelude	SIM	C, Python	WebUI	GPL	`http://www.prelude-ids.com`
OSSIM	SIM	C, Python, PHP	WebUI	GPL[2]	`http://ossim.net`
Drools	Rules engine	Java	API	AL	`http://www.jboss.org/drools/`
(N)Esper	CEP, ESP	Java (C#)	API	GPL	`http://esper.codehaus.org`
IBM TEC	General purpose	Java	GUI, WebUI	(Commercial)	`http://www-01.ibm.com/software/tivoli/`
HP ECS	General purpose	?	GUI	(Commercial)	`http://openview.hp.com/products/ecs/`

Table 3.5: Overview of event correlation software.

Software	Input Formats	Event Correlation Approach	Capabilities and Strengths
Swatch	Line based	Rule based (plain-text rules)	Advanced filtering of log messages
LogSurfer	Line based	Rule based (plain-text rules)	Advanced filtering and grouping of log messages
SEC	Line based	Rule based (plain-text rules)	Complex correlation of line-based input based on simple operations
OSSEC	Line based	Rule based (using XML and C)	Filtering, thresholding, temporal operations, escalation, . . .
OpenNMS	Various	Rule based (using Drools)	Event correlation based on Drools
Prelude	Various	Rule based (using Lua)	Standardized event format (IDMEF), general purpose EPL (Lua)
OSSIM	Various	Vulnerability, inventory, rule based	Automatic correlation of events with gathered vulnerability data
Drools	Various[3]	Rule based (DRL, as well as DSLs)	Rete based, allows the creation of DSLs, Eclipse based GUI
(N)Esper	Various[3]	Rule based (SQL-style rules)	SQL-like statements for high-speed ESP and CEP
IBM TEC	Various[3]	Prolog based rules engine	Deep correlation, powerful general purpose language (Prolog)
HP ECS	Various[3]	Network of processing nodes	Graphical design of event flow circuits with processing nodes

Table 3.6: Event correlation capabilities of existing software.

[1] LogSurfer uses it's own license, which allows redistribution as source code or in binary form.

[2] As of March 13th 2009, OSSIM switched from the Berkeley Software Distribution (BSD) license to GPL.

[3] Usually integrated with other products, which deliver input data.

Chapter 4

Specification

This chapter discusses the specification of our Event Correlation Engine (ECE). Although the main purpose of the presented ECE is to deal with event patterns, such as those identified in Section 2.3, and the ECE is targeted mainly at the correlation of log messages and network events, we try to keep the design as versatile as possible, without making the configuration too complex. The goal is to find a good trade-off between flexibility and ease of use.

Various of the applications discussed in Section 3.4 influenced the design, notably SEC, OSSIM and OSSEC, but also other tools, such as ruleCore and Esper. Additionally, suggestions and comments from Open Systems engineers provided valuable input. Furthermore, the available events and the identified patterns served as an additional guideline, to decide, which correlation operations and elements are required.

From a developers perspective, this chapter answers the question, *what* the correlation engine should do, whereas Chapter 5 will explain, *how* it is done.[1]

4.1 Requirements and Assumptions

Before specifying a suitable event correlation engine, it makes sense to take a look at the requirements again. From a high-level perspective, the main goals for the correlation engine are to keep the number of false positives low without generating any additional false negatives, and to help the operator find the root cause for a problem faster. To achieve this, the correlation engine should correlate incoming events and detect known patterns, as identified in Section 2.3.

For a specification, more detailed requirements are necessary. The following requirements are specified in the thesis assignment (cf. Appendix D):

- Quasi real-time processing of incoming events
- Flexible configuration, with the possibility to extend the engine for future constellations in an uncomplicated way, and without reprogramming the whole engine
- Configuration language, which allows to cover all arising event patterns, and which supports different types of alerting
- Efficiency, scalability, low resource consumption

[1] From a users perspective, the question, *what* needs to be correlated was already answered in Chapter 2, and this chapter answers, *how* the events can be correlated – alas, there is some ambiguity.

- Ability to take additional information sources into account (besides the events themselves)
- Distributed architecture with correlation in two steps, on the source host and at a central location

From the considerations in Chapters 2 and 3, as well as from feedback by Open Systems engineers, the following additional requirements can be identified:

- Reproducibility: The behaviour of the ECE should be deterministic and depend solely on the input events, the rules and the internal state (which again should depend only on past events and the rules). If the correlation engine is run multiple times with the same input data, the result should be the same each time.[1] As it is also possible to make use of external information sources, reproducibility is however not fully under the control of the correlation engine. Thus, if external information sources are used (which may e.g. be time dependent), the results are not necessarily reproducible.
- Traceability: The decisions of the ECE should be understandable and traceable (and the correlation engine should support the traceability with descriptions and references to rules, for each change to an event).
- Since there are a lot of hosts, it is not only important that the rules can be created easily, but also their handling should be simple (e.g., it should be possible to specify a rule, that will be executed on various hosts, without having to specify a rule for each host individually).
- Generalized distributed architecture: Rather than making a separation between agent and central correlation engine, it would be nice to be able to use the same application everywhere. Rather than just allowing two correlation steps, this would make it possible to do correlation in any number of steps, as long as the network of processing nodes forms a tree, where the events flow from the leaves towards the root node.

On the other hand, several criteria are explicitly and intentionally ignored, such as the following ones:

- Integrity and authenticity: It is assumed, that the event sources and the rule repository are trustworthy, and this is not verified in any way.
- Communication confidentiality: There is no protection against eavesdropping.
- Input formats: Although it should be possible to add new input data formats easily, it is not the goal to support as many input formats as possible by default.
- Rule repository: As a proof-of-concept, some rules for the identified patterns will be specified. It is however not the goal to provide a large set of rules along with this thesis, as this requires a lot of both time and experience.

[1] Assuming, there are no dependencies on the absolute time, and no external factors beyond the control of the correlation engine. Furthermore, there should also be no dependencies on the current CPU usage, e.g. because the order of processed events depends on race conditions between input events and internally generated events (e.g. by timeouts). While this is hard to avoid in practice, it could be avoided in simulation, by either making all input events available at the start, or by having the correlation engine and input event generation share a common time base, which is independent of the real time.

The interpretation of the fact, that these criteria will be ignored for the moment should however not be, that they are not important[1], but rather, that these qualities should be asserted by another layer, which is not part of the correlation engine (e.g., integrity of the communication should be guaranteed by TCP, and confidentiality could be achieved by using Transport Layer Security (TLS), or a VPN). Furthermore, some parts (like the rule repository) simply require more work, beyond the scope of this thesis.

4.1.1 A Case for a Rule Based Correlation Engine

Even though many of the approaches discussed in Section 3.3 have advantages over a rule based engine, the solution presented here is in fact a rule based approach. The main reason for this decision is that reproducibility and traceability are not only important requirements, but also a strength of rule-based approaches (and difficult to achieve with many other approaches). As explained in Section 3.3.2, a correlation engine with rules operating on clearly defined and comprehensible input data is easy to understand for humans, and the decisions of the correlation engine are thus understandable, especially if the correlation engine indicates, which rule was responsible for a specific decision.

Since a false positive results merely in more work, whereas a false negative can result in serious problems, it is preferable to accept false positives. This is another point in favor of a rule-based system, because the correlation process can start with an empty rule repository, where the engine simply outputs events as they appear at the input, and rules can be gradually and carefully added.

Another strong point of rule-based approaches is their modularity – since it is easy to separate the rules into independent sets, they can also be distributed on different hosts easily. Furthermore, modularity is essential to allow multiple operators to work with the same system, because the possibility to specify rules independently of the rest of the system allows an operator to add a new rule, without having to know the whole system.

One of the major points of criticisms regarding rule-based approaches is the large maintenance effort, which is needed to keep the rule repository up to date. The analysis in Chapter 2 however revealed, that there are not that many frequent event patterns. As the corresponding events currently have to be handled manually *every time* they occur, having to specify a rule manually *once* for each pattern is certainly acceptable, as long as the creation of the rule is not overly time consuming. Additionally, it can be expected, that the number of different patterns grows merely with the number of managed services, rather than with the number of hosts, and manual rule creation should thus scale well.

4.2 High-level Function Model

The presented approach uses a network of correlation nodes, which forms a tree, where the events flow from the leaves towards the root node. Each node represents an independent correlation engine, which is agnostic of the other nodes, and knows only, what event formats to expect at the input, and where to deliver output events. A complete setup might look as shown in Figure 4.1. Although event sources and correlation nodes are separated in the diagram, this separation is merely a logical one, and correlation engine and event source could just as well be running on the same piece of hardware. As explained in Section 3.3.1, in that case it is however important to avoid unwanted influences between the correlation engine and the host. From the root node,

[1] In fact, they are definitely important – for example, if an arbitrary rule could be injected, hosts where the correlation engine is running could easily be compromised, as the rules allow the execution of arbitrary scripts.

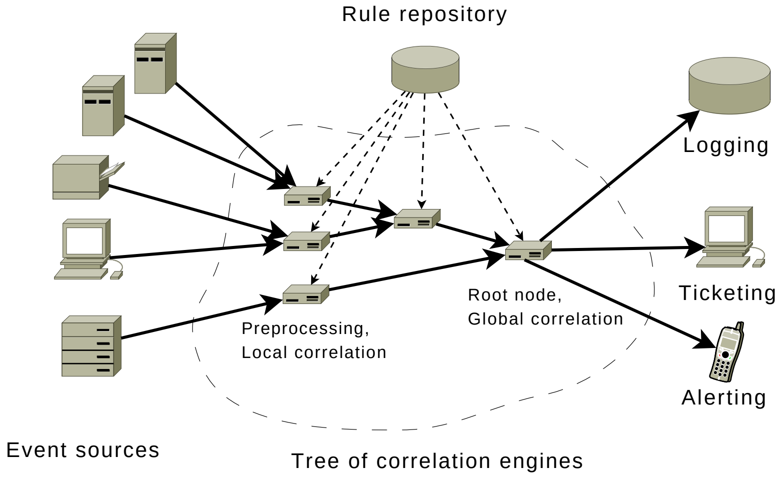


Figure 4.1: Event correlation process.

the events can be entered into a ticketing system for manual handling, stored in a database for logging, or alerting can be done.

The function of each individual node depends only on the input (events, as well as information from external sources besides the events), the internal state and the rules. The consequence of an executed rule can be the creation, modification or deletion of an event, or a modification of the internal state. Because a direct interaction with the environment may be desirable in some cases, there is one exception however: The correlation engine allows the use of plugins, which may interact with the environment. There are two types of plugins, condition and action plugins. Condition plugins provide additional possibilities to specify conditions, such as a condition on the current weekday, or on the return value of an executed script, whereas action plugins provide possibilities for actions with external effects, such as the execution of a script, or the sending of an email. Furthermore, action plugins can also be used to enrich the events with external information. A single processing node works roughly as shown in Figure 4.2. The translators receive events from the network or from a local file and transform input events from their native format (e.g. syslog) to the internal event format (this process will be explained more detailed in Section 4.5). The input queue stores the events from the translators, until they are picked up by the correlation engine core, which processes the events according to the rules in the rule repository. The rule repository contains a local copy of all rules from the central repository, which are relevant for that specific node. If events may be needed later, they are stored temporarily in the event cache. Once the processing is over, events are stored in the output queue, until they can be forwarded across the network.

The correlation engine core is a reactive Event Condition Action (ECA) rule engine, which selects rules based on incoming events, and executes them sequentially. The rules can make use of the input events, the events in the cache, internal state and the plugins, to determine, what output events to generate. Besides selecting, sorting and executing rules, it is also the task of the correlation engine core to manage events and contexts, based on the rules. Section 4.7 describes the rule format, which will hopefully also make the task of the correlation engine core more understandable.

The queues are specified as FIFO queues for the moment. If it should turn out, that another

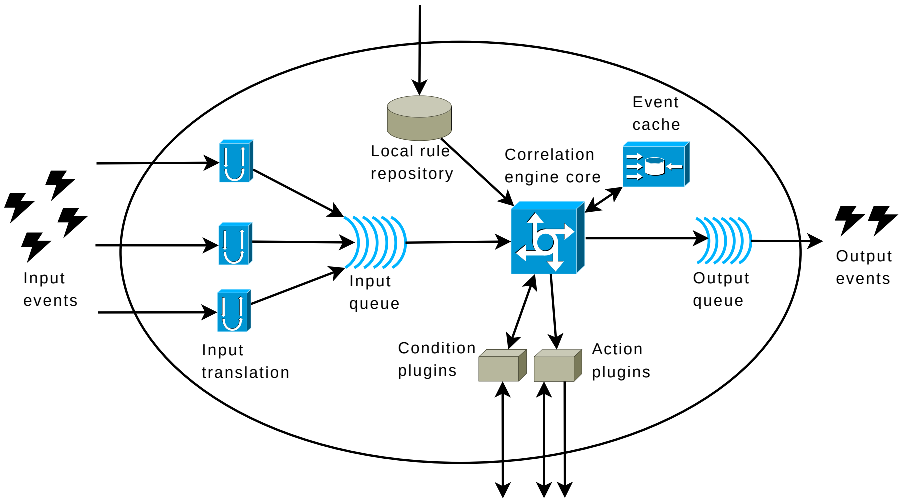


Figure 4.2: Overview of a single event correlation node.

approach is more adequate (e.g. giving priority to certain events), more complex approaches will be tested later.

4.3 Event Format

This section describes the event format. For the moment, the representation of the events in the program, on the network or in a data base is not too important, and it will be assumed that events are simply records with various fields (members), even though an implementation might just as well store the events as objects, or in other formats. An event contains the following fields:

name An immutable string indicating the event family. The string is not required to follow a specific convention, and any format is acceptable, such as the format of the known event `NIC:ETHERNET:LINKDOWN`, or another format, such as `service-xy-down`.

description A human readable description of the event.

id An event ID, which is immutable and unique across *all* events from all hosts.[1]

type A string indicating the event type. The following types exist:

- *raw* – events, which were created directly by an event source, i.e. all events, which were not created by the correlation engine.
- *compressed* – an event representing multiple identical raw events.

[1] This can be achieved by using a cryptographic hash of the host name, the current time (with full precision, so that two events generated right after each other on the same host get different hashes, even if the random number happens to be the same for both) and date and a random number as the ID. The hash length should be at least 128 bits. In that case, if we assume (rather pessimistically) that $n = 10^{15}$ events will be generated throughout the lifetime of the correlation engine, the probability for duplicates according to the birthday paradox approximation formula from [70] would still only be (assuming, there are no collisions in the input data) $p \approx 1 - \exp(-n(n-1)/(2 * 2^{128})) \approx 1.5 * 10^{-9}$‘. With 64 bits on the other hand, duplicates would be almost certain.

- *aggregated* – an event, that collects multiple other events.
- *synthetic* – an event created directly by a rule.
- *timeout* – derived events resulting from a context timeout.
- *internal* – events, which were generated by the correlation engine itself, because of a special condition (e.g. a non-fatal error).

There is a separate namespace for each event type, i.e. two events of different types can have the same name without conflict.

status A value indicating the status of the event, which can be:

- *active* – the event requires further handling.
- *inactive* – the event requires no further handling (this can also be seen as an instruction to a ticketing system, that no ticket needs to be opened because of this event).

While both active and inactive events can be used for further correlation, only active events have the possibility to directly trigger rules.

count For an event of the type *compressed*, this field lists the number of events represented by that event.

host The name of the host, where the event was generated.

creation An integer number, indicating when the event was created. Time and date is saved as the number of seconds since January 1$^{\text{st}}$ 1970 (Unix time).

attributes An unordered list of arbitrary tuples (key/value pairs), specifying additional event attributes.

references An optional list of IDs of events, that are connected to this event. There are three types of references: Child references (pointing to events, which somehow depend on the current event or were caused by the current event), parent references (pointing to events, on which the current event depends, or which were responsible for the creation of the current event) and cross references (pointing to related events on the same level). Just like the status, references can be seen as an instruction for the ticketing system, indicating how to represent events (e.g. if an event has a parent reference, it should end up in the same ticket, and be visually grouped below its parent).

history An ordered list of quintuples, describing the sequence of modifications to the event. The quintuples contain the following information:

- `rule` – group and rule name, indicating, which rule was responsible for the change.
- `host` – host name, indicating, where the change was made.
- `timestamp` – indicating, when the change was made.
- `fields` – an optional list of fields, indicating, which fields were changed.
- `reason` – optional human readable description of the reason for the change.

For events created by the correlation engine, the first history entry references the rule responsible for the creation.

Additionally, the following fields are used internally:

arrival The time, when the event arrived at the local node. If the event was generated locally, the arrival time is equal to the creation time.

local A boolean value, indicating, whether the event is a local event only (which will not be forwarded).

As can be seen, not all fields are required, and we allow the modification of certain fields. Table 4.1 provides an overview of the fields and their properties, and Table 4.2 lists the internal fields.

Field	Type	Mandatory	Mutable	Applicable event types
name	String	Yes	No	All
description	String	Yes	No	All
id	String	Yes	No	All
type	String	Yes	No	All
status	String	Yes	Yes	All
count	Integer	No	Yes	Compressed
host	String	Yes	No	All
creation	Integer (Unix time)	Yes	No	All
attributes	List of tuples	No	Yes	All
references	List of IDs	No	Yes	All
history	List of quintuples	No	Yes	All

Table 4.1: Event record fields.

Field	Type	Mandatory	Mutable	Applicable event types
arrival	Date & Time	Yes	No	All
local	Boolean	Yes	Yes	All

Table 4.2: Internal event record fields.

The corresponding Document Type Definition (DTD) for a representation of events in XML, along with some examples, can be found in Appendix B.1. The XML format can be used to store events in a file, and possibly also for the transportation of events across a network.

4.4 Events Generated by the Correlation Engine Itself

Some events can be generated by the correlation engine itself. The following events, which are local and have the type *internal*, exist:

- `CE:STARTUP` – generated once, when the correlation engine is started.
- `CE:SHUTDOWN` – generated, before the correlation engine is shut down.

Additionally, the following events can be generated, which are of the type *internal* (i.e. generated by the correlation engine), but not local (i.e. they will be forwarded).

- `CE:CACHE:LIMIT:EXCEEDED` – generated, if the number of events in the cache exceeds the specified maximum.

- CE:STARTUP:AFTERCRASH – generated additionally to the CE:STARTUP event, if the correlation engine is started after an unclean shutdown.

4.5 Input Translation

The input translation is responsible for translating input data into the event format specified in Section 4.3. The translator specified here is targeted only at line based input (such as syslog messages); other translators may be presented later.

4.5.1 Line-based Input Rule Format

The translator is based on regular expression matching, similarly to what is used in other applications, such as SEC. Furthermore, the specification allows nested matches, which are evaluated only if the parent matches, similarly to correlation rule dependencies in OSSEC.

Just like the format of correlation rules and events, the format for input translation rules is XML-based. The root element has the name translation_linebased. This element may contain any number of match elements, which must contain an attribute regexp, specifying a pattern to match input lines (the format of the used regular expressions is explained in [32]). The match element may contain event descriptions, actions or nested match elements. The contained elements are evaluated only if the regular expression given in the enclosing match element matched the current input line.

Event descriptions provide information about the event, which will be created. The following XML elements are available for descriptions:

description A human readable description of the event. If omitted, the description will be empty.

host The name of the host, where the event was created. If omitted, the name of the local host will be used.

attribute An attribute with additional information. This element must have an XML attribute name, indicating the name of the attribute (i.e. the key of the key/value pair). As an event may contain any number of attributes, this element can also be used multiple times, with different names (using the same name overwrites the previous value).

datetime The date and time of the event creation. This element must have an attribute format, with a string describing th time format (this string follows the syntax described in [33], which is very similar the syntax of the format string used by the Unix utility date). As some log messages only provide month and day, the additional attribute use_current_year can be used with the value *true*, to use the current year with the specified month, day and time. If the datetime element is omitted entirely, the current date and time is used.

All of the above elements may contain a mix of text and zero or more matchgroup elements, which can be used to recall parts of the regular expression matched by the enclosing match element, by specifying either the name or the number in the attribute group.

The following actions are allowed inside a match object:

create Create an event with the name given as content of the element and the parameters specified so far (i.e. all parameters specified inside the current and enclosing match blocks, but not parameters from previous match blocks). The optional attribute status can be used with the values *active* (default) or *inactive* to create an active or inactive event.

drop Stop the processing of the current input line and continue with the next one. Using a `drop` element is not strictly necessary, as no event will be generated, unless a `create` element is encountered somewhere — but it can be used to speed up the input translation process, as it avoids having to evaluate subsequent `match` elements.

Input lines are processed sequentially, and the processing starts with the first `match` element, which is evaluated recursively. If no `create` or `drop` element is encountered inside the first `match` element, processing goes on with the second `match` element, and so on. Processing of a given line ends, as soon as either a `drop` or a `create` element is encountered. By default, no event is created, but it is of course possible to use a catch-all `match` element, with a regular expression, such as ".*".

The corresponding DTD and an example of translation rules to match SSHd log messages can be found in Appendix B.2.

4.6 Concepts

Before explaining the rule format, it makes sense to comment on some concepts, as the further description builds on these concepts.

4.6.1 Contexts

A context represents an internal state of the correlation engine. For instance, if the correlation engine receives `HOST:UNREACHABLE` events for several hosts, and derives, that a specific ISP link must be down, this knowledge can be represented by a context. This concept is used by various event correlation applications, e.g. by SEC.

Contexts are dynamic. They are always created by a rule, and removed either because they are deleted by a rule, or because their specified lifetime (which can also be infinite) is over. In the second case, a timeout event is generated (if so specified, when the context was created).

Besides representing an internal state, there are two additional uses for a context in our correlation engine. First, events can be associated with a context, to recall them later. In the above example for instance, the `HOST:UNREACHABLE` events could be associated with the created context. If, e.g., later another event is suppressed due to this context, the failing ISP link can be given as reason for the suppression, and the associated events can be referenced.

The second additional use of a context is it's counter, which can be modified and used as a condition by rules. As an example, the counter of a context can be used to create a threshold, without having to store events in the cache, by creating a context with a silent timeout, when the first event of a given type arrives ("first" simply means here, that no context already exists), increasing its counter value upon the arrival of each new event, and checking, whether the threshold is exceeded each time. This would yield a threshold with a fixed window and a dynamic start, whereas the `count` element (explained below) can be used to create a threshold with a sliding window (cf. Appendix A).

4.6.2 Time

There are two different time values associated with an event – the creation time and the arrival time (i.e. the time, when the event arrived at the local node). It is important to realize, that the order of events in the stream may vary, depending on whether the events are sorted by the creation or the arrival time, i.e. event `A` may have been generated before event `B`, but still arrive at a specific node later. Many elements, which require a time value thus allow for an attribute, such as `sort_by`, to specify, whether the creation or the arrival time should be used.

Whenever a time value is required, either an integer can be specified (which will be interpreted as seconds), or the letters s, m, h or d can be used (e.g. *12h*), to indicate that seconds, minutes, hours or days are meant.

4.6.3 Event Caching

Some events need to be kept in the event cache, because they are needed later. This can be done either by keeping a copy of the event, or by delaying the event. Obviously, if only a copy is kept, any changes made to the event will only be local, but on the other hand, events should not be delayed for too long, because it may be important, that the operator sees an event as soon as possible. Furthermore, even copies of events should not be kept too long, because this might use up all available memory.

The decision, how long to cache an event, and whether to keep an copy, or the original, is thus an important one, and should be made with care. Events can be kept in the cache for two reasons, either because the event matches an event query (an event query is a selection of a subset of the events in the cache; event queries will be explained in detail in Section 4.7.2.5) in a rule, or because the event is associated with a context. In total, there are thus four ways to keep an event:

- A copy of an event is kept because of a context: This happens, if the event is associated to a context, which was created with the attribute `delay_associated` set to *false*. The event will be kept, until all relevant contexts were deleted, or until it is no longer associated to any context.

- An event is delayed because of a context: This happens, if the event is associated to a context, which was created with the attribute `delay_associated` set to *true*. The event will be kept, until all relevant contexts were deleted, or until it is no longer associated to any context.

- A copy of an event is kept because of an event query: This happens, if there is any matching event query with attribute `delay` set to *false*, and the event will be kept in cache, until there is no more matching query.

- An event is delayed because of an event query: This happens, if there is any matching event query with attribute `delay` set to *true*, and the event will be delayed, until there is no more matching query.

The evaluation, which case or cases apply to a specific event, and how long each event has to be cached, is the task of the correlation engine.

4.7 Rule Format

The rule format is a bit more complicated than the event format, since many possible conditions and actions are necessary. For better understanding, it is suggested to keep the DTD and the examples given in Section B.3 at hand, while reading this section. The examples also show, how a specific element can be used, and for what kind of event patterns each element is needed.

4.7.1 Rule Groups

The rule repository may be stored in a file or a data base. For the moment, we will assume, that we have a single XML file, which represents the repository. The root element of the corresponding DTD has the name `rules`. This element contains zero or more groups of rules. Although the rule groups were inspired by OSSEC, the meaning is a different one: Groups provide private namespaces for rules, contexts and thresholds. This allows the specification of rule groups without having to worry, whether someone else working on the same rule repository might already have used a specific name (except obviously for the group name, as well as for event names).

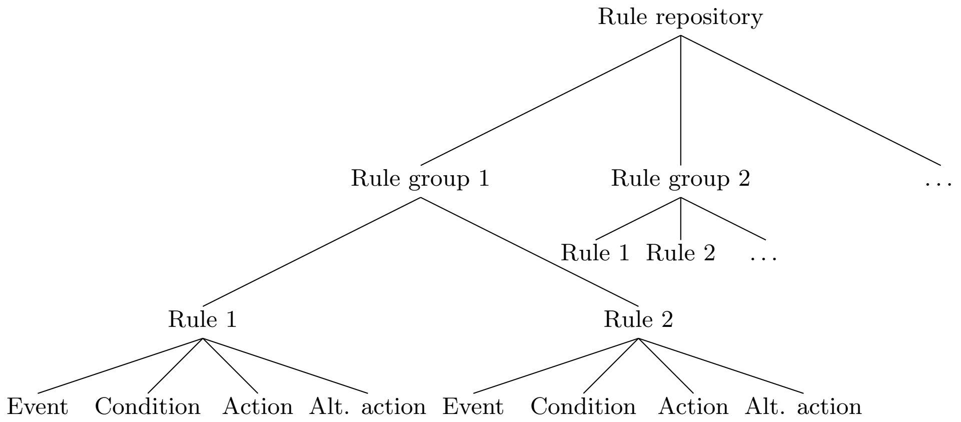
Groups are designated by the element `group` and have the following attributes:

name A mandatory, unique (across all groups) group name. This is a string without any formal restriction (still, it is a good idea to use a naming scheme with short, consistent names).

order A mandatory, unique integer value (greater or equal to zero), which influences the order of execution of the different groups. The group with the smallest order value is executed first.

description An optional human readable description of the groups purpose.

As elements, each group contains one or more rules (element: `rule`), which are specified in the Event Condition Action (ECA) style, extended with an alternative action, which is executed if the conditions are not fulfilled. So far, we can visualize the rule repository as follows:



Next, let's take a closer look at individual rules.

4.7.2 Format of Individual Rules

Just like a group, each rule has the mandatory attributes `name` and `order` and the optional attribute `description`. The name needs to be unique only across all names in the group, and the order attribute influences the rule execution order inside the group in the same way as the group orders influence the execution of the groups (the group order takes precedence, i.e. all rules of a group are executed, before the first rule of the group with the next higher order is executed, independently of the rule orders). The description can be used to provide a human readable explanation of the rule.

A rule contains the following elements, which have to be specified in the same order as they are explained here:

events A list of events, that can trigger the execution of the rule. The rule is executed when *any* of the listed events occurs. This element will be explained in Section 4.7.2.1.

conditions An optional list of conditions. By default, the list evaluates to true only if *all* conditions are fulfilled at the time, when the rule is executed (→ and combination). An empty condition list evaluates to true. Detailed explanations follow in Section 4.7.2.2.

actions A list of actions, which are executed, if the conditions are fulfilled. Detailed explanations follow in Section 4.7.2.3.

alternative_actions An optional list of alternative actions, which are executed, if the conditions are not fulfilled. Detailed explanations follow in Section 4.7.2.4.

All four elements have no attributes.

4.7.2.1 Format of the Events List

The event list specifies, which events can trigger the execution of the rule. The following elements can be used to specify events:

when_class A previously defined event class (which is a list of event names).

when_event A single event name.

when_any Any event can trigger the rule.

All three elements allow the use of an attribute `type`, which specifies, which event type is able to trigger the rule. Multiple types can be listed, separated by a vertical bar (e.g. *aggregated|synthetic*), or *any* can be used to indicate, that any event type is acceptable. The default is *raw|compressed*.

Using a bar for type separation is a slight deviation from the usual XML style, but something like

```
<events>
    <when_event type=”raw|compressed”>SERVICE:UP</event>
    <when_event type=”timeout”>SERVICE:DOWN</event>
</events>
```

has much better readability than a syntax more in XML fashion, like

```
<events>
    <when_event>
        <types>
            <type>raw</type>
            <type>compressed</type>
        </types>
        <name>SERVICE:UP</name>
    </when_event>
    <when_event>
        <types>
            <type>timeout</type>
        </types>
        <name>SERVICE:DOWN</name>
    </when_event>
</events>
```

4.7.2.2 Format of the Condition List

The optional condition list specifies zero or more conditions. Although by default, an Boolean *and* combination of the listed conditions is used, other logical combinations are possible with the following three elements, which can also be nested:

or Evaluates to true, if one or more of the contained conditions are true.

and Evaluates to true, if all of the contained conditions are true.

not Evaluates to true, if the contained condition is false.

The `not` element must contain exactly one child element, while `or` and `and` may contain one or more child elements. None of the three elements allow for any attributes.

The following elements are available for the condition:

context This element evaluates to true, if a context with the given name (specified as content of the element) exists. By default, contexts from the own group are accessed, but the `group` attribute allows to specify a different group. Additionally, the attribute `counter` allows a condition on the value of the context's counter. By default, the actual value must be greater or equal to the given value, but the comparison operator can be changed with the attribute `counter_op`, which can have one of the three values *ge* (greater than or equal), *eq* equal or *le* (less than or equal).

The content of the `context` element, which specifies the name of the context, can be intermixed with a special element, `trigger`, which allows to use a field from the trigger event as part of the string. The `trigger` element is empty, and contains only one mandatory attribute, `field`, which specifies the name of the field (an attribute can be specified with the dot notation, e.g. *attributes.username*). The `trigger` element can be used, whenever a name of a context is expected.

trigger_match Allows to specify a condition on the trigger. The element can contain any of the following child elements, which specify conditions on different event fields: `event_class`, `event_name`, `event_type`, `event_status`, `event_host`, `event_attribute`, `event_min_age`. These elements can also be used in event queries, and will be explained in the corresponding section. The element evaluates to true only if all given conditions are satisfied.

count Allows the specification of a condition on the number of events, which match the contained event query (event queries will be explained is Section 4.7.2.5). The following attributes are allowed:

- `threshold` – number of required events.
- `op` – an operator for the comparison, just like the attribute `counter_op` for the context.

sequence This element verifies, whether the events matched by the queries given as child elements appear in the event stream in the same order, as the queries are specified. The following optional parameters are available:

- `sort_by` – either *creation* or *arrival*, to decide whether to use creation or arrival time as a reference; the default is to use creation time.
- `match` – *any* or *all*, to specify, whether it is sufficient, that any of the event from each query follows the given sequence, or if all events have to follow the sequence.

As an example, let's assume, that we have specified queries for events `A`, `B`, and `C`, in that order. Then,

- The event sequence `A`, `B`, `C` matches.
- The event sequence `A`, `B`, `B`, `D` `C` matches as well.
- The event sequence `B`, `A`, `B`, `C` matches with *any*, but not with *all*.
- The event sequence `A`, `B` does not match.

pattern The pattern element allows to verify, whether the given regular expression pattern occurs in the event stream. It is important to realize, that this is not a regular expression on the event names, but rather, the events themselves are used as symbols. Thus, it is first necessary to specify an alphabet, with a symbol for each event subclass. This is done with the first subelement of the `pattern`, the `alphabet` element. The `alphabet` element in turn, contains one or more `symbol` elements, which specify a symbol (i.e. a single letter) in their attribute `letter`, and an event query for the corresponding event class as content. The `alphabet` element can have one attribute, `sort_by`, which specifies, whether to use creation (default) or arrival time as a reference for sorting. The second subelement of the `pattern` is called `regexp`, and simply contains the regular expression, which must match the events. The element is evaluated by replacing each event in the event stream, which matches a query, with the corresponding letter, and matching the resulting string against the specified regular expression. The specified queries should ideally select disjoint subsets of the event stream. Otherwise, an event, which matches multiple queries, is replaced by the symbol of the first matching query only.

While the use of a pattern condition is a bit laborious, this element is also quite powerful. As noted in Section 2.3.9, some patterns are suitable to be detected by FSMs. As it would be rather cumbersome to specify an FSM for each pattern, the decision was made to allow for regular expressions instead, which are equivalent in terms of the patterns they can match,[1] but simpler to specify (rather than an input alphabet, a set of states, an initial state, a transition function and a set of accepting states, only an input alphabet and a regular expression has to be specified) and better known among system administrators.

The `pattern` element could theoretically be used to replace the `count` and the `sequence` conditions, but as these elements are usually easier to use, they are intentionally still provided, for cases, where they are sufficient.

within This condition can be used to verify, whether the events from the contained queries appeared within the specified time window. The element contains one or more event queries, and has the following attributes:

- `timeframe` – the length of the time window.
- `timeref` – whether to use *creation* (default) or *arrival* time as reference.
- `match` – either *any*, or *all* (default). This attribute specifies, whether it is sufficient, that there is a subset with at least one event from each query, which fits inside the specified window, or whether all events from each query must fit inside the window. In any case, the condition evaluates to false, if not all event queries contain at least one event.

[1] As modern regular expression libraries (which are called regular only for historic reasons) allow for back-references, look-around assertions and other extensions, practical implementations are actually often by far more expressive than FSMs.

condition_plugin This element allows the execution of a plugin to verify external conditions. The `condition_plugin` element has one mandatory attribute `name`, which indicates the plugin name, and contains zero or more `plugin_parameter` elements (which specify a parameter name as attribute (`name`), and the value as content), followed by zero or more `plugin_parameter` elements, which specify a parameter name as attribute (`name`), and the value as content. The number and names of parameters depends on the plugin; but the return value of a plugin is always either true or false.

4.7.2.3 Format of the Actions List

The actions list specifies zero or more actions, which will be executed when the rule is triggered and the conditions evaluate to true.

Besides actions, the actions list may contain two special elements. The first one is `subblock`. This element allows the specification of a complete nested block with conditions, actions and (optionally) alternative actions. The format of each part is the same one as in the rule itself, and recursive nesting is allowed.

The second special element is `select_events`. This element selects the events, to which the following actions apply. The element contains a mandatory event query, which specifies the events to select, followed by any number of actions. `select_events` elements can not be nested and no `subblock` is allowed, but otherwise, the same content as in the actions list is also allowed inside a `select_events` block. Outside of a `select_events` block, the actions apply to the trigger only.

The actual actions are the following ones:

drop Drop the selected events immediately and unconditionally (even if there are event queries on an event, which would lead to caching otherwise).

forward Forward the selected events immediately. If there are matching queries, a local copy will still be kept, but the event will not be delayed.

compress Compress all events of the type raw or compressed into a single compressed event with a count. If the query selects events with different names, a compressed event for each name will be generated. If other fields conflict, the conflict resolves as follows:

- Different descriptions → empty
- Different hosts → name of the local host is used for the compressed events
- Different statuses → *active*
- Different creation times → oldest creation time
- Different values for a given attribute → attribute is omitted
- Different references → union of all references
- History → history is omitted
- Different values for `local` → *false*
- Different arrival times → oldest value

aggregate Creates a new event with references (of the type *child*) to all selected events. The element contains a single element `event`, which describes the new event (this element will be explained in Section 4.7.2.6).

modify Modify the `status` and/or `local` field of the selected events. The element is empty. The following optional attributes exist:

- `status` – can be used to change the status to *active* or *inactive*. The default is to leave the current value.
- `local` – can be used to change the `local` field to *true* or *false*. The default is to leave the current value.
- `reason` A reason for the change, which will end up in the event's history.

modify_attribute Modifies an attribute of the selected events. The element contains the value, and the following attributes exist:

- `name` (required) – the name of the attribute to change.
- `reason` (optional) – a reason for the event history.
- `op` (optional) – an operator, either *set* (default), to set the new value, *inc*, the increment the attribute value by the given value, or *dec* to decrement it. If *inc* or *dec* is selected, and the event's value is not an integer, the attribute remains unchanged.

suppress Suppress the selected events (i.e. set the status to inactive and add the given references as parent references). The element must contain one event query, which should select the events responsible for the suppression. The optional attribute `reason` specifies a reason for the history of the suppressed events.

associate_with_context Associate the selected events with the context, whose name is given as element content. The element content may by intermixed with the `trigger` element.

add_references Add references to the selected events. The mandatory contained event query determines, which events are used as references, and the following attributes exist:

- `type` – whether the references are *child*, *parent* or *cross* (default) references (as explained in 4.3).
- `reason` (optional) – a reason for the event history.

create Create a new event. The element has no attributes and must contain a single `event` element, which describes the new event (as explained in Section 4.7.2.6)

create_context Create a new context. This element contains the following child elements:

- `context_name` (mandatory) – the name the new context. This element has no attributes and specifies the name as content. The content may be intermixed with the `trigger` element.
- `event` (optional) – a description of the event, which will be generated upon a timeout. If omitted, no event will be generated upon a timeout.

Additionally, the following attributes exist:

- `timeout` (mandatory) – the lifetime of the context. Zero can be used to specify an infinity lifetime.
- `counter` (optional) – an initial value for the counter of the context. The default is zero.
- `repeat` (optional) – whether to recreate the context after a timeout. The default is *false*. If this attribute is set to *true*, and a timeout event is specified, the context can be used to generate events in a regular interval.

- `delay_associated` (optional) – if set to *false* (default), only copies of the associated events are kept in the cache. With *true*, the original events are delayed.

delete_context Delete the context with the name given as element content. The element content can be intermixed with the `trigger` element.

modify_context Modify the context with the name given as element content. The element content can be intermixed with the `trigger` element. The following attributes describe the modifications (the default is not to modify anything):

- `reset_timer` – if *true*, the countdown for the contexts timeout is restarted.
- `set_counter` – set a new value for the context's counter.
- `add_counter` – add the given value (a positive or negative integer) to the context's counter.

action_plugin Execute the specified plugin with the selected events as input. The plugin may also change the events (e.g. for enrichment with information from an external source). This element contains a single, mandatory attribute `name`, which specifies the name of the plugin, and contains zero or more `plugin_parameter` elements as child elements. The `plugin_parameter` elements specify parameters for the plugin as key/value pairs (the key is given in the attribute `name`, the value as content).

4.7.2.4 Format of the Alternative Actions List

The alternative actions list specifies zero or more actions, which will be executed, when the rule is triggered and the conditions evaluate to false. Under any conditions, after the rule is triggered, either the actions or the alternative actions will be executed, but never both or none of them.

The syntax and the possible action elements are exactly the same ones as in the actions list. Contrary to the action list, the list with alternative actions is optional.

4.7.2.5 Event Queries

Event queries can be used to select a subset of events from the event stream, i.e. from all past events. This is done with the `event_query` element, which has the following attributes:

max_age This attribute specifies, what the maximum age of the selected events should be. This attribute is very important, as it is used by the correlation engine to derive a cache time for a specific event (the cache time is the largest `max_age` value from *any* possibly matching query). If only events associated with a context, the trigger event, or events matching an other query are selected, it is not necessary to specify a maximum age – but otherwise, this is crucial, as it is not possible to keep events forever.

delay Specifies, whether to keep a copy, or delay the original event. If the goal is to modify an event later, `delay` should be set to *true*. The default is *false*.

timeref Specifies, whether the maximum age is given as creation or arrival time. The default is *arrival*.

name A name for the event query, which can be used to reference it at another location in the same rule group.

The `event_query` element may contain various elements, which select subsets from the event stream. By default, the query returns the intersection of all subsets, but the following elements, which may be nested, allow different set operations:

intersection Returns the intersection of all contained subset.

union Returns the union of all contained subset.

complement Returns the absolute complement of the contained subset (this element must contain exactly one child element, but `intersection` or `union` can be used as child element with more subelements).

For most purposes, it is easier to think of the event query as a boolean condition, which is applied to every single event in the stream, and selects those, for which the condition evaluates to true. From that point of view, the intersection could be seen as a Boolean *and* combination of the contained conditions, the union as *or* combination, and the complement as Boolean negation.

There are however some elements, which select subsets from the contained sets, which can be seen only as set operations. These are the following ones:

first_of Selects the earliest event from the contained subset.

last_of Selects the latest event from the contained subset.

unique_by Selects a subset from the contained subset, for which each value for a given field is unique (e.g. if the field is *host*, the new set contains only one event from each host). The mandatory attribute `field` determines, which field to use, and the optional attribute `keep` determines, whether to keep the *first* or *last* event for each unique value (the default is *first*).

The elements `first_of` and `last_of` are useful e.g. with the `within` condition, e.g. to check, whether the youngest events from various sets of events were all generated within a given window. The element `unique_by` on the other hand is useful mainly in combination with the `count` condition, e.g. to check, whether at least x different hosts generated a specific event. For all three elements, the attribute `sort_by`, which defaults to *creation*, determines whether to use creation or arrival time for sorting.

The actual elements, which select subsets from the event stream, rather than combining other subsets (or, from the boolean point of view, specify conditions on a single event, rather than building a logical combination of other conditions), are the following ones:

is_trigger Selects only the event, which triggered the rule.

in_context Selects all events, which are associated with the given context. The content of this field can be intermixed with the `trigger` element, which was discussed above.

match_query Selects all events, which match the query which the name given as element content. The names again have group scope here.

event_class Selects all events with a name from the given event class.

event_name Selects all events with the given name.

event_type Selects all events with the given type.

event_status Selects all events with the given status.

event_host Selects all events from the given host. The content can be intermixed with the `trigger` element.

event_attribute Selects all events with the given value of the specified attribute. The value is specified as content (which can be intermixed with the `trigger` element), and the name of the attribute is specified with the mandatory attribute `name`. Furthermore, the attribute `op` allows the specification of a comparison operator, which can be *eq* for equals (which is the default), *ge* or *le* (greater than or equal resp. less than or equal) or *re* for a regular expressions. Obviously, *ge* and *le* require integers as both arguments; otherwise, they evaluate to false. *re* requires a valid regular expression in the additional attribute `regexp`.

event_min_age Selects all events, which had at least the given age at arrival (i.e., the difference between arrival and creation time must be at least that large).

The correlation engine must determine automatically, how long to keep a specific event, by looking at all queries. For this reason, it is important to be careful when specifying queries, to avoid an accidental caching of many events. For instance, one might want to know, whether there were more than 1000 events in the last 7 days. To that end, it would be possible, to use a count condition with an empty query and a maximum age of 7 days, as shown in the following listing:

```
<conditions>
    <count threshold="1000">
        <event_query max_age="7d">
        </event_query>
    </count>
</conditions>
```

This would however mean, that all events from the last seven days would be kept in the cache (even if there are more than 1000), and it would therefore be much better to use the counter of a context, to count the events without caching them.

4.7.2.6 Event Specification

Some action elements create new events. Such events are always specified as `event` element. This element has the following optional attributes:

- `status` – the status of the newly created event, either *active* (default) or *inactive*.
- `local` – whether to forward the event to the parent node after processing. The default is *true*, i.e. the event will be kept local and not be forwarded.
- `inject` – where to inject the new event. The default is *input*, which means, the event is placed in the input queue. The alternative is *output*.

The `event` element has the following subelements:

name Specifies the name of the new event as content.

description An optional description of the new event.

attribute An attribute for the event. The key is specified with the attribute `name`, and the value is given as content. Any number of attributes can be used.

The content of the elements `description` and `attribute` may be intermixed with the `trigger` element, which uses fields from the trigger event as part of the string (as explained earlier).

4.7.3 Rule Scoping

If the rules are stored in a local file, they can simply be read directly. As stated earlier, it should also be possible to read rules from a database. In that case, a rule group may apply to more than one host.

An interesting idea, which was suggested by an Open Systems engineer is to allow scoping for event correlation rules. Currently, the configuration for hosts managed by Open Systems is stored centrally in a database. There are configuration settings with a global scope, company scope, service scope or a host scope, and a configuration setting with a more specific scope always takes precedence over a more general setting. The same scheme can be used for correlation rule groups.

Such a feature requires, that each node knows the precedence of the scopes, and the personal value for each scope (country, company, etc.). In the database, each rule group requires two additional attributes, specifying, on which scope the group applies, and for which values (e.g. scope: country, values: Germany, France).

4.7.4 Formal Specification

A complete DTD for rules in XML, along with examples for rules, which could be used to process the patterns identified in Section 2.3, is given in Appendix B.3.

Chapter 5

Implementation

5.1 Preliminary Notes

This chapter discusses the implementation of the developed event correlation engine, which is called *ace* (short for "a correlation engine"). Before we look at the implementation, some preliminary remarks will be made however.

5.1.1 Additional Documentation

5.1.1.1 Code Documentation

This chapter provides a high level description of the implementation. In most cases, this includes a high-level (functional) overview, but no internal details. For a function and class documentation, please refer to the HTML documentation in the folder `doc` on the CD-ROM, that comes with this thesis.

5.1.1.2 The Python Standard Library

Ace makes extensive use of modules from the Python standard library. A documentation of these modules is available in [30].

5.1.1.3 Python Glossary

An explanation of terms specific to Python can be found in the Python glossary [29].

5.1.2 `lxml` Documentation

A documentation of the used XML library, `lxml`, can be found in [25].

5.1.3 Programming Language

For the selection of the programming language, the factor "time to prototype" was considered more important than speed or memory efficiency. While traditional programming languages such as C/C++ usually meet the latter criteria, scripting or high-level programming languages meet the former. From the wide variety of scripting- and high-level programming languages, Python was selected because the developer has considerable expertise in it.

5.1.4 Dependencies

Ace has the following dependencies:

- Python 2.6 – *ace* requires at least Python 2.6. Unfortunately, Python 3 is not backwards compatible and can thus not be used [68] (the `2to3` tool bundled with Python 3 can however be used to generate a patch for *ace*, which should make it work with Python 3[1]). Since Python 3 is currently not yet available in many distributions, *ace* is for the moment however targeted primarily at Python 2.6.
- For XML parsing and generation, `ace` makes use of the `lxml` module (which in turn uses libxml2).
- For daemon mode (optional), the packages `python-daemon` [2] and `lockfile`[3] are required.
- For the IPython console (optional), IPython[4] is required.

Under a recent version of Ubuntu, installing the necessary dependencies should be as simple as:

- `$ sudo apt-get install python2.6 python-lxml`

The optional dependencies can be installed with:

- `$ sudo apt-get install ipython python-setuptools`
- `$ sudo easy_install python-daemon lockfile`

5.1.5 Privileges

Ace requires only normal user privileges in its default configuration. Root privileges are required only if ace is configured to use a privileged server TCP port for a source or for the Remote Procedure Call (RPC) server (as no privilege dropping is currently implemented, this is not recommended).

5.1.6 Document Type Definitions

Ace requires, that the DTDs for rules and events can be found in the location specified in the configuration file. Furthermore, the DTD for rules is also required in the location, which is specified in the document type string in the rules itself (usually `rules.dtd` in the same directory as the rules themselves). *Ace* does not accept rules or input data, which is not valid against the DTDs. On the CD-ROM, the DTDs can be found in the directory `dtd`.

5.1.7 Installation

Ace can be executed from any place. The only requirement is that the environment variable `PYTHONPATH` contains the path, in which the package `ace` is placed (or alternatively, that the package is placed in the same directory as the main script).

[1] A caveat is however, that the 2to3 script only modifies files with a `.py` extension, so the script `ace` should be renamed to something like `ace-main.py` first.

[2] Available at `http://pypi.python.org/pypi/python-daemon/1.4.6`.

[3] Available at `http://pypi.python.org/pypi/lockfile`.

[4] Available at `http://ipython.scipy.org`.

5.2 Top Level Packages

The directory `code` contains the following three Python packages:

- `ace` – the main package, which contains the correlation engine.
- `ace-websink` – a simple web event sink, which can be used to display events in a browser.
- `ace-webui` – a simple web UI, which can be used to display and modify the internal state of the correlation engine (cache, contexts, etc.) in a browser.

As the latter two packages are irrelevant for the functionality of the correlation engine, and contain comparatively few code, they will not be described here.

Additionally to the packages mentioned above, the directory `code` contains one subdirectory, `util`, which contains a small script, `rewrite_logs.py`. This script can be used to scale and shift the timestamps of events in a Comma Separated Values (CSV) event dump, as well as to sort the entries (this is useful mainly for simulation). The script is pretty self-explanatory and will not be explained any further here either.

5.3 The Package `ace`

The main package is called `ace`. `ace` contains several other packages and modules, which can be connected to form a correlation engine. Figure 5.1 shows a high-level diagram of the most important elements of a correlation engine node with three event sources and two event sinks. The dashed lines indicate the individual threads of the correlation engine (elements outside dashed lines belong to the master thread). Additionally to the elements in the previous diagram

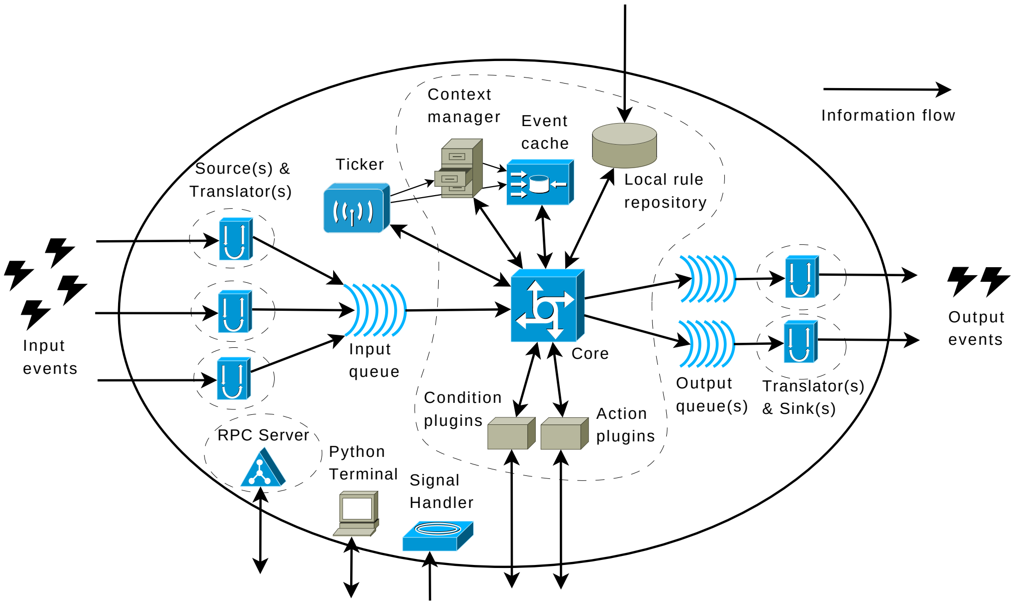


Figure 5.1: Updated correlation engine node diagram from Figure4.2.

(Figure 4.2), there is a ticker (the internal time source), and a context manager (which keeps track of contexts). Furthermore, there is now a representation of the sinks (which generally also implies an output translator for each sink). If there is more than one sink, each sink requires its own output queue[1], as the processing speed of the sinks is unpredictable (e.g., a sink that sends events via a network connection may be stalled by a broken connection). Finally, on the lower left of the diagram, the possibilities for user interaction and external control can be seen. There is an optional RPC server (which is used by the WebUI), an optional Python console, and signal handlers for various Interprocess Communication (IPC) signals.

In the following sections, the individual elements will be explained in more detail.

5.3.1 The Script `ace`

The script `ace` is the main script. Although some of the modules also provide `__main__` functions, and can thus be executed directly, this is only for testing purposes, while `ace` is the usual entry point for normal execution.

The tasks of the script are rather limited – it does command line argument parsing, instantiates an object with configuration information and detaches from the terminal, if the daemon mode was requested. After that, `ace` transfers the control to an instance of the `Master` class, which is subsequently responsible for the instantiation of a correlation engine.

Please note that although the script `ace` is placed in the package `ace`, it is technically not part of the package, but rather, it makes use of the package to instantiate a correlation engine. The script `ace` can also be placed outside of the package (e.g. in some `bin` directory). In any case, it is necessary, that the environment variable `PYTHONPATH` is set to the directory, which contains the package `ace`.

An overview of the available command line parameters and configuration options can be found in Appendix C.

5.3.2 The `util` Package

The package `util` provides some modules with general purpose functions and classes, for tasks, such as:

- Generating syslog messages (the module `logging` provides a wrapper class for the `syslog` module from the Python standard library).
- Configuration file parsing (the module `configuration` provides a suitable class).
- Exceptions (the module `exceptions` provides several exception classes).
- Constants (constants are stored in the module `constants`).
- A `help` module with a class containing some help for the interactive mode.

As the provided functions and classes are fairly common, no further explanation of these modules should be needed.

5.3.3 The `tests` Package

The package `tests` contains several modules with test classes, which provide unit tests for function verification. Each module can either be executed individually, or all tests can be run at once with the script `run_tests` from the package `ace`.

[1] As the queues only store references, which point to the same events for each queue, the memory usage is not significantly higher, if there is more than one output queue.

5.3.4 Master

The `Master` class (which can be found in the `master` module), is responsible for starting and controlling the individual threads. The master directly instantiates the queues, the source and sink threads, a ticker instance and a core thread, as well as a thread with an RPC server, if configured to do so. The further behaviour then depends on the execution mode.

5.3.4.1 Normal Mode

In the default mode, the master starts all threads and goes to sleep, until a command arrives or until a child dies (in which case, the master stops all threads and exits[1]). By default, the master sleeps indefinitely.

5.3.4.2 Debugging Mode

The debugging mode can be requested with the `-p` or the `-i` command line switch, or by setting either of the configuration variables `python_console` or `ipython_console` to *true*. The debugging mode works just like the normal mode, except that rather than going to sleep, the master starts a Python or IPython console, which can be used for debugging and interaction with the correlation engine.

The console allows direct execution of code, and can thus be used to control or observe any part of the correlation engine. Possible applications could be to inspect or change variables, to start and stop threads, or even to redefine functions and classes during runtime.

When the user exits from the console, the master stops all threads and exits.

5.3.4.3 Simulation Mode

Simulation mode can be started by setting the configuration variable `simulation` to *true*. In simulation mode, the master only starts the threads for the sinks. The source and core threads are not started, but rather, the master calls their `work()` functions individually in an ordered fashion. The master first calls the `work()` function of each source once, so the sources can generate input. Secondly, the master sets the ticker to the time of the first event (it is thus necessary, that the input events are already sorted). Finally, the master repeatedly calls the `work()` function of the core, until all input is processed, and exits.

Calling the sources and the core one after another is necessary for simulation, to make to results reproducible. If the threads would run asynchronously, the results could depend on the scheduling of different sources (in normal mode, this is inevitable, as we can not prevent, that e.g. one event arrives slightly earlier or later due to varying network latency).

Furthermore, synchronous operation is also faster for simulation, as it would otherwise be necessary to make at least one context switch per thread and time step, to see whether a given source has new input, respectively whether the core can process input or generate new output (in normal mode, the overhead is unproblematic, as each time step lasts one second, and it is not even necessary to call each source at least once per step).

5.3.4.4 Control of the Correlation Engine via IPC Signals

In all three modes, the Master registers handlers for the following three IPC signals, which can be used to control the correlation engine:

[1] An alternative would be to only generate a warning and restart the corresponding thread. Since the threads are connected only via the queues, each thread is completely independent, and restarting a thread would require no synchronisation or the like.

- SIGHUP – can be used to request a reload of the correlation rules.
- SIGTERM – can be used to request an ordered shutdown (i.e. the master asks each thread to stop, and waits until all threads are finished).
- SIGINT – can be used to request an immediate shutdown (i.e. the master exits directly, without asking the threads to stop).

5.3.5 The RPC Server

The RPC server, which can be started with the `-R` command line parameter, or with the `rpcserver` configuration variable set to *true*, serves as an interface to an external controller. The rpcserver is implemented in the `RPCHandler` class in the `rpc` module. It makes use of the `SimpleXMLRPCServer` class from the Python standard library and provides the following three functions, which can be called via RPC:

- `getStats()` – get some statistics from the correlation engine (returns a list of key/value pairs).
- `getContent(page)` – get the content of the page with the given name for display in the UI.
- `execAction(action, arguments)` – execute an action and return the results to the client.

Currently, the only client for the RPC server is the WebUI, which renders the content in HTML. The server is however generic, and it should be possible to implement e.g. a GUI, without changing the server. Independently of whether *ace* runs in simulation or normal mode, the RPC server is always started as an individual thread.

5.3.6 Events

The module `event` contains the class `Event`. This class is used to represent single events, and contains several helper functions to get, set or modify event details. Although using a class to represent events has some memory overhead, the overhead is not excessive, as a class instance only stores the attributes, but no individual copies of the class functions.[1]

Additionally, the `event` module also contains an `EventGenerator` class, which can be used to generate random events for testing.

5.3.7 Queues

The queues store events until they can be processed, and provide an interface between the sources and the core, respectively the core and the sinks. The different threads are independent from each other (but not from the master thread, however), and they do not interact in any way, other than by producing and consuming events.

For the queues, we use the class `Queue` from the Python standard library module `Queue` in the package `queue` [31]. Conveniently, this module already provides a FIFO queue, which also implements the necessary locking mechanisms for multiple concurrent producers and consumers. The queues themselves are not threads; they are simple objects, shared between the threads.

While the `Queue` module is thoroughly documented in [31], it is noteworthy, that queues can be joined. A join operation on the queue blocks until all items (i.e. all events, in our case) from

[1] Tests show, that an instance of an average event requires about 900 bytes.

that queue have been processed (signaling, that an item has been processed, is the responsibility of the consumer thread; the mere fact, that a queue is empty, is not a sufficient condition for a successful join). This mechanism is used by the master thread, to make sure that all events from the queues have been processed before shutting down, if the configuration variable `fast_exit` is set to *False* (which is the default). To achieve this, the master simply executes a join on the input queue, before stopping the core thread, and a join on each output queue, before stopping the corresponding sink thread.

5.3.8 Ticker

For simulation, the correlation engine can not use the system clock as time source directly, because of the following problems:

- In a simulation run with events collected throughout one month, having to wait a month until the simulation is completed, is not acceptable.
- Independently of how much physical time is used, the correlation engine must not advance to the next simulation step, until all processing for the current step is done (actually, this is preferable also in normal operation, as the results should not depend on how much processing has to be done).

It is thus useful to have a degree of abstraction from the system time. In *ace*, this abstraction is provided by the `Ticker` class in the module `ticker`. This class is quite simple – its main task is to keep track of the current tick, which is stored in the internal variable `tick`. The Ticker class never advances the tick itself, but rather, the core asks the ticker to advance the tick, as soon as the core has processed all input events with a creation time smaller or equal to the current tick.

The advance function of the ticker looks as follows:

```
def advance(self):
    if self.config.realtime:
        while self.tick >= int(time.time()):
            time.sleep(self.config.thread_sleep_time)
    self.tick += 1
    self.logger.logDebug("Tick advanced to %d." % self.tick)
    return self.tick
```

While this is really straightforward, there is one important property, which is noteworthy. If the boolean variable `realtime` is set to *True*, the ticker waits, until the system time has reached the next second, before advancing to the next tick itself. These three lines make the difference, whether *ace* runs in real-time (and uses at least one second per tick, as it sleeps, if the processing didn't last that long), or whether it runs as fast as it can (and thus with maxed out CPU usage, as it advances directly to the next tick, when all input events have been processed, or if there were no input events).

With the two boolean variables `simulation` (discussed in Section 5.3.4.3) and `realtime`, there are theoretically four possible combinations. In practice, the execution is however also influenced by the input, i.e.,

- whether the input events already have timestamps (both for their creation and arrival time),
- and whether the input events arrive continuously or are all known at the start.

This leads to different use cases for the four combinations of `simulation` and `realtime`.

- `simulation` false, `realtime` true – this is the normal execution mode; it makes most sense with events, that arrive continuously. On the other hand, use of this mode with a-priori known events is useful for a simulation, that is as close as possible to the normal execution mode. This however requires, that the events already have timestamps, and that the timestamps lie in the near future.[1]

- `simulation` true, `realtime` false – this is the normal simulation mode, which makes most sense with events, which already have timestamps (and are sorted according to these timestamps). Continuously arriving events can not be used whenever `simulation` is set to true, because input events are then only read at the start.[2]

- `simulation` true, `realtime` true – this configuration can be used to do a simulation in realtime, which can be useful with events, which already have timestamps, but again only if the timestamps lie in the near future.

- `simulation` false, `realtime` false – this configuration could be used to do a threaded simulation in non-realtime. The results would not be reproducible, but closer to the normal mode than with `simulation` true, `realtime` false, while the simulation would still run faster than with `simulation` false, `realtime` true.

In most cases, the first two configurations are sufficient.

5.3.9 Sources and Sinks

Sources and sinks can be found in the packages `ace.io.sources` and `ace.io.sinks` respectively. They are responsible for data input and output, but work independently of the data format. The base class for sources and sinks can be found in the `base` modules in the corresponding package directories. The base classes themselves are derived from Pythons `Thread` class. Currently, the following sources are available (listed by their module name):

- `file` – reads data from a file or from standard input (if no filename is given).

- `tcp` – provides a socket server, which listens for input data on a TCP port.

- `ticker` – generates events at a regular interval (this source is available mainly for testing, and is usually not needed otherwise; during normal operation a context with its attribute `repeat` set to *true* would be a better way to regularly generate events). Please note, that this source has no relation to the `Ticker` class explained in the previous section.

- `null` – a null source, which never generates an event (useful mainly for testing).

The available sinks are the following ones:

- `file` – writes data to a file or to standard output (if no filename is given).

- `tcp` – sends output data to the specified TCP port on the specified server.

[1] With timestamps in the past, all input events would simply be processed in the first processing step, whereas with events in the future, the correlation engine does not process the first event, before the real time reaches the time of the event. For events in CSV format, the script `rewrite_logs.py` can be used, to shift and scale the timestamps of events.

[2] The result of such a configuration would be that only the first batch of input events is read, e.g. until the first End of File (EOF) is reached (file source) or the first TCP connection is closed (tcp source), and processing would not start, before the first EOF is reached or the first TCP connection is closed.

- `rpc` – provides an RPC server, from which the output data can be read. This sink is special in that it does not require an output translator, as it provides events directly as dictionaries.

- `null` – a null sink, which discards all events it receives (useful primarily for testing).

A more detailed explanation of each source and sink, which also lists the available options, can be found in the HTML documentation of the respective classes.

5.3.10 Translators

Translators are responsible for translating events from and to Python objects and various data formats. Input and output translators can be found in the packages `ace.translators.input` and `ace.translators.output` respectively. Again, the base classes can be found in the respective `base` modules. Translators are simple objects, and each source and sink always instantiates exactly one input and output translator, respectively. Any combination of source, input translator, sink and output translator is possible. Currently, *ace* provides the following input translators:

- `letter` – translates each alphabetical letter in the input data stream to an event (this is useful mainly for manual testing).

- `linebased` – translates input lines into events, according to translation rules as specified in Section 4.5. This translator can be used e.g. to match log messages.

- `xml` – translates XML events, as specified in Section 4.3. This data format is useful primarily for exchange of data with other applications.

- `pickle` – translates input in the ASCII format generated by the Python `pickle` module into events. This format is both more compact and can be parsed significantly faster than XML events, and is thus more suitable for transport of events between different correlation engine nodes. The drawback is however, that this format is less universal, and thus not suitable for data exchange with other applications. It should also be noted, that `pickle` is not safe for use with untrusted data sources.

- `csvdump` – reads input data from a CSV database dump (suitable mainly for simulation).

- `nop` – does nothing. This translator is meant as placeholder for use with sources, that do not require a translator (such as the `null` and the `ticker` sources).

The available output translators are the following ones:

- `linebased` – generates a short textual representation in one line for each event (note: this translator is NOT meant to be the counterpart to the input translator with the same name).

- `pickle` – generates a stream of events represented in Python's `pickle` format. This module is the counterpart to the `pickle` input translator.

- `xml` – generates a stream of XML events. This module is the counterpart to the XML input translator.

- `nop` – does nothing. This translator is meant as placeholder for use with the RPC sink.

Again, a more detailed explanation can be found in the HTML documentation.

5.3.11 Plugins

Two types of plugins exist, condition plugins and action plugins. Currently, the following condition plugins are provided:

- Weekday – allows a condition on the current week day.
- ScriptReturnValue – allows a condition on the return value of an executed script.

Action plugins:

- EventLogger – allows the logging of individual events.
- MailAction – allows the dispatching of mails with events.
- EnrichRegexp – allows the use of a regular expression to extract a port from an attribute into a new attribute.

These plugins are provided mainly for demonstration purposes. They can be found in the packages `plugins.condition` and `plugins.action`, again with base classes in the respective `base` modules.

5.3.12 Core

The core class is implemented in the module `core` and is called `EventHandler`. This class is derived from Python's `Thread` class, and it is responsible for creating instances of the `EventCache`, `ContextManager` and `RuleManager` classes.

As most of the work is done in the instantiated classes, the main loop in the core (implemented in its `work()` function) is rather simple:

1. If a rule reload has been requested by the user, ask the rule manager to do it.
2. Ask the context manager to update the contexts and possibly generate context timeout events.
3. Ask the cache to clean up old events and possibly forward events, that were delayed.
4. While there are either input events generated internally or input events in the input queue with a timestamp smaller or equal to the current tick, do the following:
   i) Get the next input event from the events generated internally or from the input queue (the former always have priority).
   ii) Ask the rule manager to determine cache and delay time for this event (indicating, how long the event needs to be kept in the cache, respectively delayed).
   iii) Insert the event into the cache.
   iv) Get all relevant rules from the rule manager, sorted according to group and rule order.
   v) As long as the event is in the cache (i.e. has not e.g. been dropped by a rule) and active, execute one rule after the other.
   vi) If the event was taken from the queue, signal to the queue, that the event has been processed.
5. If a clearing of the cache has been requested by the user, ask the cache to do it.

6. If there are any events, which were modified, reevaluate their cache and delay time.

7. Ask the ticker to advance to the next tick.

This routine is either executed in an infinite loop, until the correlation engine is stopped (normal mode), or it is called repeatedly by the master (simulation mode).

5.3.13 Cache

The cache, which is implemented in the `EventCache` class in the `cache` module, is responsible for storing events. Each event has a delay time (which indicates, how long an event should be delayed before being forwarded), and a cache time (which indicates, how long a copy of the event must be kept in the cache, because a rule may need the event), which are both derived from the rules by the rule manager. Furthermore, contexts can also extend the cache and delay times (as discussed in Section 4.6.3).

The cache is based on three main data structures:

- `events` – a set with references to the events in the cache.

- `delay_list` – a list of timestamps with forwarding times (i.e. the time, when the delay is over) of events, and references to the corresponding events.

- `cache_list` – a list of timestamps, with cache removal times of events, and references to the corresponding events.

When a new event arrives (and already contains cache and delay time, which were determined by the rule manager), the cache adds the new event into the cache and inserts timestamps for the times, when the cache and delay times are over, into the corresponding lists. As the two lists are always kept in a sorted state, the events must be inserted in the right place, which is done with the help of Pythons `bisect` module.

When an event was changed, the old timestamps are removed from the lists, and new ones are inserted.

Whenever the core asks the cache to clean out old events, the cache simply needs to check the events at the start of the cache and delay lists, which have a timestamp smaller than the current tick, and forward, respectively remove them from the cache if necessary.

Besides keeping track of events, the cache also contains some functions for minor tasks, such as compressing events.

5.3.14 Context Manager

As the name implies, the context manager keeps track of all active contexts. It is implemented in the `ContextManager` class in the module `contexts` and makes use of the `Context` class (which represents a single context) in the same module.

To keep track of timeouts, the context manager uses a sorted list with timestamps of possible context timeouts (the variable `context_timeouts`), just as the cache keeps a list with timestamps of cache and delay times. The contexts themselves are stored in the variable `contexts`, which is a nested dictionary, with the group name as key on the first level, and the context name as the key on the second level.

5.3.15 Rule Manager

The rule manager is responsible for managing correlation rules. It is implemented in the `RuleManager` class in the `rulebase` module, and makes use of the classes `RuleParser` (which is responsible for parsing XML rules), `RuleGroup` (which represents a single rule group) and `Rule` (which represents a single correlation rule) in the same module.

The rule parser creates the rules by making use of the rule elements from the module `rulecomponents` in the package `basisfunctions`. For each XML rule element, there is a corresponding function with the same name as the element[1] in `rulecomponents`. This function can be

- directly the function, which represents the element (this is the most simple case, but only possible, if the represented element does not have any attributes or child elements),
- a function, which returns a new function, which is a closure over the element attributes and which represents the rule element (if the rule has attributes, but no children),
- or a function, which returns a new function, which is a function generated from one or more other functions (if the element contains child elements).

An example for the first case is the `drop` element, which is represented by the following function:

```
def drop(**kwargs):
    kwargs['cache'].dropEvents(kwargs['selected_events'])
```

As the actual work is done in the cache, this function is as simple as calling the appropriate function in the cache with the events passed in the argument `selected_events` (the function does still accept any argument, so the caller does not have to discern, which action element it is calling).

An example for a function, which returns a closure over the arguments, is the `modify` element, which can be used to modify the status and local fields of the events:

```
def modify(status, local, rule, reason):
    def modify_generated(**kwargs):
        events = kwargs['selected_events']
        core = kwargs['core']
        tick = core.ticker.getTick()
        hostname = core.config.hostname
        kwargs['cache'].removeStaleEventsFromList(events)
        for event in events:
            fields = []
            if status != None and status != event.getStatus():
                event.setStatus(status)
                fields.append('status')
            if local != None and local != event.getLocal():
                event.setLocal(local)
                fields.append('local')
            if len(fields) > 0:
                event.addHistoryEntry(rule, hostname, tick, fields, reason)
        core.addModifiedEvents(events)
    return modify_generated
```

This function basically updates the `status` and `local` fields of selected events, adds an appropriate history entry and notifies the core about the modified events (whose cache and delay times must be reevaluated at end of the current processing round).

[1] Except where the name is a reserved keyword in Python, in which case an underscore is added.

An example for a function generating a new function (which is also a closure) from an existing function (in this case the `query` function) is the function for the `count` element, which checks, if the passed query selects at least, exactly or at most the given number of events (depending on the attribute `op`):

```
def count(threshold, op, query):
    if op == "eq":
        return lambda **kwargs: len(query(**kwargs)) == threshold
    elif op == "le":
        return lambda **kwargs: len(query(**kwargs)) <= threshold
    elif op == "ge":
        return lambda **kwargs: len(query(**kwargs)) >= threshold
```

The code is in this case pretty self explanatory. It is noteworthy though, that the query function is evaluated, when the rule is executed (and may of course return different events each time), and not at build time (since the arguments to the function are not known at build time).

An equally short, but more complicated example is the function for the element `and`:

```
def and_(conditions):
    if len(conditions) == 0:
        return true # this is a function returning True, not a boolean value!
    else:
        return reduce(lambda a,b: lambda **kwargs: a(**kwargs) and b(**kwargs),
                      conditions)
```

If there are no child elements, the function returns a function, which returns true (because the element was defined to evaluate to true, if there are no child elements). If there are children, the children are reduced with a function (the outer lambda), which takes two functions as arguments and returns a new function (the inner lambda), which evaluates to true, if the two functions passed as arguments both evaluate to true.

While the use of functional programming and closures can lead to slight headaches, it also allows us to build actual rule functions when the correlation engine is started, rather than having to use `eval()` or `exec()` during runtime, as many of the existing correlation engines do (the main problem with `eval()` in a correlation engine is its slowness, due to having to parse the expression string, each time a function is evaluated; furthermore, `eval()` has also been criticized for being insecure [71], and it can be harder to read and debug than other solutions).

Besides building the rules, the rule parser also extracts some additional information from the rules, such as the set of event and class names. Furthermore, the rule parser also generates a hash from a recreated XML representation of each rule group (the advantage of recreating the XML is that the comments and whitespace can be stripped, and the hash does not change, if only a comment or indenting was changed). With this hash, the rule parser can decide, which rule groups were unchanged, when the user requests a reload of the correlation rules. Not having to recreate unchanged rules is not only more efficient, but also has the advantage, that contexts of an unchanged rule group do not have to be deleted, when reloading the rules.

After the rule parser has parsed the rules, the rule manager builds a lookup table of the relevant rules for any given event name and type. When the core asks the rule manager for the relevant rules for a specific event, this table is used together with a list of rules relevant for all events, to determine the response.

5.3.15.1 Event Queries

The event queries, which are responsible for selecting events from the cache, can be built in the same way as the rest of the rules, by combining components from `rulecomponents`, as explained above. While building the queries, some sanity checks are also done, e.g. for the `match_query` element, which can be used to select events, which match another query. For this element, we need to check, that the matched query exists, and that there are no "`match_query`-loops" (this check is implemented in the recursive function `detectQueryLoops` in the `RuleParser`).

The rule manager however also has the task to determine cache and delay times for events from the queries, which requires some additional work.

To determine the cache and delay time, we have to decide, which queries match a given event. Thus, for each query, we need an additional function, which can determine, whether that specific query matches a specific event. A simpler approach would be to simply extract all relevant event names and other needed information from the XML representation of the query with an XML Path (XPath) expression, but such an approach could only work in the most basic cases, as a condition might also be inverted ("all events not originating from host `X`"), or combined with other conditions. By building a function for each query according to the Boolean query interpretation from Section 4.7.2.5 (i.e. interpreting the elements as Boolean condition on a single event, e.g. "the name of the event is `X`", or as a combination of several conditions, such as the interpretation of the `intersection` element as *and*-combination of the contained conditions), we can use the Boolean evaluation built into the underlying interpreter, to evaluate the queries.

As explained earlier, not all elements can be evaluated for a single event. For instance, the elements `first_of` and `last_of` have a meaning only in context of a set of events, and in this case, we need to cache or delay all events, which could possibly be the first or last of the selected set (i.e. all events in the set). For other elements, it may not be possible to decide, whether the event matches a query when the event arrives, e.g. because the query requires information from an event arriving later (this is the case specifically whenever the `trigger` element is used). For these cases, we have to introduce a third state, *undefined*, which is treated according to the truth tables shown in Table 5.1. The tables also introduce a fourth state, *defined*, which represents the case, where we assume that we can determine the boolean value, but it is currently not known (this state is identical to *undefined*, except that a combination of *defined* and *undefined* always evaluates to undefined). This fourth state is however only relevant for building query tables, which will be explained later.

With these tables and the elements in the module `querycomponents`, we can build query determinator functions, which can be evaluated for a given event to either *true* (the query applies to the tested event), *false* (the query does not apply to the event) or *undefined* (we can't decide, whether the query applies to the tested event).

Once all of these functions are built (which has to be done, whenever new rules are loaded), each new event can simply be tested against all queries, to decide how long to delay and cache that specific event – the delay (cache) time is the maximum `max_age` of all queries with `delay` set to *true* (*false*).

While there is no reason against this simple "brute-force solution" from a functional perspective, this solution would obviously result in bad scalability, as *each* new event would have to be tested against *all* queries, whenever a new event arrives.

5.3.15.2 Query Tables

An alternative solution is to build query tables, which can be used to look up the relevant queries for a given event, similar to how the rule table can be used to look up the relevant rules.

For instance, we could build a table for all combinations of event name, state and type, to look up the corresponding queries. Obviously, there are quite many combinations, and building the query tables instead of trying each query for each event is thus a time-memory trade-off. In the current implementation, the query tables are built for each event name only, without discerning between different types or states. As most queries are likely to apply only to specific

x	y	$x \wedge y$
0	0	0
0	1	0
0	u	0
1	0	0
1	1	1
1	u	u
u	0	0
u	1	u
u	u	u
0	d	0
1	d	d
u	d	u
d	0	0
d	1	d
d	u	u
d	d	d

x	y	$x \vee y$
0	0	0
0	1	1
0	u	u
1	0	1
1	1	1
1	u	1
u	0	u
u	1	1
u	u	u
0	d	d
1	d	1
u	d	u
d	0	d
d	1	1
d	u	u
d	d	d

x	$\neg x$
0	1
1	0
u	u
d	d

Table 5.1: Truth tables for the Boolean operations *and*, *or* and *not* with input values true, false, undefined or defined.

event names (which can also be seen in the queries shown in Appendix B.3.2), and because there are quite many event names, this provides a good trade-off.

To build the tables, we need the possibility to check, whether a given query matches an event with a given values for some fields, but undefined values for other fields (e.g.“does this query apply to events with name `ISP:OUTAGE` and any event type, host name, etc.?”). This functionality is provided by the `predet_wrapper` function in the `querycomponents` module, which allows to preset values for some of the components. The tables are currently built roughly as follows (the corresponding code can be found in the function `buildQuerytable` in the `RuleManager` class):

- The query is first tested with all components depending on event data set to undefined. If this test evaluates to *false*, we know that no event needs to be held back due to this query (as the query would always evaluate to *false*), and this query can be ignored (this is the case e.g. for a query, which only selects the event, which triggered the rule, using the `is_trigger` element).

  In this case, it is also allowed to omit the `max_age` attribute of the event query in the rule, as the maximum age of the events is already limited (if the `max_age` attribute is omitted and the query does not evaluate to *false*, *ace* complains, that the maximum age can not be inferred for the query).

- If the above test evaluates to *true*, all events have to be delayed or cached (depending on the attribute `delay` of the query element) due to this query (at least as long as specified by the by the `max_age` attribute of the query), as this query evaluates to *true* for any event. This is the case e.g. for an empty query (no conditions).

  Since such queries always evaluate to *true*, there is no need to store each individual query in the table, but rather, we simply need to know the maximum delay and cache time of all queries, which always evaluate to *true*. This value is the minimum delay, resp. cache time for any event.

- If the above test evaluates to *undefined*, another test is done, with all components depending on event data set to *defined*.
    - ◇ If this second test still evaluates to *undefined*, we know that even with event data known, the query can't be evaluated. In that case, we also have to cache or delay all events at least as long as specified by this query, and there is no reason to store the query, but rather, we just remember the global maximum value for delay and cache time.
    - ◇ Otherwise, the outcome can be determined with event data available. In that case, a third test is made, with all components depending on event data set to *undefined*, except components depending on the event name (i.e. the elements `event_class` and `event_name`), which are set to *false*.
        - ∗ If this third test evaluates to *false*, we know that the query is not matching every event with any name, and it makes sense to store the query only at the correct positions in the table (i.e. under all event names, for which this query is relevant). In that case, the query is tested against each event name, resp. all event names, which appeared either directly or indirectly (via a class name) in a query. If this individual test evaluates to *true*, only the cache or delay time has to be stored in the table; if it evaluates to *undefined*, the query itself is stored, so that it can be evaluated when an event actually arrives. The query is however stored only if it would delay or cache the event longer than the local (i.e. per event name) and global minimum value.
        - ∗ Otherwise (i.e., if the third test, returned *true* or *undefined*), the query is stored in the global table, rather than storing it at each position in the table. Again, this is done only if the default value for cache or delay time is not larger than this querys value.

The function `updateCacheAndDelayTime` in the `RuleManager` is used to actually determine the cache and delay time for a given event. This is done by first determining the relevant queries for a given event. These are then sorted according to the cache or delay time they would require, if the event matches, and consequently evaluated one after the other. Because of the sorting, the queries only have to be evaluated until the first match is found.

Since the whole process is rather complicated, the brute-force solution has also been implemented, and can be used in debug mode, to verify that it results in the same cache and delay time as when using the query tables.

Chapter 6

Evaluation and Refinements

This chapter discusses the evaluation of *ace*, as well as some refinements.

6.1 Functional Verification

6.1.1 Unit Tests

As already mentioned in Section 5.3.3, the directory `ace/tests` contains several unit tests. At the moment, this directory contains 13 classes with 66 tests for core components as well as for plugins, translators and I/O classes.

As these tests are implemented only for functional verification, they will not be discussed any further.

6.1.2 Event Balance

To provide some kind of a self check, the core keeps track of the number of events read from the input queues generated, dropped, compressed or written to the output queues.

A balance of these numbers can be requested from the core via the core's function `getEventBalance()`. The returned number should be (and indeed is) usually zero (it may however be temporarily off by one, when the function is called while the core is processing an event), and can be used as a sanity check.

6.2 Profiling

As the Python standard library provides some aid for profiling through the modules `cProfile` and `pstats`, profiling is directly built into *ace*. Profiling can be enabled from the command line with the `-P` switch.

As profiling was implemented at an early stage during the implementation phase, it already led to some improvements during development, such as:

- Originally, the event cache used Pythons `list` data type to internally store events. As the `list` stores individual elements according to indexes, the membership test has a time complexity, which is linear to the number of elements in the list (as each element has to be tested individually). The data type was thus changed to Pythons built-in `set` type, which stores elements according to their hashes, and whose membership test is orders of magnitude faster than the membership test of a `list`.

- Tests also showed, that the function for the logging of debug messages (which is done only at the highest log level, as debug messages are very verbose) used up a significant amount of the CPU time, even with debugging turned of. Further investigation revealed that this was because all elements of the format string were converted, even if the resulting string was not printed. While this behaviour is not surprising, and usually no problem, some of the conversions can be quite slow.[1] A slight change in the `logDebug()` function, to build strings only if they are going to be printed (or logged to syslog), led to a significant improvement here.

- At an early stage, the event cache was generally too slow. This led to a new implementation with timestamps in permanently sorted lists (as discussed in Section 5.3.13).

With these improvements, the profile of an average simulation now shows, that the main time is spent in the `work()` function of the core, the `updateCache()` function of the cache, and the `updateContexts()` function of the context manager. A rather large amount of the time is also spent in the function `generateOutputEvent()` of the core, as this function makes an in-memory copy of each event, before putting it into the output queue (as an event may still be kept in the cache, after it has been put into the output queue, and the specification also allows the modification of such events in the cache, making a copy is necessary to avoid the accidental modification of an event in the output queue, by making a modification to the same event in the cache).

In real-time runs, the profile shows, that *ace* spends most of its time sleeping.

6.3 Evaluation with Random Events

To evaluate the time and memory complexity of the correlation engine, several tests with random events were done. The scripts used throughout this Section can be found in the directory `evaluation`.

6.3.1 Rule Execution Time

The script `plot_rules_runtime.py` was used to evaluate the impact of the number of rules on the processing time used for an individual event. The script generates between 0 and 1000 rule groups, each with one rule, which is triggered by all events. The script then lets the correlation engine process 100 random events for each step.

As shown in Figure 6.1, the processing time required for an individual event is linear to the number of rules. While the time needed to process a single event with 1000 rules in the repository is, with 0.1 second, quite high, it should be noted that such a situation is unlikely to arise in a practical setting, as each rule should normally only apply to a small fraction of the events, and even with hundreds of rules in the repository, only a few rules should apply to a specific event. Furthermore, as long as compression and filtering is done early, events arriving with a high rate should usually only trigger these rules.

[1] This is especially true for the conversion of a whole event into a string. Each time an event needs to be converted to a string (e.g. because it appears in a format string), Python calls the `__str__()` function of the `Event` class, which then generates a string representing the event.

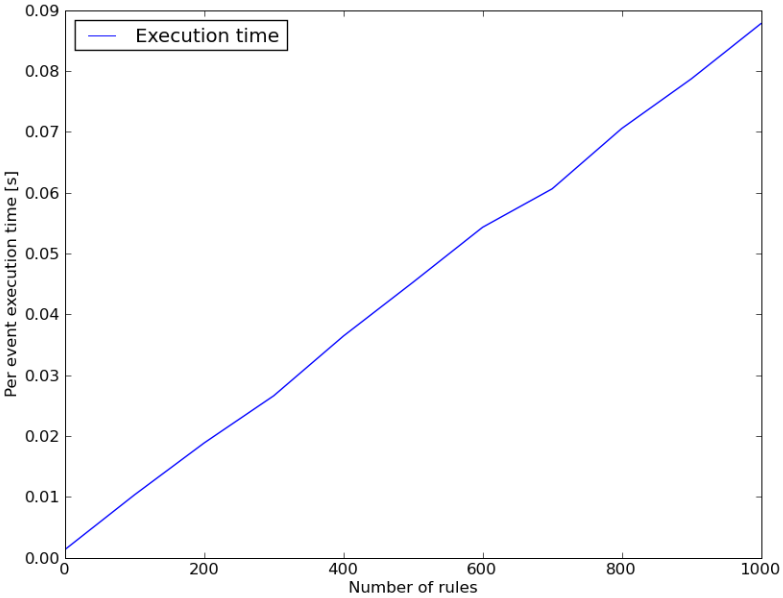


Figure 6.1: Event processing times with various numbers of relevant rules.

6.3.2 Evaluation of the Cache

The insertion of a new event into the cache consists of adding the event to the `events` set of the cache, and the insertion of appropriate timestamps and references into the delay and cache lists (cf. Section 5.3.13). As the addition of an event to a set can be done very fast (only a hash and a reference have to be added), the insertion of timestamps into the delay and cache lists is likely the determining factor for the total insertion time. As the lists are permanently kept in a sorted state, the corresponding function can use a binary divide-and-conquer algorithm, rather than searching through the whole list. From intuition, we would thus except, that the whole operation has a complexity of $O(\log_2 n)$, with n being the number of events in the cache.

The plot in Figure 6.2 (generated by the script `plot_cache_speed.py` shows the insertion time for a single event (the test inserts 1000 events and averages the result) into a cache containing between 0 and 100'000 events. This plot contradicts the expectation, as the insertion time grows linear with the number of events in the cache.

Profiling however confirms, that an overwhelming amount of the time is indeed spent in the `insort_right` function (which is responsible for inserting the timestamp into the cache and delay lists) from Pythons `bisect` module.

A look at the code of this function further reveals, that a binary divide-and-conquer algorithm is indeed used in this function:

```
def insort_right(a, x, lo=0, hi=None):
    if lo < 0:
        raise ValueError('lo must be non-negative')
    if hi is None:
        hi = len(a)
```

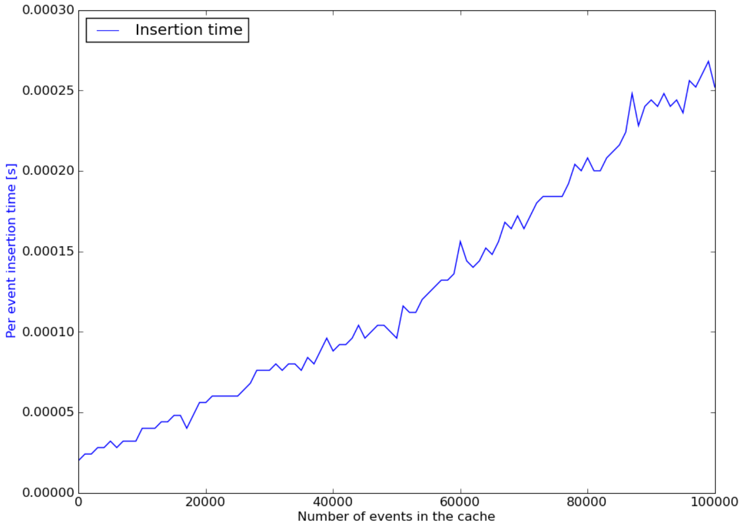


Figure 6.2: Event insertion times into a cache containing between 0 and 100'000 events.

```
    while lo < hi:
        mid = (lo+hi)//2
        if x < a[mid]: hi = mid
        else: lo = mid+1
    a.insert(lo, x)
```

The answer lies in the list insert operation in the last line of this listing: since the list data structure is array-based, all references behind the index, where the new element is inserted, have to be moved, and the insertion operation thus has $O(n)$ complexity.

The solution is thus to use a data type with better insertion complexity, such as a tree structure. The `blist` [60] provides a replacement for the built-in list data type, based on bushy trees. The `blist` provides $O(\log_2 n)$ `insert` and `delete` operations at the cost of slower `get` and `append` methods and a larger memory requirement (a more comprehensive comparison is provided in [60]). Since the cache uses mostly `insert` and `pop` operations, this is a favorable trade-off. The cache was thus changed to use the `blist` type (since `blist` is not in the Python standard library, the usual `list` type is still used as a fall back).

With these changes, the insertion complexity now looks much better, as shown in Figure 6.3. The plot also shows minor page faults (which indicate, that the OS needs to allocate more memory, which was earlier requested by Python), as an explanation for the occasional spikes in the insertion time.

When removing events from the cache (which consists of removing the event from the events set and removing the cache and delay timestamps from the corresponding lists), a similar picture can be seen, as show in Figure 6.4.

As expected, the memory usage also grows linearly with the number of events in the cache, and is slightly larger than the size of the events itself (as the cache also needs to save the cache and delay timestamps). Figure 6.5 shows the results of an evaluation with the script `plot_cache_mem.py`.

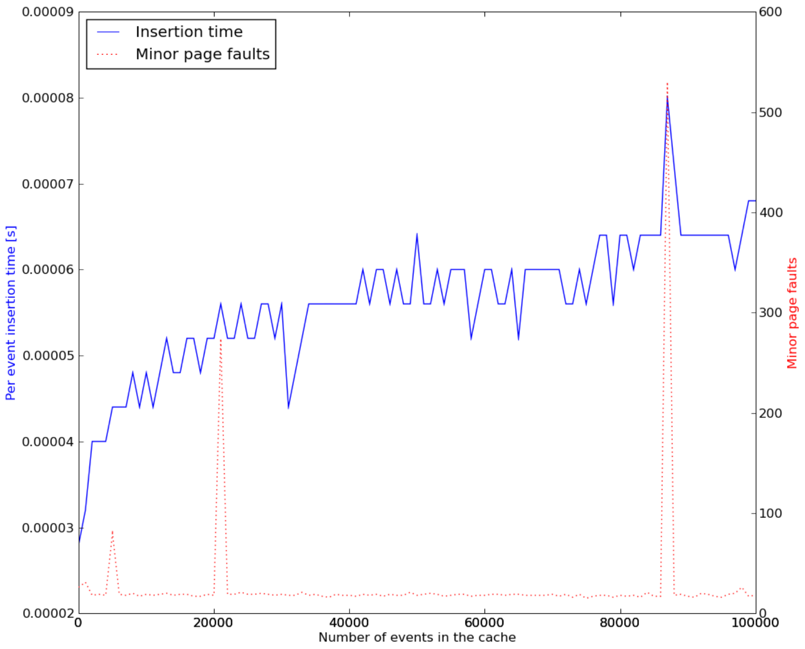


Figure 6.3: New (faster) insertion times for a cache using the `blist` data type.

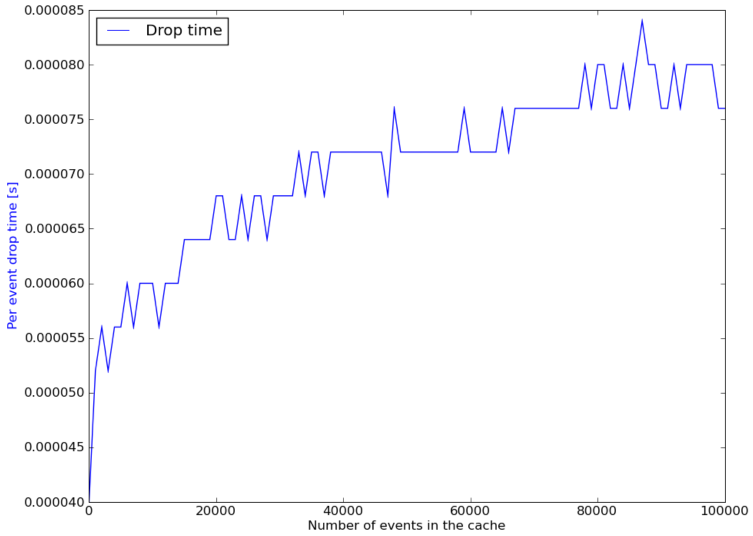


Figure 6.4: Per event time to drop an event from the cache.

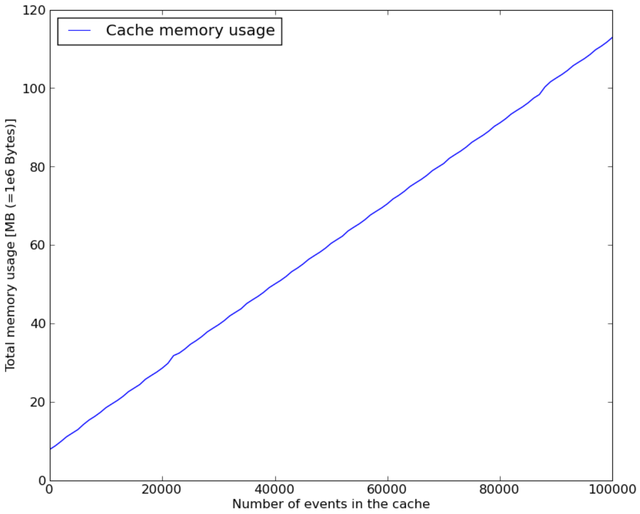


Figure 6.5: Memory usage of a cache containing between 0 and 100’000 events.

6.4 Evaluation with Real-world Events

More interesting than the evaluation with random events and dummy rules is of course the evaluation with real events and meaningful rules. The correlation engine was thus tested with replayed real log messages from two different months.

As each pattern requires corresponding correlation rules to detect the pattern, it would not be possible in due time, to write correlation rules for all patterns. For this reason, a set of frequent patterns was tested only. This section presents some of the results.

The rules used throughout this Section can be found in the directory `evaluation-rules` on the CD-ROM.

6.4.1 Compression

One of the first correlation steps is filtering and compression. For compression, the `compress` rule element can be used. As more than one event is needed for compression, events need to be delayed for some time to allow compression.

In a test run, a compression of the 5 most frequent events with a time window of 2 minutes led to a reduction of the number of events of about 50%.

In practice, the information loss inherent in compression may not always be acceptable. In this case, aggregation can be done in the same way as compression, using the element `aggregate` instead of `compress`.

6.4.2 Changing Bursts to Start/End Signaling

As explained in Section 2.3.2, it is sometimes useful to generate only one event when a burst starts and one when a burst ends, rather than forwarding all events of the burst. This can be achieved as follows (assuming we want to change the signaling of events `A`):

1. When the first event `A` from a specific host arrives:
   - Create a context (specific to that host).
   - Create an `A:FIRST` event.
2. As long as the context exists, drop further `A` events from the same host.
3. When the context times out, create an `A:LAST` event.

In a practical test (applied to `WINBIND:CONF:ADCONN` events), this approach worked without problems. It can however only be applied, when the information contained in the individual events is of no interest, as the events are (intentionally) lost.

6.4.3 Aggregation of Old Events

As discussed in Section 2.3.3, another application is the aggregation of old (outdated) events. This can be achieved with the `event_min_age` rule element, which allows the specification of a condition on an events age at arrival (the element compares the events creation and arrival timestamps).

In a test run, a rule, which created a context, when the first event older than a day arrived, and aggregated all old events after the context timeout 10 minutes later, worked without problems.

6.4.4 Irrelevant Unique Events

Sometimes, it is desirable to drop a given event, if it arrives only once (cf. Section 2.3.5). This can again be achieved with a context, which is created, when the first event arrives. When an event arrives, and no context exists yet, the event can be dropped, otherwise, the event is forwarded.

While this approach worked in tests, it was of limited use, as the cases, where only exactly one event of the targeted kind was created, were rare. Furthermore, the rule is also possibly dangerous – if events are generated at an interval, which is slightly larger than the contexts timeout, all events are dropped. It might thus make sense to use a second context to count the dropped events over a larger time period.

6.4.5 Flickering Detection

As discussed in Section 2.3.6, a short flickering of a service or an other component can sometimes be seen. An example is the pair `NIC:ETHERNET:LINKDOWN` and `NIC:ETHERNET:LINKUP`. Several such events from different hosts and interfaces are often generated in short succession, and it is thus helpful, if matching pairs (down and up event from the same host concerning the same interface) can be collected (rather than having to look for the corresponding up event for numerous down events manually). This can be done in a two-step process:

1. If a `NIC:ETHERNET:LINKDOWN` or a `NIC:ETHERNET:LINKUP` event arrives, enrich it with the information about the corresponding interface, extracted from the log message with a regular expression (using the `EnrichRegexp` plugin).

2. If a `NIC:ETHERNET:LINKUP` event arrived, check whether there was a (not yet correlated) down event from the same host, concerning the same interface during the last 10 seconds, and aggregate or suppress the two events (all `NIC:ETHERNET:LINKDOWN` events are delayed for up to 10 seconds, to allow suppression).

In a test with real-live events, this correlation behaviour worked without any problems.

As there is sometimes a distance of more than 10 seconds between the two events, a third rule was however added, to detect pairs within a window of up to 10 minutes. As it is not desirable to delay events that long, this rule did not suppress the original events, but rather just created a new event with references to the two events in the pair. In that case, it is the task of the front-end to update the presentation of the events, as soon as the new event arrives (this behaviour is implemented in the demonstration event sink web front-end).

6.4.6 Suppression of Dependent Events

In Section 2.3.7, we discussed the requirement to suppress dependent events, to help the operator find the root-cause of a problem. This was tested on two practical examples.

6.4.6.1 IP theft events

IP theft events indicate, that there is another host, which is using the same IP address as the host sending the event. While this would usually constitute a serious problem, which needs investigation, it is an expected problem in some cases. An example is the case, where the slave in a hot standby firewall configuration becomes master, while the master is still active (e.g. because the master is unresponsive due to high load). In this case, both firewalls try to use the same IP address, resulting in IP theft events. As an event for the duplicate hot standby master is also created, the IP theft events are of no interest (solving the root-problem, i.e. removing the duplicate master, will also solve the IP theft problem).

Ideally, this problem could be treated by simply creating a context when the event indicating the duplicate master arrives, and suppressing IP theft events, as long as the context exists. As an event also arrives, when there is no more duplicate master, the context can be removed upon this event. Unfortunately, the IP theft is often detected before the duplicate master, and the corresponding events arrive first. This means, that another rule is needed, to suppress the IP theft events from e.g. the last minute, when the duplicate master event arrives (the IP theft events thus have to be delayed for one minute, as we do not know at the time of their arrival, whether a duplicate master event will arrive in the future).

With this extension, the suppression of the IP theft events works well. This example however shows, that the creation of correlation rules is often an incremental task, which does not succeed at the first try. On the other hand, if a correlation rule group does not match all targeted event patterns, there is usually a way to extend the rules, to detect all patterns.

6.4.6.2 Reboots

Since reboots routinely cause certain events, a suppression of these events in the context of a reboot is desirable. An example is the suppression of `NIC:ETHERNET:LINKDOWN` and `NIC:-ETHERNET:LINKUP` events. This can be done in a two-step process:

1. Collect `NIC:ETHERNET:LINKDOWN` and `NIC:ETHERNET:LINKUP` pairs, as described above (Section 6.4.5).

2. Suppress these aggregated events, if there was a reboot event in the last minute.

In a practical test, this approach worked without problems.

6.4.6.3 Complex Patterns

As an example for a more complex pattern, the detection of a successful Virtual Router Redundancy Protocol (VRRP) transition and the corresponding return to normal was tested. The transition usually consists of the three events for the first transition and another three events for the transition back to normal. While a first test with a simple regular expression on these events worked in some cases, it could not detect the pattern in other cases, because the events within both groups are not always created in the same order.

While it is possible to simply specify all six permutations for each group in the regular expression, this leads to quite a long expression. Luckily, the used regular expressions also allow look-ahead assertions, which can be used to specify a pattern without having to mention each permutation individually. With this update, it was possible to detect complete VRRP transitions (including the transition back to normal) independently of the event order inside the two groups.

As a VRRP transition usually includes further aspects (such as e.g. an ISP outage, which caused the transition in the first place), the rules would however likely have to be extended for practical use.

6.4.7 Speed Considerations

Processing all events from a whole month (about 40'000 to 60'000 events) with the example rules usually took the correlation engine less than 10 minutes. While a practical setting would likely require many more rules, than were used in these test runs, the used rules were usually able to reduce the number of events considerably. For this reason, it can be safely assumed, that real-time operation in a practical setting would be possible, especially as the tests were mostly done using only a single node only.

6.4.8 Real-time Testing

For a test in real-time mode, the timestamps of events from one month were rewritten, so the events appeared to have been generated within one hour. Processing these events in both simulation and real-time mode (with the `simulation` option set to *false* and the `realtime` option set to *true*, as explained in Section 5.3.8) led to the same results in both cases, except that the output events were sometimes ordered in a slightly different way. This happens, because it may not be possible to process all events arriving within one second during the same second, and an internally created event may thus have a slightly higher arrival time in real-time mode, than it would have in simulation mode. Furthermore, the test also revealed, that events created within the same second are not always sorted in the same order in the output (as the time is not stored to a resolution higher than one second, and the sorting depends on the events themselves, including their random id, for events arriving during the same second).

While it would be possible to change the behaviour to have the same output event order in both cases, this would be a change with far reaching (and possibly unforeseen and unnoticed) consequences, and was not considered worth the risk of breaking functionality.

6.4.9 Conclusions

The tests runs show, that the patterns identified in Section 2.3 can generally be correlated. As the prototype was implemented and evaluated in a rather short time, more testing is however certainly needed.

A production implementation further requires more plugins, especially action plugins for enrichment of the events with information from data bases, such as:

- Enrichment with the information about the ISP link for a given event, to detect widespread ISP outages.
- Enrichment with information about the name of the master and slave in the cluster, to which an event's host belongs (for testing, this information was extracted from the semantics of the host name, which is however not always possible).
- Enrichment of events with location information for the events host, e.g. to detect a location dependent temperature problem (with location information available, this could be done with a threshold on the number of high-temperature events from the same location, but different hosts).

With such plugins, further correlation would be possible.

Chapter 7

Conclusions and Outlook

7.1 Conclusions

The evaluation of the implemented prototype shows, that the chosen approach is well suited to correlate the targeted event patterns. In Section 6.4, representative cases for most of the patterns explained in Section Section 2.3 could be correlated, and the creation of simple additional plugins should allow the coverage of further patterns. The evaluation also showed, that the prototype is able to process events in real-time without any problems.

The implementation of the prototype also confirms, that functional programming is well suited to build a correlation engine more elegantly, than having to use string evaluation or similar constructs. FSMs can be nicely incorporated into the correlation engine by allowing regular expression matches on events, which are both easier to use and – due to modern extensions to regular languages, such as look-around assertions – actually more powerful than FSMs. The use of plugins as the only interaction with external factors (besides the events themselves) makes a clear separation between events and external factors. This makes the correlation process more transparent, and should also facilitate future developments.

Last but not least, the implementation of the correlation engine as a homogeneous node – without any separation between agent or central node – allows us to use the engine in any structure, as a single node, as a central node with "agents" for preprocessing, or as a complete tree, with any number of processing steps between leafs and the root-node.

7.2 Outlook and Future Developments

7.2.1 Rule Generation

As the correlated event patterns can be rather complex, so is the writing of correlation rules. Writing new rules can thus be a challenging task for the human operator, and usually requires intimate knowledge of the correlated events. While this is difficult to avoid, there are various possibilities to support the operator.

As a first step, a good XML editor can already make the creation of rules more efficient. Having a customized tool to create new rules might be even more helpful.

An interesting challenge is certainly the mining of collected events to suggest new rules. Some techniques for automated log clustering and pattern mining are described in [65].

7.2.2 Central Rule Repository

Currently, the correlation rules are always loaded from a local file. While it is possible to use a Version Control System (VCS) to centrally store the rules, this would still require an individual rule file for each node (unless two nodes share *exactly* the same set of rules).

As mentioned in Section 4.7.3, a preferable solution would be to have a central data base with rules, and a corresponding scope for each rule (e.g. "this rule is valid for nodes, which directly correlate events from hosts of company X").

The implementation of such a feature should be technically straightforward, and might be useful especially in a setup with many correlation nodes.

7.2.3 Automatic Rule Destination Selection

Currently, the operator must decide, on which node a given rule group is executed. Since the only difference between the nodes is the set of child nodes, whose events a given node can see, it should be possible to decide about the optimal target node for a given rule group automatically.

This could be done in a way, similar to how the cache and delay times for events are determined, by looking at the queries and deciding, which hosts are relevant for a given rule group. The rule groups could then be pushed from the root node towards the leaf nodes, by simply letting each node push a given rule group towards one if its child nodes, as long as the child node still can see all the required hosts.

A caveat here however are the plugins, which may depend on a specific property of the host, on which they are executed (such as the availability of a specific external script).

Appendix A

Notes on Measuring Event Rates

A recurring requirement is the measurement of the event rate, or – more often – the specification of a threshold for the event rate. If the specified threshold is exceeded, another event is generated, or some action is triggered. Although the term event rate seems to be clearly defined, there are several ways to measure it, with different results. The language is often unclear – for instance "at most 100 events per day" is likely to be interpreted as "no more than 100 events between 0.00 and 23.59 of any day", whereas "at most 100 events per 24 hours" could just as well be interpreted as "no more than 100 events during *any* time window of 24 hours".

In this section, the different approaches to measuring event rates will be discussed, particularly with respect to their fitness for measuring network and log event rates. Furthermore, the memory requirements of each method will be discussed. As an event correlation engine may be required to measure a large number of event rates, with possibly long measuring windows for each event rate, a careful consideration of the memory usage is quite important.

The computational complexity is generally a linear function of the event rate and will not be examined further.

Throughout the whole section, the variable r will be used for the event rate (events per time unit), r_t for the threshold event rate (events per time unit), w for the window length (time units), b for the number of bins[1], c for an event count and t for the time. The function $O(.)$ designates the asymptotic memory complexity of each algorithm.

A.1 Sliding Window

One way to determine the event rate is to observe a time window of a given duration w and count the events inside this window. This can be done discretely only at specific points in time (fixed window) or continuously[2] for every point in time (sliding window). In either case, the event rate is $r[t] = \frac{c[t]}{w}$, where $c[t]$ is the number of events inside the specific window instance at time t.

A reasonably long window size is generally assumed; in particular, $w > \frac{1}{r_t}$ is required for a threshold decision, as *each* event will otherwise trigger the corresponding action.

If the event count inside the window is evaluated for every possible time point, the procedure can be thought of as having a window of a fixed length that is moving over the input data stream[3],

[1] If the window is split into equally sized parts, each part is called a bin.

[2] Time is of course always discreet inside a computer. In this context, continuous simply means that the window step size is as small as the time-resolution of the events.

[3] In the case of an event correlation engine, it is actually more correct to think of an input data stream that is moving through the window.

while the events inside the window are continuously counted[1]. Formally, the sliding window W at time t can be defined as the interval $W[t] = [t, t + w)$. More casually, this is a window of length w starting at time t. The event count is then $c[t] = |\{e \in E|\, \mathrm{eventtime}(e) \in W[t]\}|$, where E is the set of events and $\mathrm{eventtime}(e)$ is the time, when $e \in E$ was generated.[2]

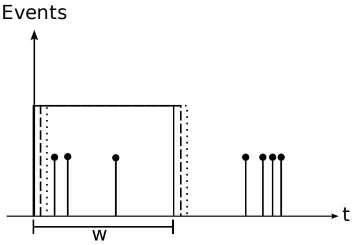


Figure A.1: First three instances of a sliding window.

The window length determines, how much averaging is done. With a short window, the values for the event rate are more locally correct, whereas a long window yields more globally correct values. Which window length is sensible, depends on the application. Statements, such as “at least 100 events in 30 minutes”, usually imply that a 30-minute window is used. In some cases, more than one threshold and window length may be appropriate, e.g. to specify a threshold for reboots per day and another one per week.

Assuming, that for a given event rate threshold, the goal is to decide whether the number of events in *any* time window of the specified length reaches the threshold, the answer found with a sliding window is the most correct one. Therefore, the sliding window approach will be used as the reference for a comparison with the other methods.

A.1.1 Memory Usage

For the examination of memory usage throughout this chapter, it will be assumed, that the events arrive in an ordered fashion (i.e., the events arrive at the correlation engine in the same order, in which they were generated); even though this is not necessarily a realistic assumption. If the events arrive in an unordered fashion, more memory is required, to cache and sort the events (the amount of required memory depends on the maximum possible delay for each individual event). Alternatively, the arrival time of each event, rather than the creation time, could be used for the threshold.

Assuming the events are ordered, the memory complexity of a sliding window, used to decide whether a threshold r_t is reached, is $O(r_t \cdot w)$, as at most $r_t \cdot w$ events have to be kept in memory (e.g. if a sliding window of one hour is used to determine whether a threshold of 100 events per hour is exceeded, at most 100 events need to be remembered; after that, the threshold is reached and as long as there are at least 100 events inside the window, an old event can be removed from the memory whenever a new one arrives, even if the old event is still inside the window). Choosing a larger window leads to averaging over a longer period, but also requires more memory.

[1] The actual implementation only needs to keep track of events moving inside or outside of the window, rather than counting the events at each step; otherwise, the computational complexity would become worse than linear.

[2] This definition assumes knowledge about the future. In practice, the window would either have to be defined as $(t - w, t]$, or the output has to be delayed by a time w (which is both essentially the same thing).

If a sliding window is used to measure the event rate, rather than to decide about a threshold, the memory complexity depends on the maximum event rate, and is generally not predictable.

A.2 Fixed Window

Instead of moving the window continuously, the fixed window approach only considers windows with a time offset that is an integer multiple of the window length. Formally, the window is the interval $W[t] = [t', t' + w)$, with $t' = \lfloor \frac{t}{w} \rfloor \cdot w$ and the same event count definition as above[1].

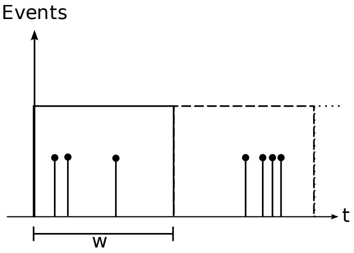


Figure A.2: First three instances of a fixed window.

Using a fixed window is often sensible, when there is an externally imposed separation of the time windows (e.g. it is natural to associate 24 hour time windows with days, as they are separated by nights).

A.2.1 Memory Usage

The fixed window approach requires a lot less memory than the sliding window approach – rather then keeping track of all events, only one event count for the current window needs to be remembered. The memory complexity is therefore $O(1)$.

A.2.2 Comparison to the Sliding Window

The question can now be asked, how the rate values measured with a fixed window compare to values measured with a sliding window of the same length, and in particular, how much the measured maxima for the event rates differ in the worst case (as this determines, whether an event for a specified threshold will be triggered).

It is clear that maximum event rate measured with a sliding window, $\max_t r_{slide}[t]$, is at least as big as the maximum measured with a fixed window, $\max_t r_{fixed}[t]$, in any event stream, because for every instance of the fixed window, there is an instance of the sliding window at exactly the same position (the same argument also holds for the minimum values).

On the other hand, $\max_t r_{slide}[t]$ can be up to twice as large as $\max_t r_{fixed}[t]$. This is the case if two fixed window instances split an isolated event cloud exactly in half (a worse case is not possible, due to the fact, that any instance of the sliding window is always encompassed by at most two consequent fixed window instances, and one of them always contains at least half of the events).

[1] $\lfloor . \rfloor$ designates the floor operation. $\lfloor x \rfloor$ is defined as the largest integer smaller or equal to x.

A.3 Fixed Window with Dynamic Start

As a special case of the fixed window, a window of a fixed length, which starts when an event arrives, could be used. This method is shown in Figure A.3.

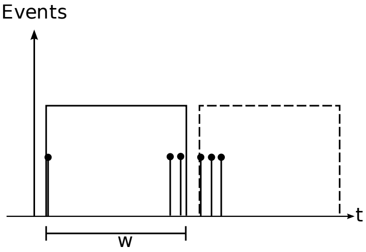


Figure A.3: Fixed window with a dynamic start.

A.3.1 Memory Usage

The memory usage is the same as for the general case of the fixed window.

A.3.2 Comparison to the Sliding Window

The drawing in Figure A.3 also illustrates, that a case can be constructed, where a fixed window with a dynamic start performs almost as bad as a fixed window with a fixed start. The only difference is that one event has to be "sacrificed" to let the fixed window start at a bad position. This is a benefit only when there are few events.

Although the worst case is not mitigated, it can be expected that in the average case, a dynamic start is an advantage. Specifically, if only bursts shorter than the window length are considered, a fixed window with a dynamic start performs as good as a sliding window, as the possibility, that the burst is split by the window, is eliminated, and the burst is always fully encompassed by the window.

A.4 Stepping Window

By separating a fixed window into b separate bins, a trade off between the sliding and the fixed window can be created. Separating the window into bins results in a window that steps over the input data in steps that are an integer fraction of the window size. Formally, the window is the interval $W[t] = [t', t' + w)$, with $t' = \lfloor \frac{t \cdot b}{w} \rfloor \cdot \frac{w}{b}$.

As an example, a fixed window of one day could be separated into 24 one-hour bins. The step size is then one hour and 24 event count values have to be kept track of.

A.4.1 Memory usage

Only b event counts have to be kept in memory at any given moment. The memory complexity is therefore $O(b)$.

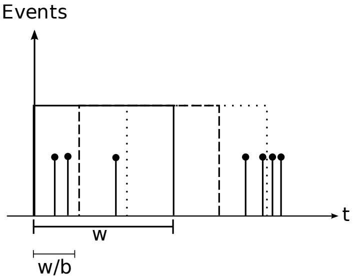


Figure A.4: First three instances of a stepping window with three bins.

A.4.2 Comparison to the Sliding Window

With the same argument as in the case of the fixed window, it can be said that $\max_t r_{step}[t] \leq \max_t r_{slide}[t]$ for any part of the event stream. On the other hand, each instance of a sliding window is fully encompassed by two consequent instances of the stepping window. Again, it can be said that one of these two instances has to contain at least half of the events contained in the sliding window instance.

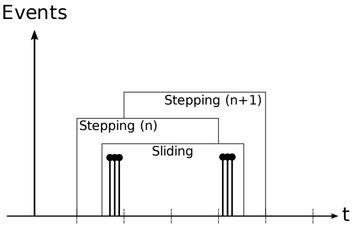


Figure A.5: Problem case for the stepping window: The sliding window contains six events, but no instance of the stepping window contains more than three events.

The worst case is therefore the same as for the fixed window. As Figure A.5 illustrates, no matter how many bins there are, a case can always be constructed, where there are twice as many events in the sliding window as in any instance of the stepping window. Still, the average result can be expected to be closer to the result found with a sliding window; with more bins leading to better results. This can be illustrated with the following reasoning: If only one event burst with evenly spaced events of a reasonable number (specifically, more events than bins) is considered, bursts which have the same length as the window are the most problematic case. A sliding window will always have one instance that contains the whole burst. The fixed window contains only 50% of the events in the worst case, 75% in average. With two bins, one bin contains 50% of the events, and another one at least 25% (if the bin left of the "full" bin contains less then 25% of the events, the one on the right contains more, and vice versa), i.e. the worst case is 75%, with an average of 87.5% (uniform probability distribution). With three bins, two bins are certain to contain $\frac{1}{3}$, and a third one at least $\frac{1}{6}$ of the burst, resulting in a worst case of $\approx$ 83%, etc. In the worst case, an event stream of the length of the window size, with evenly spaced events, that is observed with a stepping window with b bins, will result in an event rate that is $\frac{2b-1}{2b} r_{slide}$.

Another interesting possibility of a stepping window is the use of an additional bin. The window is then $W[t] = [t', t' + \frac{b+1}{b} \cdot w)$, with $t' = \lfloor \frac{t \cdot b}{w} \rfloor \cdot \frac{w}{b}$. Now, each instance of the sliding window is fully encompassed by some instance of the stepping window, and each instance of the stepping window is fully encompassed by two instances of the sliding window. The arguments above therefore can be applied in the reverse direction, resulting in the relation $\max r_{slide} \leq \max r_{step} \leq 2 \cdot \max r_{slide}$. Using an additional bin can thus help to find a better upper bound.

A.5 Overlapping Stepping Windows

Another idea might be to use n independent stepping windows with b bins, each shifted in time by b/n against the previous one (i.e. overlapping). The results are however the same as when a single stepping window with $b * n$ bins is used (which also has the same memory usage), as there are the same window instances at the same places in both cases. The idea is shown in Figure A.6.

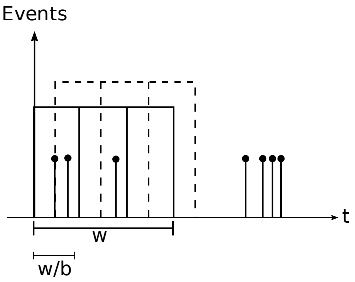


Figure A.6: Two overlapping stepping windows with three bins each.

A.6 Event Distance

Another possible approach to measure the event rate is to measure the time distance $d[t]$ of an event to the next one. The event rate is $r[t] = \frac{1}{d[t]}$. The measured rate is locally correct, i.e. it is correct at the exact moment when it is measured.

The method is similar to using a very short sliding window; in fact, if the goal is to decide, if a given rate threshold r_t is reached, measuring the event distance will yield the same results as using a sliding window of a duration infinitesimally larger[1] than $w = \frac{1}{r_t}$ (e.g., if the threshold is 60 events per minute, the condition is satisfied if two events appear within no more than one second with both methods)[2].

Unfortunately, measuring the event distance is unsuitable for log or network events, as the events arrive rather randomly, and the measured event rate would be unpredictable.

[1] The duration of the window is chosen infinitesimally larger to avoid that one event inside the window satisfies the condition.

[2] Depending on the implementation, the two methods may however yield different results for the moment when the rate falls below the threshold.

A.6.1 Memory usage

For this method, only the last occurring event has to be remembered. The memory complexity is therefore $O(1)$.

A.6.2 Comparison to the Sliding Window

If the events arrive evenly spaced, the event rate measured with the event distance is approximately the same as measured with a sliding window[1]. If the events are not arbitrarily spaced, there has to be at least one t' where there are two events with a distance $d[t'] < \frac{1}{r_{slide}}$, therefore $\max_t r_{dist}[t] > \max_t r_{slide}[t]$. On the other hand, events can be arbitrarily close, and therefore, $\max_t r_{dist}[t]$ can become arbitrarily high.

A.7 Dynamic Window

As a last approach, a window could be specified to begin with the first event, and end a certain time after the last event of a burst (i.e. there is a timeout to close the window, which is reset by every new event).

A.7.1 Memory Usage

As only one event count, as well as the time of the last event have to be remembered at any time, the memory usage is $O(1)$.

A.7.2 Comparison to the Sliding Window

The results depend heavily on the specified timeout T to close the window. The measured rate is always either zero (if there are no events), or at least $\frac{1}{T}$, as the space between consecutive events in the window is always smaller than, or equal to T (otherwise, the window is closed).

Unfortunately, the results are otherwise rather unpredictable. For instance, if there are multiple short bursts, with a spacing between the bursts, which is larger than the timeout, the measured event rate would jump between zero and rather high values. As burst parameters, such as the spacing between two bursts, are usually not known, such problems are hard to avoid[2].

Another problem, especially if a large timeout is specified, is that there is no upper limit for the window length.

A.8 Summary

Which implementation is preferable depends on what is measured. In many cases, a sliding window is the most suitable method; however it has the largest memory usage as well. A method with a smaller memory usage is the use of a fixed window, possibly with a dynamic start time. A dynamic start time is useful especially when the event bursts are shorter than the window length. A compromise between these two methods is the stepping window.

[1] For a reasonably long window. If the window is almost as short as the event distance, the measured rate values differ up to a factor of two; for window lengths smaller than the event distance, the rate measured with the sliding window can become arbitrarily high.

[2] Furthermore, even if there is a way to avoid certain problems, adding more complexity to the measuring method is likely to cause unpredictable measurements in other cases. If in doubt, the simplest measuring method should always be preferred.

Using the event distance or a dynamic window is not a good idea for network events, as the results can be unpredictable with these methods.

Table A.1 lists the memory usage required to decide about a rate threshold r_t with a window duration w and b bins, as well as the worst-case error of the measured maximum event rate, relative to the maximum event rate measured with a sliding window.

Algorithm	Memory usage	Relation of $\max r$ to $\max r_{slide}$
Sliding window	$O(t \cdot w)$	$\max r = \max r_{slide}$ (reference value)
Fixed window	$O(1)$	$\frac{\max r_{slide}}{2} \leq \max r_{fixed} \leq \max r_{slide}$
Fixed window, dynamic start	$O(1)$	$\frac{\max r_{slide}}{2} \leq \max r_{fixed} \leq \max r_{slide}$
Stepping window, b bins	$O(b)$	$\frac{\max r_{slide}}{2} \leq \max r_{step} \leq \max r_{slide}$
Stepping window, $b+1$ bins	$O(b+1) = O(b)$	$\max r_{slide} \leq \max r_{step} \leq 2 \cdot \max r_{slide}$
Event distance	$O(1)$	$\max r_{slide} \lesssim \max r_{dist} < \infty$
Dynamic Window	$O(1)$	$(r \geq \frac{1}{\text{timeout}})$

Table A.1: Overview of rate measuring approaches.

Appendix B

XML Document Type Definitions and Examples

This section contains the DTDs and examples of corresponding XML documents.

B.1 Events

B.1.1 Document Type Definition

The following listing shows the DTD for events:

```
<!-- root element: list of events -->
<!ELEMENT events (event*)>

<!-- level 1: event -->
<!ELEMENT event (name,description,id,type,status,count?,host,creation,
                 attributes?,references?,history?)>

<!-- level 2 -->
<!ELEMENT name (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT type (#PCDATA)>
<!ELEMENT status (#PCDATA)>
<!ELEMENT count (#PCDATA)>
<!ELEMENT host (#PCDATA)>
<!ELEMENT creation (#PCDATA)>
<!ELEMENT attributes (attribute)*>
<!ELEMENT references (reference)*>
<!ELEMENT history (historyentry)*>

<!-- level 3 -->
<!ELEMENT attribute (#PCDATA)>
<!ATTLIST attribute key CDATA #REQUIRED>
<!ELEMENT reference (#PCDATA)>
<!ATTLIST reference type (parent|child|cross) #REQUIRED>
<!ELEMENT historyentry (rule,host,timestamp,fields?,reason?)>

<!-- level 4 -->
<!ELEMENT rule (groupname,rulename)>
<!ELEMENT timestamp (#PCDATA)>
<!ELEMENT fields (field*)>
```

```
<!ELEMENT reason (#PCDATA)>

<!-- level 5 -->
<!ELEMENT groupname (#PCDATA)>
<!ELEMENT rulename (#PCDATA)>
<!ELEMENT field (#PCDATA)>
```

B.1.2 Examples

An example of some independent, fictional events, which are valid against the DTD specified above, can be seen in the following listing:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE events SYSTEM "events.dtd">
<!-- some independent, fictional events -->
<events>
  <event>
    <name>NIC:ETHERNET:LINKUP</name>
    <description>The ethernet network internet controller is up.</description>
    <id>e9294e806d02fd8ebd90e345434c16a3</id>
    <type>raw</type>
    <status>active</status>
    <host>host-a-0</host>
    <creation>1243039102</creation>
    <attributes>
      <attribute key="interface">1</attribute>
    </attributes>
  </event>
  <event>
    <name>MAIL:FRESHCLAM:ERROR</name>
    <description>The anti-virus signatures could not be updated.</description>
    <id>ae84e1e89470ddd1ecaf33ffb1b7538f</id>
    <type>raw</type>
    <status>inactive</status>
    <host>host-b-0</host>
    <creation>1244014810</creation>
    <history>
      <historyentry>
        <rule>
          <groupname>freshclam</groupname>
          <rulename>detect-single-events</rulename>
        </rule>
        <host>host-b-0</host>
        <timestamp>1244014940</timestamp>
        <fields><field>status</field></fields>
        <reason>Single errors can be ignored.</reason>
      </historyentry>
    </history>
  </event>
</events>

<!--
vim: sw=2 ts=2
-->
```

B.2 Line-based Input Translation

B.2.1 Document Type Definition

The following listing shows the DTD for line-based input translation:

```
<!-- root element: list of matches -->
<!ELEMENT translation_linebased (match)*>

<!-- level 1: match -->
<!ELEMENT match (match|description|host|attribute|datetime|create|drop)*>
<!ATTLIST match regexp CDATA #REQUIRED>

<!-- level 2: match element -->
<!-- event description -->
<!ELEMENT description (#PCDATA|matchgroup)*>
<!ELEMENT host (#PCDATA|matchgroup)*>
<!ELEMENT attribute (#PCDATA|matchgroup)*>
<!ATTLIST attribute name CDATA #REQUIRED>
<!ELEMENT datetime (#PCDATA|matchgroup)*>
<!ATTLIST datetime
    format              CDATA           #REQUIRED
    use_current_year    (true|false)    "false"
    >
<!-- actions -->
<!ELEMENT create (#PCDATA)>
<!ATTLIST create status (active|inactive) "active">
<!ELEMENT drop EMPTY>

<!-- level 3: matchgroup -->
<!ELEMENT matchgroup EMPTY>
<!ATTLIST matchgroup
    group CDATA #REQUIRED
    >
```

B.2.2 Examples

This input translation specification can be used for any line-based input, such as syslog messages. For instance, we might want to match SSHd log messages, such as the ones shown in the following listing:

```
Jun 15 09:50:33 server-002 sshd[4620]: Failed password for root from 10.0.2.68 port 52361 ssh2
Jun 15 09:50:35 server-002 sshd[4620]: Accepted password for root from 10.0.15.18 port 52411 ssh2
```

In that case, the following XML translation rules could be used, which also show how a catch-all match can be specified:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE translation_linebased SYSTEM "translation_linebased.dtd">

<translation_linebased>
  <!-- SSHd log messages -->
  <match regexp="^([^ ]+ \d{1,2} \d\d:\d\d:\d\d) ([^ ]+) sshd">
    <datetime format="%b %d %H:%M:%S" use_current_year="true">
      <matchgroup group="1"/>
    </datetime>
    <host><matchgroup group="2"/></host>
    <match regexp="Failed password for ([^ ]+) from ([^ ]+)">
```

```
      <attribute name="username"><matchgroup group="1"/></attribute>
      <attribute name="srchost"><matchgroup group="2"/></attribute>
      <description>An unsuccessful SSH login happened.</description>
      <create>SSH:LOGIN:FAILED</create>
    </match>
    <match regexp="Accepted password for ([^ ]+) from ([^ ]+)">
      <attribute name="username"><matchgroup group="1"/></attribute>
      <attribute name="srchost"><matchgroup group="2"/></attribute>
      <description>A successful SSH login happened.</description>
      <create>SSH:LOGIN:SUCCESS</create>
    </match>
  </match>

  <!-- catch all match -->
  <match regexp=".*">
    <description>An unknown log message.</description>
    <!-- match group 0 is the whole match: save it as attribute -->
    <attribute name="logline"><matchgroup group="0"/></attribute>
    <create>SYSLOG:UNKNOWN</create>
  </match>
</translation_linebased>
```

B.3 Rules

B.3.1 Document Type Definition

The following listing shows the DTD for rules:

```
<!-- root element -->
<!ELEMENT rules (group*)>

<!-- level 1: groups -->
<!ELEMENT group (rule+)>
<!ATTLIST group
    name                CDATA   #REQUIRED
    order               CDATA   #REQUIRED
    description         CDATA   #IMPLIED
    >

<!-- level 2: rules -->
<!ELEMENT rule (events,conditions?,actions,alternative_actions?)>
<!ATTLIST rule
    name                CDATA   #REQUIRED
    order               CDATA   #REQUIRED
    description         CDATA   #IMPLIED
    >

<!-- level 3: ECAA -->
<!-- events (rule triggers) -->
<!ELEMENT events (when_class|when_event|when_any)*>
<!-- conditions -->
<!ENTITY % conditions "and|or|not|context|trigger_match|count|sequence|
                       pattern|within|condition_plugin">
<!ELEMENT conditions (%conditions;)*>
<!-- actions (executed if conditions match) -->
<!ENTITY % actions "drop|forward|compress|aggregate|modify|modify_attribute|
                    suppress|associate_with_context|add_references|create|
                    create_context|delete_context|modify_context|action_plugin">
<!ELEMENT actions (subblock|select_events|%actions;)*>
<!-- alternative_actions (executed if conditions do not match) -->
```

```
<!ELEMENT alternative_actions (subblock|select_events|%actions;)*>

<!-- level 4 and below -->

<!-- events list -->
<!ELEMENT when_class (#PCDATA)>     <!-- a class of events -->
<!ATTLIST when_class type CDATA #IMPLIED>
<!ELEMENT when_event (#PCDATA)>     <!-- an event -->
<!ATTLIST when_event type CDATA #IMPLIED>
<!ELEMENT when_any EMPTY>           <!-- any event can trigger the rule -->
<!ATTLIST when_any type CDATA #IMPLIED>

<!-- conditions list -->
<!-- boolean expression construction -->
<!ELEMENT and (%conditions;)+>
<!ELEMENT or (%conditions;)+>
<!ELEMENT not (%conditions;)>
<!-- conditions on contexts -->
<!ELEMENT context (#PCDATA|trigger)*>
<!ATTLIST context
    counter      CDATA       #IMPLIED
    counter_op   (ge|le|eq)  "ge"
    group        CDATA       #IMPLIED
    >
<!-- conditions on the trigger -->
<!ELEMENT trigger_match (event_class|event_name|event_type|event_status|
                         event_host|event_attribute|event_min_age)*>
<!-- conditions on events -->
<!ELEMENT count (event_query)>
<!ATTLIST count
    threshold    CDATA       #REQUIRED
    op           (eq|ge|le)  "ge"
    >
<!ELEMENT sequence (event_query)+>
<!ATTLIST sequence
    sort_by      (creation|arrival)  "creation"
    match        (any|all)           "all"
    >
<!ELEMENT pattern (alphabet,regexp)>
<!ELEMENT alphabet (symbol)+>
<!ATTLIST alphabet sort_by (creation|arrival) "creation">
<!ELEMENT symbol (event_query)>
<!ATTLIST symbol letter CDATA #REQUIRED>
<!ELEMENT regexp (#PCDATA)>
<!ELEMENT within (event_query)+>
<!ATTLIST within
    timeframe    CDATA  #REQUIRED
    timeref      (creation|arrival)  "creation"
    match        (any|all)           "all"
    >
<!ELEMENT condition_plugin (plugin_parameter*,event_query*)>
<!ATTLIST condition_plugin name CDATA #REQUIRED>
<!ELEMENT plugin_parameter (#PCDATA)>
<!ATTLIST plugin_parameter name CDATA #REQUIRED>

<!-- actions/alternative_actions list -->
<!-- subblock (nested conditions-actions-alternative_actions block) -->
<!ELEMENT subblock (conditions,actions,alternative_actions?)>
<!-- event selection -->
<!ELEMENT select_events ((event_query),(%actions;)*)>
<!-- event operations -->
<!ELEMENT drop EMPTY>          <!-- drop events -->
```

```
<!ELEMENT forward EMPTY>          <!-- forward events -->
<!ELEMENT compress EMPTY>         <!-- replace multiple events by a count -->
<!ELEMENT aggregate (event)>      <!-- aggregate events -->
<!ELEMENT modify EMPTY>           <!-- change the event's status/local field -->
<!ATTLIST modify
    status  (active|inactive)  #IMPLIED
    local   (true|false)       #IMPLIED
    reason  CDATA              #IMPLIED
    >
<!ELEMENT modify_attribute (#PCDATA)> <!-- modify an event attribute -->
<!ATTLIST modify_attribute
    name    CDATA          #REQUIRED
    reason  CDATA          #IMPLIED
    op      (set|inc|dec)  "set"
    >
<!ELEMENT suppress (event_query)> <!-- supress the selected events -->
<!ATTLIST suppress reason CDATA #IMPLIED>
<!ELEMENT associate_with_context (#PCDATA|trigger)*>
<!ELEMENT add_references (event_query)>
<!ATTLIST add_references
    type    (child|parent|cross)   "cross"
    reason  CDATA                  #IMPLIED
    >
<!ELEMENT create (event)>         <!-- create a new event -->
<!-- context operations -->
<!ELEMENT create_context (context_name,event?)>
<!ATTLIST create_context
    timeout            CDATA          #REQUIRED
    counter            CDATA          #IMPLIED
    repeat             (true|false)   "false"
    delay_associated   (true|false)   "false"
    >
<!ELEMENT context_name (#PCDATA|trigger)*>
<!ELEMENT delete_context (#PCDATA|trigger)*>
<!ELEMENT modify_context (#PCDATA|trigger)*>
<!ATTLIST modify_context
    reset_timer               (true|false)    "false"
    reset_associated_events   (true|false)    "false"
    counter_value             CDATA           #IMPLIED
    counter_op                (set|inc|dec)   "set"
    >
<!-- action plugin -->
<!ELEMENT action_plugin (plugin_parameter*)>
<!ATTLIST action_plugin name CDATA #REQUIRED>
<!-- trigger -->
<!ELEMENT trigger EMPTY>
<!ATTLIST trigger field CDATA #REQUIRED>

<!-- event selection -->
<!ENTITY % query_operations "intersection|union|complement|first_of|last_of|
                             unique_by|is_trigger|in_context|match_query|
                             event_class|event_name|event_type|event_status|
                             event_host|event_attribute|event_min_age">
<!ELEMENT event_query (%query_operations;)*>
<!-- note: max_age is required in some cases, but not in others -->
<!ATTLIST event_query
    max_age        CDATA                 #IMPLIED
    delay          (true|false)          "false"
    time_source    (creation|arrival)    "arrival"
    name           CDATA                 #IMPLIED
    >
<!ELEMENT intersection (%query_operations;)*>
```

```
<!ELEMENT union (%query_operations;)*>
<!ELEMENT complement (%query_operations;)>
<!ELEMENT first_of (%query_operations;)>
<!ATTLIST first_of sort_by (creation|arrival) "creation">
<!ELEMENT last_of (%query_operations;)>
<!ATTLIST last_of sort_by (creation|arrival) "creation">
<!ELEMENT unique_by (%query_operations;)>
<!ATTLIST unique_by
    field    CDATA    #REQUIRED
    sort_by  (creation|arrival)  "creation"
    keep     (first|last)        "last"
    >
<!ELEMENT is_trigger EMPTY>
<!ELEMENT in_context (#PCDATA|trigger)*>
<!ATTLIST in_context group CDATA #IMPLIED>
<!ELEMENT match_query (#PCDATA)>
<!ELEMENT event_class (#PCDATA)>
<!ELEMENT event_name (#PCDATA)>
<!ELEMENT event_type (#PCDATA)>
<!ELEMENT event_status (#PCDATA)>
<!ELEMENT event_host (#PCDATA|trigger)*>
<!ELEMENT event_attribute (#PCDATA|trigger)*>
<!ATTLIST event_attribute
    name     CDATA             #REQUIRED
    op       (eq|ge|le|re)     "eq"
    regexp   CDATA             #IMPLIED
    >
<!ELEMENT event_min_age (#PCDATA)> <!-- age at arrival -->

<!-- event specification -->
<!ELEMENT event (name,description?,attribute*)>
<!ATTLIST event
    status      (active|inactive)    "active"
    local       (true|false)         "true"
    inject      (input|output)       "input"
    >
<!ELEMENT name (#PCDATA)>
<!ELEMENT description (#PCDATA|trigger)*>
<!ELEMENT attribute (#PCDATA|trigger)*>
<!ATTLIST attribute name CDATA #REQUIRED>
```

B.3.2 Examples

The following listing shows an example XML rule file (which is valid against the DTD specified above) with rules for some of the patterns identified in Section 2.3 (please note that these rules are examples only and were not tested):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE rules SYSTEM "rules.dtd">

<!--
Note: these rules are given as examples and were not tested with real-live
events.
-->

<rules>
  <!--
  A group of rules to handle events that are generated at a high rate;
  these rules are suitable for correlation on the source hosts only, as
  they do not distinguish between events from different hosts.
```

```

=> solves some of the problems described in section 2.3.1 and 2.3.2
-->
<group name=”event-burst-handling” order=”1”>
  <!-- Forward first event, then drop for 5 min. -->
  <rule name=”forward-first-drop-others” order=”1”>
    <events>
      <when_class>BURSTY_EVENTS</when_class>
    </events>
    <conditions>
      <context>DROP_<trigger field=”name”/></context>
    </conditions>
    <actions>
      <drop/> <!-- applies to the trigger -->
    </actions>
    <alternative_actions>
      <create_context timeout=”300”>
        <context_name>DROP_<trigger field=”name”/></context_name>
      </create_context>
      <!-- trigger will be forwarded unless prevented by another rule -->
    </alternative_actions>
  </rule>
  <!-- alternative: change to up/down signalling -->
  <rule name=”change-signaling-winbind” order=”2”
     description=”Change signaling of WINBIND events to up/down signaling.”>
    <events>
      <when_event>WINBIND:CONF:ADCONN</when_event>
    </events>
    <conditions>
      <not>
        <context>WINBIND_CONF_ADCONN<trigger field=”host”/></context>
      </not>
    </conditions>
    <actions>
      <create_context timeout=”5m” counter=”1”>
        <context_name>WINBIND_CONF_ADCONN<trigger field=”host”/></context_name>
        <event local=”false” inject=”output”>
          <name>WINBIND:CONF:ADCONN:STOP</name>
          <description>No more messages since 5 min.</description>
        </event>
      </create_context>
      <create>
        <event local=”false”>
          <name>WINBIND:CONF:ADCONN:START</name>
          <description>Got a first message.</description>
        </event>
      </create>
      <drop/>
    </actions>
    <alternative_actions> <!-- context already exists -->
      <modify_context reset_timer=”true” counter_op=”inc” counter_value=”1”>
        WINBIND_CONF_ADCONN<trigger field=”host”/>
      </modify_context>
      <drop/>
    </alternative_actions>
  </rule>
  <!-- alternative: compression -->
  <rule name=”compress” order=”3” description=”Compression rule.”>
    <events>
      <when_class>COMPRESSABLE</when_class>
    </events>
    <conditions>
```

```
      <count threshold="5">
        <event_query max_age="2m" delay="true">
          <event_class>COMPRESSABLE</event_class>
          <event_type>raw</event_type>
        </event_query>
      </count>
    </conditions>
    <actions>
      <select_events>
        <event_query max_age="2m" delay="true">
          <event_class>COMPRESSABLE</event_class>
        </event_query>
        <compress/>
      </select_events>
    </actions>
  </rule>
</group>

<!--
A group to aggregate old events (e.g. from a host that was down for some time)

=> solves the problem described in section 2.3.3
-->
<group name="collect-old" order="50" description="Aggregation of old events.">
  <rule name="create-context" order="1">
    <events>
      <when_any/>
    </events>
    <conditions>
      <trigger_match>
        <event_min_age>1d</event_min_age>
      </trigger_match>
    </conditions>
    <actions>
      <subblock>
        <conditions>
          <not><context>OLD_EVENTS</context></not>
        </conditions>
        <actions>
          <create_context timeout="10m">
            <context_name>OLD_EVENTS</context_name>
            <event>
              <name>OLD:EVENTS</name>
            </event>
          </create_context>
        </actions>
      </subblock>
    </actions>
  </rule>
  <rule name="aggregate" order="2">
    <events>
      <when_event type="timeout">OLD:EVENTS</when_event>
    </events>
    <conditions>
      <count threshold="3">
        <event_query name="old_events" delay="true" max_age="10m">
          <event_min_age>1d</event_min_age>
        </event_query>
      </count>
    </conditions>
    <actions>
      <select_events>
```

```
        <event_query>
          <match_query>old_events</match_query>
        </event_query>
        <modify status="inactive" reason="The event is outdated."/>
        <aggregate>
          <event local="false">
            <name>OLD:EVENTS:COLLECTION</name>
            <description>This event collects outdated events.</description>
          </event>
        </aggregate>
      </select_events>
    </actions>
  </rule>

</group>

<!--
Expect up after a down (because it is not necessary to reopen the ticket)

=> partially solves the problem described in Section 2.3.4
-->
<group name="nurse-handling" order="100">
  <rule name="make-context" order="1">
    <events>
      <when_event>NURSE:SERVICE:DOWN</when_event>
    </events>
    <conditions>
      <not><context>NURSE_SERVICE_DOWN_<trigger field="host"/></context></not>
    </conditions>
    <actions>
      <create_context timeout="2h">
        <context_name>NURSE_SERVICE_DOWN_<trigger field="host"/></context_name>
        <event>
          <name>NURSE-HANDLING:SERVICE:DOWN</name>
        </event>
      </create_context>
      <associate_with_context>
        NURSE_SERVICE_DOWN_<trigger field="host"/>
      </associate_with_context>
    </actions>
  </rule>
  <rule name="context-timeout" order="2">
    <events>
      <when_event type="timeout">NURSE-HANDLING:SERVICE:DOWN</when_event>
    </events>
    <conditions>
    </conditions>
    <actions>
      <select_events>
        <event_query>
          <in_context>NURSE_SERVICE_DOWN</in_context>
        </event_query>
        <aggregate>
          <event>
            <name>NURSE:SERVICE:STILL:DOWN</name>
            <description>Service is still down.</description>
          </event>
        </aggregate>
      </select_events>
    </actions>
  </rule>
  <rule name="catch-up" order="3">
```

```
    <events>
      <when_event>NURSE:SERVICE:UP</when_event>
    </events>
    <conditions>
      <context>NURSE_SERVICE_DOWN</context>
    </conditions>
    <actions>
      <!-- no need to reopen the ticket: -->
      <modify status="inactive" reason="Situation back to normal."/>
      <delete_context>NURSE_SERVICE_DOWN</delete_context>
    </actions>
    <alternative_actions>
      <!-- UP without a down before; maybe generate a warning here -->
    </alternative_actions>
  </rule>
</group>

<!--
Ignore irrelevant unique events

=> solves the problem described in section 2.3.5
-->
<group name="freshclam-handling" order="110">
  <!-- AV: ignore if it occurs only once in 5h, but create threshold -->
  <rule name="freshclam-ignore-isolated" order="1">
    <events>
      <when_event>MAIL:FRESHCLAM:ERROR</when_event>
    </events>
    <conditions>
      <not><context>CLAM_IGNORED_<trigger field="host"/></context></not>
    </conditions>
    <actions>
      <create_context timeout="5h">
        <context_name>CLAM_IGNORED_<trigger field="host"/></context_name>
      </create_context>
      <!-- applies to trigger event: -->
      <modify status="inactive" reason="Single event can be ignored."/>
      <associate_with_context>
        CLAM_IGNORED_<trigger field="host"/>
      </associate_with_context>
    </actions>
    <alternative_actions>
      <!-- create a new event, that references both that occured -->
      <add_references>
        <event_query max_age="5h">
          <union>
            <is_trigger/>
            <in_context>CLAM_IGNORED_<trigger field="host"/></in_context>
          </union>
        </event_query>
      </add_references>
    </alternative_actions>
  </rule>
  <!--
  But catch if there were more than 50 in 30 days on all host - we use a
  context with a counter here, to avoid having to keep the events
  -->
  <rule name="freshclam-longtime-monitoring" order="2">
    <events>
      <when_event>MAIL:FRESHCLAM:ERROR</when_event>
    </events>
    <conditions>
```

```
      <context>CLAM_LONGTIME</context>
    </conditions>
    <actions>
      <modify_context counter_value="1" counter_op="inc">
        CLAM_LONGTIME
      </modify_context>
      <subblock>
        <conditions>
          <context counter="50">CLAM_LONGTIME</context>
        </conditions>
        <actions>
          <create>
            <event>
              <name>CLAM:LONGTIME:THRESHOLD</name>
              <description>
                More than 50 MAIL:FRESHCLAM:ERROR events were generated in
                30 days.
              </description>
            </event>
          </create>
        </actions>
      </subblock>
    </actions>
    <alternative_actions>
      <create_context timeout="30d" counter="1">
        <context_name>CLAM_LONGTIME</context_name>
      </create_context>
    </alternative_actions>
  </rule>
</group>

<!--
Check whether VPN flickering was related to an ISP outage

=> solution for the problem described in section 2.6
-->
<group name="correlate-isp-outage" order="120">
  <rule name="check-isp-outage" order="1">
    <events>
      <when_event>VRRP:MONITOR:VPN:DOWN</when_event>
      <when_event>VRRP:MONITOR:VPN:UP</when_event>
      <when_event>ISP:HOST:UNREACHABLE</when_event>
    </events>
    <conditions>
      <!-- correalate if all 3 were created within 3 min -->
      <within timeframe="3m" match="any">
        <event_query max_age="5m" name="isp">
          <event_name>ISP:HOST:UNREACHABLE</event_name>
          <event_type>raw</event_type>
          <event_status>active</event_status>
          <event_attribute name="host">
            <trigger field="attributes.host"/>
          </event_attribute>
        </event_query>
        <event_query max_age="5m" name="up">
          <last_of>
            <intersection>
              <event_name>VRRP:MONITOR:VPN:UP</event_name>
              <event_type>raw</event_type>
              <event_status>active</event_status>
              <event_attribute name="host">
                <trigger field="attributes.host"/>
              </event_attribute>
            </intersection>
          </last_of>
        </event_query>
        <event_query max_age="5m" name="down">
          <last_of>
            <intersection>
              <event_name>VRRP:MONITOR:VPN:DOWN</event_name>
              <event_type>raw</event_type>
              <event_status>active</event_status>
              <event_attribute name="host">
                <trigger field="attributes.host"/>
              </event_attribute>
            </intersection>
          </last_of>
        </event_query>
      </within>
    </conditions>
    <actions>
      <select_events>
        <event_query>
          <union>
            <match_query>up</match_query>
            <match_query>down</match_query>
          </union>
        </event_query>
        <add_references type="parent" reason="happened during isp outage">
          <event_query><match_query>isp</match_query></event_query>
        </add_references>
      </select_events>
    </actions>
  </rule>
</group>

<!--
Service dependencies - example: DNS

=> appropriate strategy for problems described in section 2.3.7
-->
<group name="dns-dependencies" order="130">
  <rule name="create-dns-context" order="1">
    <events>
      <when_event>DNS:PROBLEM</when_event>
    </events>
    <conditions>
      <not><context>DNS_PROBLEM_<trigger field="host"/></context></not>
    </conditions>
    <actions>
      <create_context timeout="1h">
        <context_name>DNS_PROBLEM_<trigger field="host"/></context_name>
      </create_context>
      <associate_with_context>
        DNS_PROBLEM_<trigger field="host"/>
      </associate_with_context>
    </actions>
  </rule>
  <rule name="delete-dns-context" order="2">
    <events>
      <when_event>DNS:PROBLEM:STOP</when_event>
    </events>
    <conditions>
      <context>DNS_PROBLEM_<trigger field="host"/></context>
    </conditions>
    <actions>
      <delete_context>DNS_PROBLEM_<trigger field="host"/></delete_context>
    </actions>
  </rule>
  <rule name="suppress-dependencies" order="3">
    <events>
      <when_class>DEPENDENCIES:DNS</when_class>
    </events>
    <conditions>
      <context>DNS_PROBLEM_<trigger field="host"/></context>
    </conditions>
    <actions>
      <suppress reason="This event was likely caused by DNS problems.">
        <!-- the query determines the references: -->
        <event_query>
          <in_context>DNS_PROBLEM_<trigger field="host"/></in_context>
        </event_query>
      </suppress>
    </actions>
  </rule>
</group>

<!--
a group to handle failovers, which should determine, if the situation is ok
again. we assume, there is the following action plugin:

enrich_failover: a plugin, which enriches the events it receives with the
                 hostname of the master (attributes.master) and slave
                 (attributes.slave) in the cluster it belongs to
                 (independently of whether the specific event was sent from
                 master or slave)

=> solution for the pattern in section 2.3.9
-->
<group name="failover-handling" order="140">
  <rule name="enrichment" order="1">
    <events>
      <when_event>VRRP:MONITOR:VPN:DOWN</when_event>
      <when_event>VRRP:MONITOR:VPN:UP</when_event>
      <when_event>KEEPALIVED:TRANSITION:SLAVE</when_event>
      <when_event>KEEPALIVED:TRANSITION:MASTER</when_event>
    </events>
    <conditions>
    </conditions>
    <actions>
      <action_plugin name="enrich_failover"/>
    </actions>
  </rule>
  <rule name="check-isp-outage" order="2">
    <events>
      <when_event>VRRP:MONITOR:VPN:DOWN</when_event>
    </events>
    <conditions>
      <within timeframe="3m">
        <event_query>
          <is_trigger/>
        </event_query>
        <event_query max_age="20m">
          <event_name>HOST:UNREACHABLE</event_name>
          <event_type>raw</event_type>
          <event_attribute name="host">
```

```
            <trigger field="attributes.host"/>
          </event_attribute>
        </event_query>
      </within>
    </conditions>
    <actions>
      <modify_attribute name="isp_outage">true</modify_attribute>
    </actions>
  </rule>
  <rule name="detect-pattern" order="3">
    <events>
      <when_event>KEEPALIVED:TRANSITION:SLAVE</when_event>
      <when_event>KEEPALIVED:TRANSITION:MASTER</when_event>
    </events>
    <conditions>
      <pattern>
        <alphabet>
          <symbol letter="D"> <!-- master VPN down, during ISP outage -->
            <event_query name="D" max_age="10m">
              <event_name>VRRP:MONITOR:VPN:DOWN</event_name>
              <event_host><trigger field="attributes.master"/></event_host>
              <event_attribute name="isp_outage">true</event_attribute>
            </event_query>
          </symbol>
          <symbol letter="U"> <!-- master VPN up -->
            <event_query name="U" max_age="10m">
              <event_name>VRRP:MONITOR:VPN:UP</event_name>
              <event_host><trigger field="attributes.master"/></event_host>
            </event_query>
          </symbol>
          <symbol letter="S"> <!-- master transition to slave -->
            <event_query name="S" max_age="10m">
              <event_name>KEEPALIVED:TRANSITION:SLAVE</event_name>
              <event_host><trigger field="attributes.master"/></event_host>
            </event_query>
          </symbol>
          <symbol letter="M"> <!-- master transition to master -->
            <event_query name="M" max_age="10m">
              <event_name>KEEPALIVED:TRANSITION:MASTER</event_name>
              <event_host><trigger field="attributes.master"/></event_host>
            </event_query>
          </symbol>
          <symbol letter="s"> <!-- slave transition to slave -->
            <event_query name="s" max_age="10m">
              <event_name>KEEPALIVED:TRANSITION:SLAVE</event_name>
              <event_host><trigger field="attributes.slave"/></event_host>
            </event_query>
          </symbol>
          <symbol letter="m"> <!-- slave transition to master -->
            <event_query name="m" max_age="10m">
              <event_name>KEEPALIVED:TRANSITION:MASTER</event_name>
              <event_host><trigger field="attributes.slave"/></event_host>
            </event_query>
          </symbol>
        </alphabet>
        <!--
        VPN down; master becomes slave and vice versa;
        VPN up; slave becomes master and vice versa
        -->
        <regexp>D[Sm]U[Ms]$</regexp>
      </pattern>
    </conditions>
```

```
      <actions>
        <select_events>
          <event_query>
            <union>
              <match_query>D</match_query>
              <match_query>U</match_query>
              <match_query>S</match_query>
              <match_query>M</match_query>
              <match_query>s</match_query>
              <match_query>M</match_query>
            </union>
          </event_query>
          <aggregate>
            <event>
              <name>FAILOVER:SUCCESS</name>
              <description>
                There was a successful failover. The following conditions are
                fulfilled:
                 - VRRP down was during an ISP outage
                 - VRRP is up again
                 - designated master is master again
                 - designated is slave again
                 - everything happened within 10 minutes
              </description>
            </event>
          </aggregate>
          <modify status="inactive" reason="Successful failover."/>
        </select_events>
      </actions>
    </rule>
  </group>

  <!--
  Pattern to detect a widespread ISP outage:
  we assume that we get ISP:HOST:UNREACHABLE events from central monitoring,
  which have the attributes host (affected host) and isp (affected ISP)

  => in the same way, location dependent problems as in 2.3.10 could be
     detected; this would require a plugin to enrich the events with
     location information
  -->
  <group name="isp-problems" order="150"
         description="Detect widespread ISP outages.">
    <rule name="enrich-isp-link" order="1">
      <events>
        <when_event>ISP:HOST:UNREACHABLE</when_event>
      </events>
      <conditions>
        <count threshold="10">
          <event_query max_age="20m" name="ISPevents">
            <unique_by field="attributes.host">
              <intersection>
                <event_name>ISP:HOST:UNREACHABLE</event_name>
                <event_attribute name="host">
                  <trigger field="attributes.host"/>
                </event_attribute>
              </intersection>
            </unique_by>
          </event_query>
        </count>
      </conditions>
      <actions>
```

```
        <select_events>
          <event_query>
            <match_query>ISPevents</match_query>
          </event_query>
          <aggregate>
            <event>
              <name>ISP:OUTAGE:WIDESPREAD</name>
              <description>
                An outage of ISP <trigger field="attributes.isp"/>
                on at least 10 different hosts was detected.
              </description>
              <attribute name="isp">
                <trigger field="attributes.isp"/>
              </attribute>
            </event>
          </aggregate>
        </select_events>
      </actions>
    </rule>
  </group>
</rules>

<!--
vim: sw=2 ts=2
-->
```

Appendix C

Parameters and Configuration File Structure

C.1 ace Command Line Parameters

The following listing shows the command line parameters available for ace, as printed by the -h command line switch:

```
Usage: ace [options]

Options:
  -h, --help            show this help message and exit
  -c FILE, --config-file=FILE
                        read configuration from FILE [default:
                        /etc/ace/ace.conf if available, built-in defaults
                        otherwise]
  -C, --print-config-template
                        print a template for the configuration file
  -d, --daemon          run as daemon (default: run in foreground)
  -r RULESOURCE, --rule-source=RULESOURCE
                        source of correlation rules (default:
                        file:filename=<ace-dir>/etc/emptyrules.xml)
  -R, --rpc-server      start RPC server for remote control (default: don't
                        start)
  -v, --verbose         be verbose about what's going on (can be used multiple
                        times for greater effect; use at least twice to enable
                        stack traces)
  -p, --start-python    start interactive Python console for debugging
                        (default: don't start)
  -i, --start-ipython   start interactive IPython console for debugging
                        (default: don't start)
  -P FILE, --profile=FILE
                        run cProfile for speed profiling and store its output
                        into FILE (default: don't run cProfile)
```

C.2 ace Configuration File Structure

The following listing shows the structure of a configuration file with default options, as generated when using the -C command line switch (an explanation of the meaning for each option can be found in the documentation of the `Config` class, in the HTML documentation, which can be found under `doc` on the CD-ROM):

```
# ace configuration file

[main]
python_console = False                  # bool
rpcserver_port = 1070                   # int
input_queue_max_size = 100000           # int
cache_max_size = 10000                  # int
thread_sleep_time = 0.1                 # float
rules_dtd = <ace-dir>/etc/rules.dtd      # string
lockfile = /var/lock/ace-lockfile       # string
realtime = True                         # bool
hostname = anm-desktop                  # string
rpcserver_host = localhost              # string
classlist = file:filename=<ace-dir>/etc/emptyclasses.xml # string
smtpserver = localhost                  # string
ipython_console = False                 # bool
output_queue_max_size = 10000           # int
events_dtd = <ace-dir>/etc/events.dtd # string
rulesource = file:filename=<ace-dir>/etc/emptyrules.xml # string
daemon = False                          # bool
loglevel = 3                            # int
rpcserver = False                       # bool
verbosity = 3                           # int
simulation = False                      # bool
fast_exit = False                       # bool
logident = ace                          # string

[input]
translator = letter                     # string
source = file                           # string

[output]
translator = linebased                  # string
sink = file                             # string
```

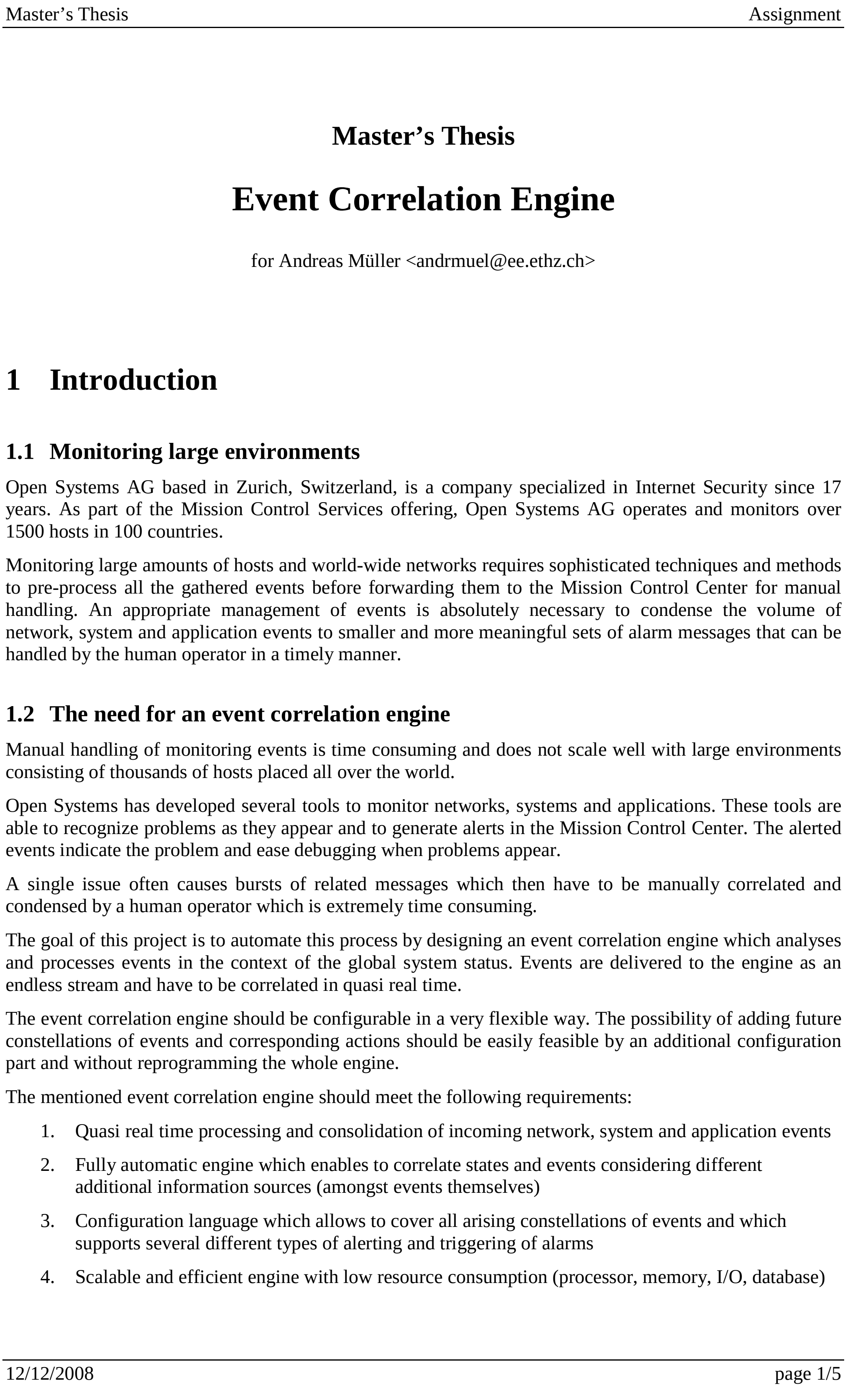
The configuration may have more than one input and output section. If a translator, source or sink requires options, they can be given in the form `:option=value:option2=value2:..`, e.g.:

- `translator=linebased:rulefile=input-translation.xml`
- `source=file:filename=test.csv`
- `sink=tcp:host=localhost:port=2000`
- `sink=rpc:port=1070`

Appendix D

Assignment

The following pages reflect the task assignment, as provided by Christoph Göldi.



Master's Thesis

Event Correlation Engine

for Andreas Müller <andrmuel@ee.ethz.ch>

1 Introduction

1.1 Monitoring large environments

Open Systems AG based in Zurich, Switzerland, is a company specialized in Internet Security since 17 years. As part of the Mission Control Services offering, Open Systems AG operates and monitors over 1500 hosts in 100 countries.

Monitoring large amounts of hosts and world-wide networks requires sophisticated techniques and methods to pre-process all the gathered events before forwarding them to the Mission Control Center for manual handling. An appropriate management of events is absolutely necessary to condense the volume of network, system and application events to smaller and more meaningful sets of alarm messages that can be handled by the human operator in a timely manner.

1.2 The need for an event correlation engine

Manual handling of monitoring events is time consuming and does not scale well with large environments consisting of thousands of hosts placed all over the world.

Open Systems has developed several tools to monitor networks, systems and applications. These tools are able to recognize problems as they appear and to generate alerts in the Mission Control Center. The alerted events indicate the problem and ease debugging when problems appear.

A single issue often causes bursts of related messages which then have to be manually correlated and condensed by a human operator which is extremely time consuming.

The goal of this project is to automate this process by designing an event correlation engine which analyses and processes events in the context of the global system status. Events are delivered to the engine as an endless stream and have to be correlated in quasi real time.

The event correlation engine should be configurable in a very flexible way. The possibility of adding future constellations of events and corresponding actions should be easily feasible by an additional configuration part and without reprogramming the whole engine.

The mentioned event correlation engine should meet the following requirements:

1. Quasi real time processing and consolidation of incoming network, system and application events
2. Fully automatic engine which enables to correlate states and events considering different additional information sources (amongst events themselves)
3. Configuration language which allows to cover all arising constellations of events and which supports several different types of alerting and triggering of alarms
4. Scalable and efficient engine with low resource consumption (processor, memory, I/O, database)

2 The Task

The thesis is conducted at Open Systems AG (http://www.open.ch) in Zurich. The task of this Master's Thesis is to develop an event correlation engine especially suited for monitoring large amounts of hosts in world-wide networks.

The correlation engine has to process all incoming events from every Mission Control host in the world. The consolidation can be achieved in two steps. In a first step, all events generated by a host can be correlated before they are sent to the Mission Control Center. At this central point where all events of every host in the world come together a further correlation in a more global context can be done.

In addition to the events themselves, the correlation engine should consider further information from other sources like configuration parameters, global network states and setup types. Depending on the correlation result, events should be summarized, passed to a Mission Control Engineer for manual handling or even dropped. It also should be possible to trigger commands if certain events occur.

The task of the student is split into five major subtasks that all will be: (i) analysis of potential event patterns, (ii) study of existing event correlation engine approaches, (iii) specification of an event correlation engine, (iv) implementation of a prototype, and (v) test of and evaluation with the prototype.

2.1 Analysis of potential event patterns

Andreas should evaluate the event history of previous months to collect all possibly arising event patterns. Interviewing experienced Mission Control Engineers will give valuable information about common patterns too and should be considered during this phase of the Thesis.

The found event patterns should be classified and appropriate operations for recognizing and consolidating these patterns should be proposed. Additionally the student will become familiar with the Open Systems environment while searching for patterns and studying the different services and possible setups.

2.2 Study of existing event correlation engine approaches

The main focus of Andreas' thesis lies in finding an appropriate event correlation engine for our specific environment. Andreas should study existing event correlation approaches to get an idea what approaches would fit our needs best and to find an own approach which suits all our requirements and fulfils the task described.

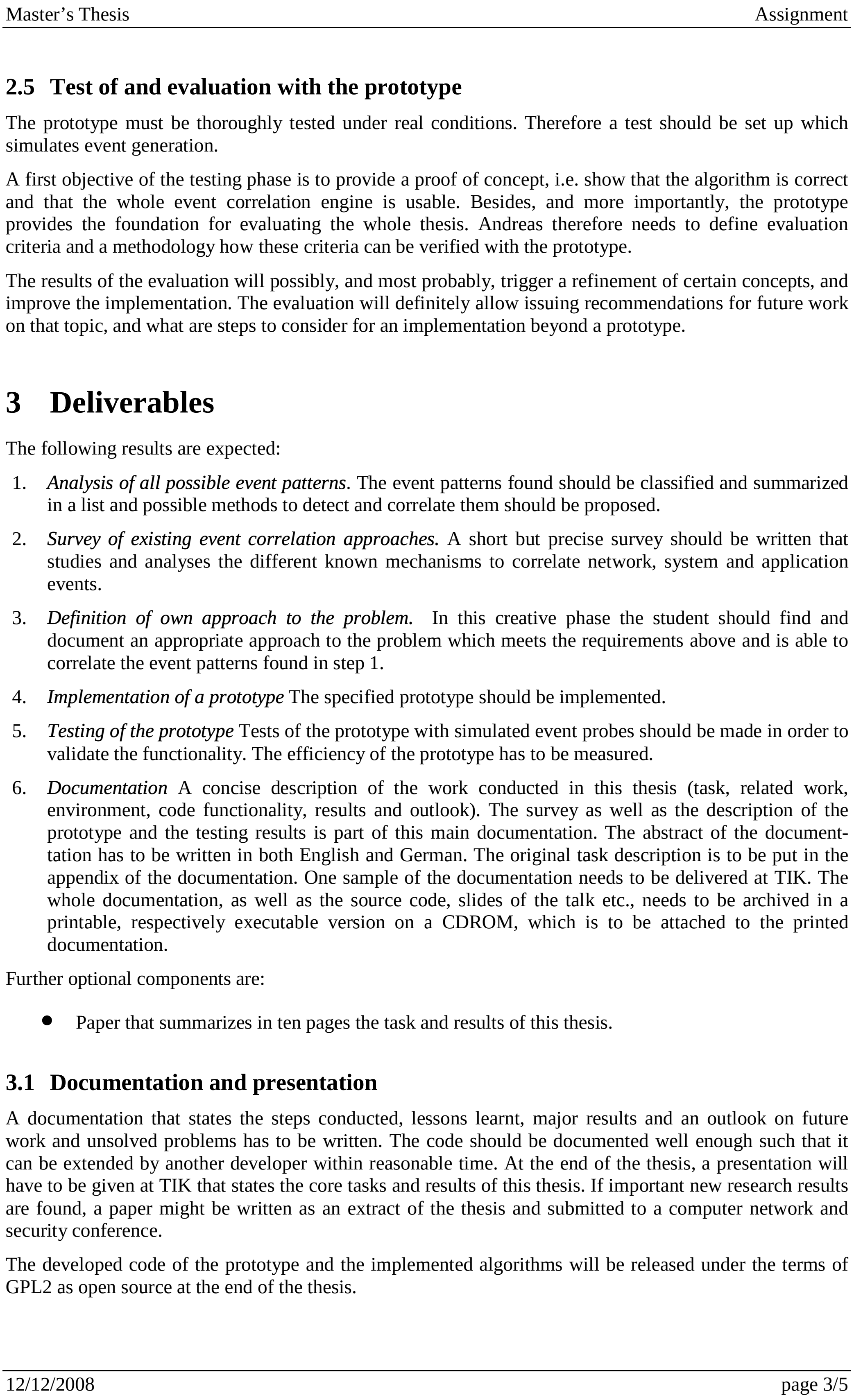
The different approaches should be compared and weighed against each other. The most promising approach can be used as inspiration for implementing the event correlation engine.

2.3 Specification of an event correlation engine

Based on the study of known event correlation mechanisms Andreas needs to propose a new and improved method to correlate the events generated by the world-wide distributed Mission Control hosts. He needs to write a specification, such that the proposed algorithm can be used as a self-contained and automatic system that allows to consolidate event patterns. Promising algorithms and ideas that have been encountered in the analysis phase can of course be incorporated.

2.4 Implementation of a prototype

Following the above mentioned specification, a prototype should be implemented on Linux (and run possibly also on Solaris). The resource consumption and performance of the prototype must be such that the server can still offer its normal services.



2.5 Test of and evaluation with the prototype

The prototype must be thoroughly tested under real conditions. Therefore a test should be set up which simulates event generation.

A first objective of the testing phase is to provide a proof of concept, i.e. show that the algorithm is correct and that the whole event correlation engine is usable. Besides, and more importantly, the prototype provides the foundation for evaluating the whole thesis. Andreas therefore needs to define evaluation criteria and a methodology how these criteria can be verified with the prototype.

The results of the evaluation will possibly, and most probably, trigger a refinement of certain concepts, and improve the implementation. The evaluation will definitely allow issuing recommendations for future work on that topic, and what are steps to consider for an implementation beyond a prototype.

3 Deliverables

The following results are expected:

1. *Analysis of all possible event patterns*. The event patterns found should be classified and summarized in a list and possible methods to detect and correlate them should be proposed.
2. *Survey of existing event correlation approaches.* A short but precise survey should be written that studies and analyses the different known mechanisms to correlate network, system and application events.
3. *Definition of own approach to the problem.* In this creative phase the student should find and document an appropriate approach to the problem which meets the requirements above and is able to correlate the event patterns found in step 1.
4. *Implementation of a prototype* The specified prototype should be implemented.
5. *Testing of the prototype* Tests of the prototype with simulated event probes should be made in order to validate the functionality. The efficiency of the prototype has to be measured.
6. *Documentation* A concise description of the work conducted in this thesis (task, related work, environment, code functionality, results and outlook). The survey as well as the description of the prototype and the testing results is part of this main documentation. The abstract of the documenttation has to be written in both English and German. The original task description is to be put in the appendix of the documentation. One sample of the documentation needs to be delivered at TIK. The whole documentation, as well as the source code, slides of the talk etc., needs to be archived in a printable, respectively executable version on a CDROM, which is to be attached to the printed documentation.

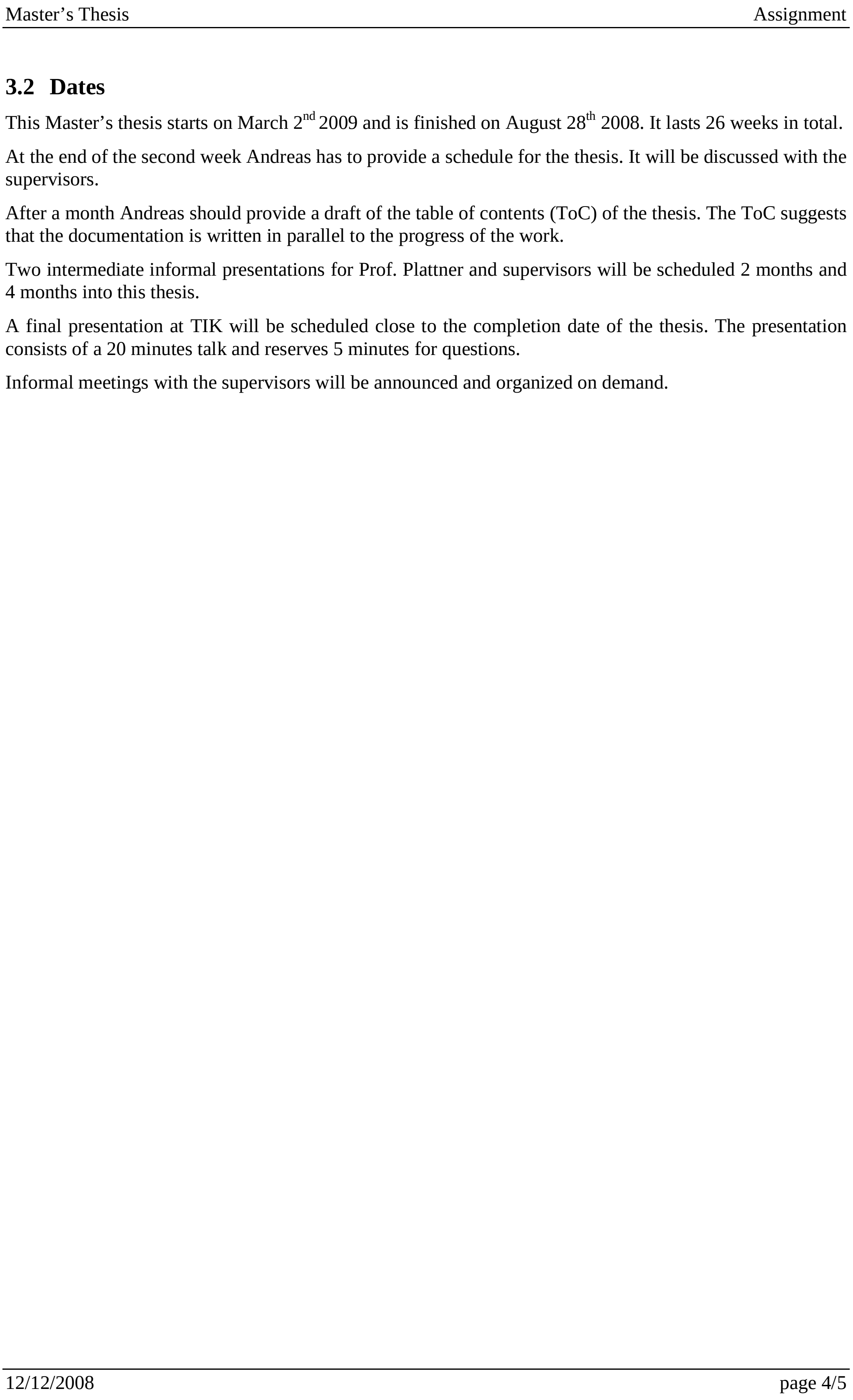
Further optional components are:

- Paper that summarizes in ten pages the task and results of this thesis.

3.1 Documentation and presentation

A documentation that states the steps conducted, lessons learnt, major results and an outlook on future work and unsolved problems has to be written. The code should be documented well enough such that it can be extended by another developer within reasonable time. At the end of the thesis, a presentation will have to be given at TIK that states the core tasks and results of this thesis. If important new research results are found, a paper might be written as an extract of the thesis and submitted to a computer network and security conference.

The developed code of the prototype and the implemented algorithms will be released under the terms of GPL2 as open source at the end of the thesis.



3.2 Dates

This Master’s thesis starts on March 2nd 2009 and is finished on August 28th 2008. It lasts 26 weeks in total.

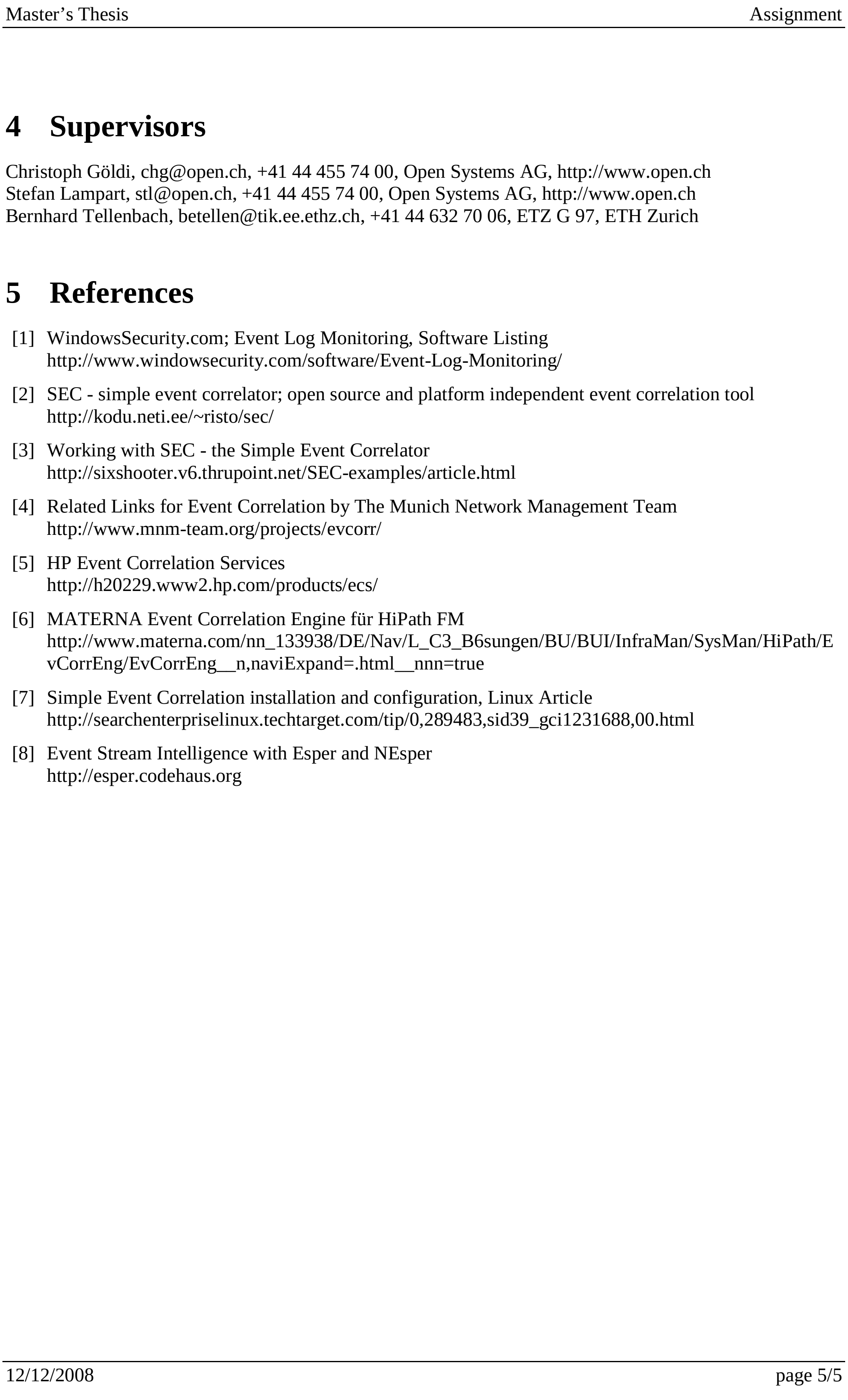
At the end of the second week Andreas has to provide a schedule for the thesis. It will be discussed with the supervisors.

After a month Andreas should provide a draft of the table of contents (ToC) of the thesis. The ToC suggests that the documentation is written in parallel to the progress of the work.

Two intermediate informal presentations for Prof. Plattner and supervisors will be scheduled 2 months and 4 months into this thesis.

A final presentation at TIK will be scheduled close to the completion date of the thesis. The presentation consists of a 20 minutes talk and reserves 5 minutes for questions.

Informal meetings with the supervisors will be announced and organized on demand.



4 Supervisors

Christoph Göldi, chg@open.ch, +41 44 455 74 00, Open Systems AG, http://www.open.ch
Stefan Lampart, stl@open.ch, +41 44 455 74 00, Open Systems AG, http://www.open.ch
Bernhard Tellenbach, betellen@tik.ee.ethz.ch, +41 44 632 70 06, ETZ G 97, ETH Zurich

5 References

[1] WindowsSecurity.com; Event Log Monitoring, Software Listing
http://www.windowsecurity.com/software/Event-Log-Monitoring/

[2] SEC - simple event correlator; open source and platform independent event correlation tool
http://kodu.neti.ee/~risto/sec/

[3] Working with SEC - the Simple Event Correlator
http://sixshooter.v6.thrupoint.net/SEC-examples/article.html

[4] Related Links for Event Correlation by The Munich Network Management Team
http://www.mnm-team.org/projects/evcorr/

[5] HP Event Correlation Services
http://h20229.www2.hp.com/products/ecs/

[6] MATERNA Event Correlation Engine für HiPath FM
http://www.materna.com/nn_133938/DE/Nav/L_C3_B6sungen/BU/BUI/InfraMan/SysMan/HiPath/EvCorrEng/EvCorrEng__n,naviExpand=.html__nnn=true

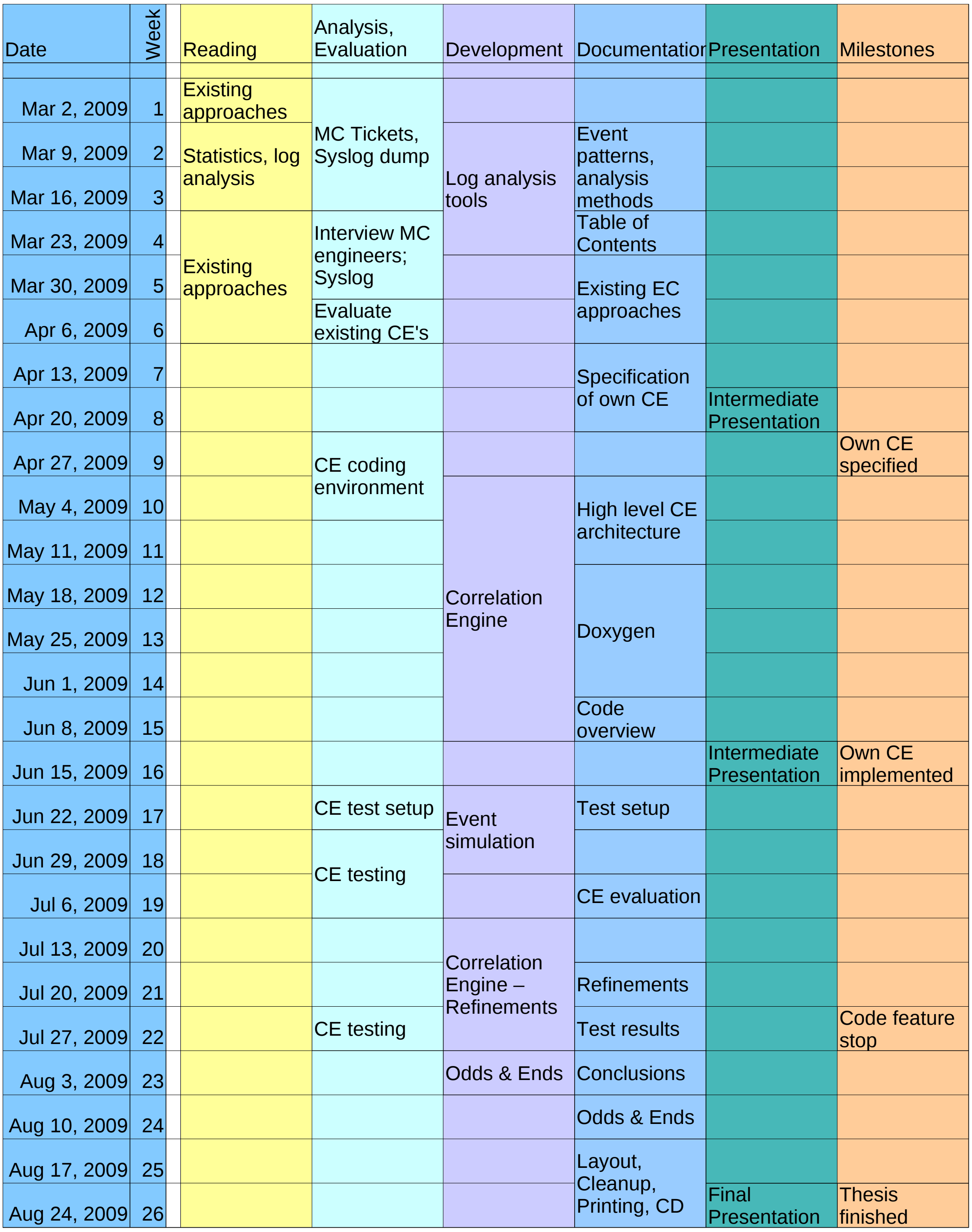
[7] Simple Event Correlation installation and configuration, Linux Article
http://searchenterpriselinux.techtarget.com/tip/0,289483,sid39_gci1231688,00.html

[8] Event Stream Intelligence with Esper and NEsper
http://esper.codehaus.org

Appendix E

Schedule

Figure E.1 reflects the initial version of the project schedule for this master's thesis.



Date	Week		Reading	Analysis, Evaluation	Development	Documentatior	Presentation	Milestones
Mar 2, 2009	1		Existing approaches	MC Tickets, Syslog dump				
Mar 9, 2009	2		Statistics, log analysis		Log analysis tools	Event patterns, analysis methods		
Mar 16, 2009	3							
Mar 23, 2009	4		Existing approaches	Interview MC engineers; Syslog		Table of Contents		
Mar 30, 2009	5					Existing EC approaches		
Apr 6, 2009	6			Evaluate existing CE's				
Apr 13, 2009	7					Specification of own CE		
Apr 20, 2009	8						Intermediate Presentation	
Apr 27, 2009	9			CE coding environment				Own CE specified
May 4, 2009	10				Correlation Engine	High level CE architecture		
May 11, 2009	11							
May 18, 2009	12					Doxygen		
May 25, 2009	13							
Jun 1, 2009	14							
Jun 8, 2009	15					Code overview		
Jun 15, 2009	16						Intermediate Presentation	Own CE implemented
Jun 22, 2009	17			CE test setup	Event simulation	Test setup		
Jun 29, 2009	18			CE testing				
Jul 6, 2009	19					CE evaluation		
Jul 13, 2009	20				Correlation Engine – Refinements			
Jul 20, 2009	21					Refinements		
Jul 27, 2009	22			CE testing		Test results		Code feature stop
Aug 3, 2009	23				Odds & Ends	Conclusions		
Aug 10, 2009	24					Odds & Ends		
Aug 17, 2009	25					Layout, Cleanup, Printing, CD		
Aug 24, 2009	26						Final Presentation	Thesis finished

Figure E.1: Initial project schedule.

Appendix F

Presentation

The following pages show the presentation slides, which were used for the final presentation of this master's thesis.

Appendix G

Acronyms

The following list provides an overview of the acronyms used throughout this thesis.

AD Active Directory

AI Artificial Intelligence

AL Apache License

AL Apache License

ANN Artificial Neural Network

API Application Programming Interface

ASCII American Standard Code for Information Interchange

BRMS Business Rule Management System

BSD Berkeley Software Distribution

CBR Case-based Reasoning

CEP Complex Event Processing

CFG Context Free Grammar

CLI Command Line Interface

CPU Central Processing Unit

CSP Constraint Satisfaction Problem

CSV Comma Separated Values

DDoS Distributed Denial of Service

DNS Domain Name Service

DRL Drools Rule Language

DSL Domain Specific Language

DTD Document Type Definition

DoS Denial of Service

ECA Event Condition Action

ECDL Event Correlation Description Language

ECE Event Correlation Engine

ECS Event Correlation Services

EOF End of File

EPL Event Processing Language

EQL Event Query Language

ESP Event Stream Processing

FAQ Frequently Asked Question

FIFO First In, First Out

FSM Finite State Machine

FST Finite State Transducer

GASSATA Genetic Algorithm for Simplified Security Audit Trail Analysis

GNU GNU's Not Unix

GPL General Public License

GUI Graphical User Interface

HIDS Host-based Intrusion Detection System

HTML Hypertext Markup Language

HTTPS Hypertext Transfer Protocol Secure

HTTP Hypertext Transfer Protocol

IDE Integrated Development Environment

IDMEF Intrusion Detection Message Exchange Format

IDS Intrusion Detection System

IPC Interprocess Communication

IPS Intrusion Prevention System

IP Internet Protocol

ISP Internet Service Provider

JMX Java Management Extensions

JVM Java Virtual Machine

LGPL Lesser General Public License

LML Log Monitoring Lackey

MBR Model-based Reasoning

NIC Network Interface Controller

NNM Network Node Manager

NP Nondeterministic Polynomial

OSI Open Systems Interconnection

OS Operating System

PAM Pluggable Authentication Modules

PDF Portable Document Format

PEP Python Enhancement Proposal

PHP PHP: Hypertext Preprocessor

POJO Plain Old Java Object

RAID Redundant Array of Inexpensive Disks

RAM Random Access Memory

RBR Rule-based Reasoning

RFC Request For Comment

RPC Remote Procedure Call

SEC Simple Event Correlator

SEM Security Event Management

SIEM Security Information and Event Management

SIM Security Information Management

SLA Service Level Agreement

SMART Support Management Automated Reasoning Technology

SMS Short Message Service

SNMP Simple Network Management Protocol

SQL Structured Query Language

SSHd Secure Shell daemon

SSH Secure Shell

SSL Secure Sockets Layer

TCP Transmission Control Protocol

TEC Tivoli Enterprise Console

TLS Transport Layer Security

TMF Tivoli Management Framework

UDP User Datagram Protocol

UI User Interface

UTC Coordinated Universal Time

VCS Version Control System

VLSI Very Large Scale Integration

VPN Virtual Private Network

VRRP Virtual Router Redundancy Protocol

XML Extensible Markup Language

XMPP Extensible Messaging and Presence Protocol

XPath XML Path

Appendix H

CD-ROM Content Listing

This master thesis is accompanied by a CD-ROM, which has the following content:

`/`

`task`	Task description
`thesis`	This master thesis as PDF file
`presentation`	Presentation slides as PDF file
`code`	The developed Python code
`log_analysis`	Tools used for log analysis
`evaluation`	Python scripts used for evaluation
`doc`	Automatically generated HTML code documentation
`dtd`	XML document type definitions
`xml-examples`	Examples of XML rules and events
`evaluation-rules`	XML rules used for simulation

Bibliography

[1] Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1):39–59, March 1994. Available at http://www.iiia.csic.es/People/enric/AICom.html.

[2] Timothy L. Acorn and Sherry H. Walden. Smart: Support management automated reasoning technology for compaq customer service. In *IAAI-92 Proceedings*, 1992.

[3] E. Todd Atkins. Swatch manual page. Accessible with `man 1 swatch`, on a host where Swatch is installed.

[4] Irad Ben-gal. Bayesian networks, 2007. Available at http://www.eng.tau.ac.il/~bengal/BN.pdf.

[5] Thomas Bernhardt and Alexandre Vasseur. Esper: Event stream processing and correlation. *ONJava*, 2007. Available at http://www.onjava.com/pub/a/onjava/2007/03/07/esper-event-stream-processing-and-correlation.html.

[6] Hans Beyer. Service-oriented event correlation. Master's thesis, Technical University Munich, 2007.

[7] Nikolai Bezroukov. Event correlation technologies. Available at http://www.softpanorama.org/Admin/Event_correlation/.

[8] Nikolai Bezroukov. Tivoli enterprise console. Available at http://www.softpanorama.org/Admin/Tivoli/TEC/.

[9] A. Bouloutas, G.W. Hart, and M. Schwartz. Simple finite-state fault detectors for communication networks. *Communications, IEEE Transactions on*, 40(3):477–479, Mar 1992.

[10] A.T. Bouloutas, S. Calo, and A. Finkel. Alarm correlation and fault identification in communication networks. *Communications, IEEE Transactions on*, 42(234):523–533, Feb/-Mar/Apr 1994.

[11] Daniel B. Cid. Log analysis using ossec, 2007. Available at http://ossec.net/ossec-docs/auscert-2007-dcid.pdf.

[12] Drools community. Drools website. Accessible at http://www.jboss.org/drools/.

[13] OSSEC community. Ossec source code. Version 2.0, available at http://www.ossec.net/files/ossec-hids-2.0.tar.gz.

[14] OSSEC community. Ossec website. Accessible at http://ossec.net.

[15] OSSIM community. Ossim documentation. Accessible at http://www.ossim.net/dokuwiki/doku.php.

[16] OSSIM community. Ossim website. Accessible at http://www.ossim.org.

[17] OSSIM community. Ossim source code, 2008. Version 0.9.9, available at http://sourceforge.net/projects/os-sim/.

[18] Prelude community. Prelude correlator source code. Version 0.9.0-beta3, available at http://www.prelude-ids.com/en/development/download/.

[19] Prelude community. Prelude documentation. Acessible at https://dev.prelude-ids.com/.

[20] Gregory F. Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial Intelligence*, 42(2-3):393 – 405, 1990.

[21] Daniel D. Corkill. Collaborating software: Blackboard and multi-agent systems & the future. In *Proceedings of the International Lisp Conference*, 2003. Available at http://dancorkill.home.comcast.net/~dancorkill/pubs/ilc03.pdf.

[22] Robert N. Cronk, Paul H. Callahan, and Lawrence Bernstein. Rule based expert systems for network management and operations: An introduction, 1988.

[23] Randall Davis, Howard Shrobe, Walter Hamscher, Kären Wieckert, Mark Shirley, and Steve Polit. Diagnosis based on description of structure and function. In *National Conference on Artificial Intelligence*, pages 137–142. American Association for Artificial Intelligence, 1982.

[24] H. Debar, D. Curry, and B. Feinstein. Rfc 4765: The intrusion detection message exchange format (idmef), March 2007. Available at http://www.ietf.org/rfc/rfc4765.txt.

[25] LXML Developers. Lxml api documentation, 2009. Available at http://codespeak.net/lxml/api/index.html.

[26] Robert B. Doorenbos. *Production Matching for Large Learning Systems.* PhD thesis, Carnegie Mellon University, 1995. Available at http://reports-archive.adm.cs.cmu.edu/anon/1995/CMU-CS-95-113.pdf.

[27] Xiaojiang Du, M.A. Shayman, and R.A. Skoog. Using neural network in distributed management to identify control and management plane poison messages. In *Military Communications Conference, 2003. MILCOM 2003. IEEE*, volume 1, pages 458–463, Oct. 2003.

[28] EsperTech. Esper website. Available at http://esper.codehaus.org.

[29] Python Software Foundation. Python glossary, 2009. Available at http://docs.python.org/glossary.html.

[30] Python Software Foundation. The python standard library, 2009. Available at http://docs.python.org/library/.

[31] Python Software Foundation. The python standard library – a synchronized queue class, 2009. Available at http://docs.python.org/library/queue.html.

[32] Python Software Foundation. The python standard library – regular expression operations, 2009. Available at http://docs.python.org/library/re.html.

[33] Python Software Foundation. The python standard library – time access and conversions, 2009. Available at http://docs.python.org/library/time.html.

[34] OpenNMS Group and The Order of the Green Polo. Opennms website. Accessible at http://www.opennms.org.

[35] Boris Gruschke. Integrated event management: Event correlation using dependency graphs. In *9th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 98)*, 1998. Available at http://www.mnm-team.org/~gruschke/.

[36] Andreas Hanemann. A hybrid rule-based/case-based reasoning approach for service fault diagnosis. In *Advanced Information Networking and Applications, 2006. AINA 2006. 20th International Conference on*, volume 2, page 5 pp., April 2006.

[37] Andreas Hanemann and Martin Sailer. A framework for service quality assurance using event correlation techniques. In *Telecommunications, 2005. Advanced Industrial Conference on Telecommunications/Service Assurance with Partial and Intermittent Resources Conference/ E-Learning on Telecommunications Workshop. AICT/SAPIR/ELETE 2005. Proceedings*, pages 428–433, July 2005.

[38] Stephen E. Hansen and E. Todd Atkins. Automated system monitoring and notification with swatch. In *USENIX*, 1993. Available at http://www.usenix.org/publications/library/proceedings/lisa93/full_papers/hansen.ps.

[39] HP. hp openview event correlation services (ecs) for network node manager and operations. Available at http://www.openview.hp.com/products/ecs/ds/ecs_ds.pdf.

[40] Nick Hutton. Preparing for security event management. *360° Inforamtion Security Magazine*, 2007. Available at http://www.windowsecurity.com/whitepapers/Preparing-Security-Event-Management.html.

[41] IBM. Ibm tec and netview product manuals. Available at http://publib.boulder.ibm.com/infocenter/tivihelp/v3r1/.

[42] G. Jakobson and M. Weissman. Alarm correlation. *Network, IEEE*, 7(6):52–59, Nov 1993.

[43] Gabriel Jakobson, John Buford, and Lundy Lewis. Towards an architecture for reasoning about complex event-based dynamic situations. In *International Workshop on Distributed Event-based Systems (DEBS 2004)*, 2004.

[44] Eamonn Kent. Distributed fault location. Master's thesis, University of Western Ontario, 1998.

[45] A.A. Lazar, Weiguo Wang, and R.H. Deng. Models and algorithms for network fault detection and identification: a review. In *Singapore ICCS/ISITA '92. 'Communications on the Move'*, pages 999–1003 vol.3, Nov 1992.

[46] David B. Leake. Cbr in context: The present and future, 1996.

[47] L. Lewis. A case-based reasoning approach to the management of faults in communication networks. In *INFOCOM '93. Proceedings.Twelfth Annual Joint Conference of the IEEE Computer and Communications Societies. Networking: Foundation for the Future. IEEE*, pages 1422–1429 vol.3, 1993.

[48] R. Lippmann. An introduction to computing with neural nets. *ASSP Magazine, IEEE*, 4(2):4–22, Apr 1987.

[49] C. Lonvick. Rfc 3164: The bsd syslog protocol, August 2001. Available at http://www.ietf.org/rfc/rfc3164.txt.

[50] David Luckham and Roy Schulte. Event processing glossary, July 2008. Version 1.1. Available at http://complexevents.com/?p=409.

[51] Jean Philippe Martin-Flatin, Gabriel Jakobson, and Lundy Lewis. Event correlation in integrated management: Lessons learned and outlook. *J. Netw. Syst. Manage.*, 15(4):481–502, December 2007.

[52] Dilmar Malheiros Meira. *A Model For Alarm Correlation in Telecommunications Networks*. PhD thesis, Federal University of Minas Gerais, 1997.

[53] Ludovic Mé. Gassata – a genetic algorithm as an alternative tool for security audit trail analysis, 2000.

[54] Ricardo Olivieri. Implement business logic with the drools rules engine. *IBM developerWorks*, 2006. Available at http://www.ibm.com/developerworks/java/library/j-drools/.

[55] Fabien Pouget and Marc Dacier. Alert correlation: Review of the state of the art. Technical Report EURECOM+1271, Institut Eurecom, France, Dec 2003.

[56] Mihaela Sabin, Robert D. Russell, and Eugene C. Freuder. Generating diagnostic tools for network fault management. In *Fifth IFIP/IEEE International Symposium on Integrated Network Management (IM'97)*, pages 700–711. Chapman & Hall, 1997.

[57] Kenneth R. Sheers. Hp openview event correlation services. *HP Journal*, October 1996. Available at http://www.hpl.hp.com/hpjournal/96oct/oct96a4.htm.

[58] Stephen Slade. Case-based reasoning: A research paradigm. *AI Magazine*, 12(1):42–55, 1991.

[59] M. Steinder and A. Sethi. A survey of fault localization techniques in computer networks, 2004. Available at http://www.cis.udel.edu/~sethi/papers/04/socp04.pdf.

[60] Daniel Stutzbach. Blist: A faster list-like type, 2007. Available at http://www.python.org/dev/peps/pep-3128/.

[61] PreludeIDS Technologies. Prelude website. Acessible at http://www.prelude-ids.com.

[62] Kerry Thompson. An introduction to logsurfer. *SysAdmin magazine*, 2004. Available at http://www.crypt.gen.nz/papers/logsurfer.html.

[63] Risto Vaarandi. Simple event correlator faq. Available at http://simple-evcorr.sourceforge.net/FAQ.html.

[64] Risto Vaarandi. Sec – a lightweight event correlation tool. In *Proceedings of the 2002 IEEE Workshop on IP Operations and Management*, 2002. Available at http://kodu.neti.ee/~risto/publications/sec-ipom02-web.pdf.

[65] Risto Vaarandi. *Tools and Techniques for Event Log Analysis*. PhD thesis, Tallinn University of Technology, 2005. Available at http://kodu.neti.ee/~risto/publ.html.

[66] Risto Vaarandi. *SEC manual page*, 2008. Accessible with `man 1 sec`, on a host where SEC is installed.

[67] Risto Vaarandi et al. Sec rule repository. Available at http://kodu.neti.ee/~risto/sec/rulesets/.

[68] Guido van Rossum. Python enhancement proposal 3000, 2009. Available at http://www.python.org/dev/peps/pep-3000/.

[69] Hermann Wietgrefe, Klaus dieter Tuchs, Klaus Jobmann, Guido Carls, Peter Fröhlich, Wolfgang Nejdl, and Sebastian Steinfeld. Using neural networks for alarm correlation in cellular phone networks. In *Proc. International Workshop on Applications of Neural Networks in Telecommunications*, 1997.

[70] Wikipedia. Birthday problem, 2009. [Online; accessed 12-June-2009].

[71] Wikipedia. Eval — wikipedia, the free encyclopedia, 2009. [Online; accessed 17-August-2009].

[72] Wikipedia. Event correlation — wikipedia, the free encyclopedia, 2009. [Online; accessed 27-April-2009].

[73] S.A. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie. High speed and robust event correlation. *Communications Magazine, IEEE*, 34(5):82–90, May 1996.