Mining Complex Event Patterns in Computer Networks

Dietmar Seipel[1], Philipp Neubeck[2], Stefan Köhler[3], and Martin Atzmueller[4]

[1] University of Würzburg, Department of Computer Science
[2] Google Germany GmbH, Munich
[3] Infosim GmbH & Co. KG, Würzburg
[4] University of Kassel, Knowledge and Data Engineering Group

Abstract. More and more ubiquitous and mobile computer networks are becoming available, which leads to a massive growth in the amount of traffic and according log messages. For handling and managing networks efficiently, sophisticated approaches for network management and analysis are necessary. In this paper, we show how to use temporal data mining in a declarative framework for analysing log files for computer networks. From a sequence of network management protocol messages, we derive temporal association rules, which state frequent dependencies between events. We also present methods for extendable and modular parsing of text messages and their analysis in log files based on XML.

1 Introduction

With the advent of mobile, dynamic, and more and more ubiquitous devices, computer networks providing the technical infrastructure, that is, the links between the individual computational nodes, are becoming more widespread at a rapid pace. In such contexts, e.g., in TCP/IP–based environments, network management is a critical issue for providing continuous quality of service and stability of the network. With the growing adoption and size of the networks, there is an increasing amount of alarm events and error messages indicating exceptional situations. Usually, the detection of those is based on log analysis, searching for faults and other exceptional events. For an appropriate management, sophisticated approaches need to be applied for handling both the amount of data and for determining patterns at the appropriate level of abstraction.

In this paper, we consider a declarative framework for analysing events in networks. More specifically, we focus on the analysis of network events using temporal data mining techniques. The main contribution are the integrated analysis as well as the preprocessing and postprocessing options of the mined patterns. The presented techniques allow for the detection of relations between different events in order to identify complex event patterns. From a log file, e.g., we could automatically extract the rule: Every "link down" event is preceded by a "lineprotocol down" event in less than 10 seconds. Thus, complex patterns consisting of sequences of events are identified and can then be applied, e.g., for managing and optimizing the system.

The rest of the paper is structured as follows: Section 2 briefly discusses related work. After that, Section 3 summarizes steps for preprocessing the log files. Section 4 describes the framework for temporal data mining including a case study of the presented approach. Finally, Section 5 concludes with a summary and discusses possible future extensions.

2 Related Work

Temporal analysis of event sequences and temporal data mining are increasingly prominent research approaches in recent years. Especially, mining alarm patterns has attracted significant attention, e.g., [5, 2, 4]: Most similarly to our setting, de Aguiar et al. [4] present an approach for alarm pattern mining. However, in contrast to their approach, the presented method and case study focuses more on the message and sequence analysis. It considers grouping equivalences, resolving word–based most specific event types, and filtering rules using background knowledge, e.g., already known relations.

Laxman et al. [6] provide a general overview of the field of temporal data mining. They discuss the problem setting and introduce several algorithms. Furthermore, Achar et al. [1] provide a unified view on apriori–based algorithms for frequent episode discovery. Mannila et al. [7, 5] provide classic instantiations of the problem and propose the WINEPI and MINEPI algorithms for mining frequent sequences and episodes; these algorithms are among the core ingredients of our presented approach. However, we extend these approaches by flexible filtering, mainly concerning overlapping sequences and repetitions, and additional message and sequence analysis.

Wu et al. [13] as well as Tatti and Cule [9] consider extensions of episode mining algorithms, focusing on complete or strictly closed episodes. As discussed below, this remains as one of the possible future extensions of this work.

For the analysis of event logs, the Simple Logfile Clustering Tool (SLCT) [12], for example, analyses text messages and determines frequent line patterns. The patterns can be reused for defining event types by the analyst. The approach is iterative and requires human interaction; an automatic approach is not mentioned. It is also important to note, that SLCT does not consider temporal relations, but only absolute frequencies of patterns. In contrast, our approach combines sequence and message analysis. Instead of a simple event type, we assign each event several key/value pairs as described below.

3 Log File Processing

A common source of log information are facilities, such as *syslog* under Unix–like operating systems and *Windows Event Logs* under Microsoft Windows. Several log facilities collect events with a text field, which is used in many ways and not further standardised. This section is concerned with analysing this unstructured text field. The other fields, like the timestamp, sender, log facility, and priority, of the events will be ignored for now. Below, we give examples from a syslog file, which has 20.000 lines and occupies 6 MB of space; it is an Excel file in CSV format (values separated by ":") covering the events of about 2 days. A small selection of text messages shows the diversity of the events in the file:

```
07.430: %SYS-5-CONFIG_I:
   Configured from console by mdoess.k5 onvty0 (23.80.40.147)
%CRYPTO-6-AUTOGEN: Generated new 768 bit key pair
%SSH-5-ENABLED: SSH 1.99 has been enabled
%LINEPROTO-5-UPDOWN: Line protocol on Interface
   FastEthernet0/26, changed state to down
%LINK-3-UPDOWN:
   Interface FastEthernet0/26, changed state to down
```

```
14.272: %OSPF-5-ADJCHG:
   Process 1, Nbr 23.80.248.135 on Serial0/2/1
   from FULL to DOWN, Neighbor Down: Interface down or detached
13.522: %IPPHONE-6-REGISTER_NEW:
   ephone-1:SEP00146A62D078 IP:23.80.250.62
   Socket:1 DeviceType:Phone has registered.
13.743: %IPPHONE-6-UNREGISTER_NORMAL:
   ephone-1:SEP00146A62D078 IP:23.80.250.62
   Socket:1 DeviceType:Phone has unregistered normally.
System: SNMP configuration change.
   SNMP access control 2 access type. 0x0003
Control Manager: 7035: Y068DPK1\a0681634:
   The control statement "start" was sent successfully
   to the service "Eventlog to Syslog".
Control Manager: 7036: Service "McAfee McShield"
   now is in the status "stopped".
Control Manager: 7036: Service "McAfee McShield"
   now is in the status "executed".
```

3.1 Removing Digits from Event Messages

The simplest analysis method involves manual reviewing of the file with a standard file viewer and searching for a message pattern. We will now present a method, which improves the overview of the file considerably without much effort: The central step is to replace all consecutive occurrences of digits by a single placeholder character, e.g., `'9'`, because in most cases numbers identify values (like a time or address). The reduced messages can then be sorted and duplicates can be dropped. This reduces the 20.000 lines to 1.048 different lines (patterns). For example, we find the patterns:

```
%LINK-9-UPDOWN: Interface FastEthernet9/9, changed state to up
%SSH-9-ENABLED: SSH 9.9 has been enabled
```

3.2 A Modular Event Format

Text messages are intended for human readers, and only a few formatting standards exist. Over time, many formats evolved, all of which need special processing. So we cannot expect a parser to handle all of them. Instead, parser elements for new types of messages have to be created, and the parser has to be easily *extendable*. The following message, for example, taken from the mentioned log file, is expressed by the XML element below:

```
'1215009055419000','31087: Jul 2 16:30:54:','unknown',
'%LINEPROTO-5-UPDOWN: Line protocol on Interface
FastEthernet0/26, changed state to up','23','5',...
```

```
<event source="syslog" timestamp="1215009055419000"
   message="%LINEPROTO-5-UPDOWN: Line pro..."
   sl_facility="23" sl_priority="5" ... />
   <content type="cisco" severity="5" mnemonic="UPDOWN"
      facility="LINEPROTO" text="Line protocol on Interf..."/>
   <content type="updown" new_state="down"
      iface="FastEthernet0/26"/>
</event>
```

The attributes of the *event* element describe general values provided by the logging facility. Each parsing step creates an additional *content* subelement storing the extracted information. Here, the first *content* element of type *cisco* contains the fields common for all cisco events. The second *content* element provides the special values of this kind of line protocol message. This modular format allows the parsing of general templates. Furthermore, we can refine these patterns by defining additional parsing rules. Such general templates are defined, e.g., by hardware or software vendors like Cisco or Microsoft.

3.3 Parsing and Grouping

In order to build an extendable parser, we formulate each parser element using transformation rules in the PROLOG–based XML transformation language FNTRANSFORM [11]. The rules will be applied repeatedly, until no more rules match. This permits transformations to refine the result of previous transformations. We have also developed more refined parsing techniques based on *extended definite clause grammars*, which can be used for elegantly specifying and parsing more complex structures. Such grammar rules have been applied, for example, to electronic dictionaries [10].

We apply the following incremental approach for grouping event messages: The log file itself is the initial group. In each step we take a large group, identify a frequent pattern, write a new parser rule and split the group accordingly. The splitting can be done automatically according to the type, which we store in each event. Repeating this step, creates a tree structure of patterns. The nodes near the root represent rather general patterns, whereas the leafs represent the most specific ones. In the case of the example log file, we can obtain the tree in Figure 1. The nodes *cisco* and *cisco/updown* correspond to the previously mentioned templates. The first numbers of an inner node, describes the number of events matched by this node and none of its children, the second one is the number of events matched by the whole subtree. Leafs are marked only with a single number accordingly.

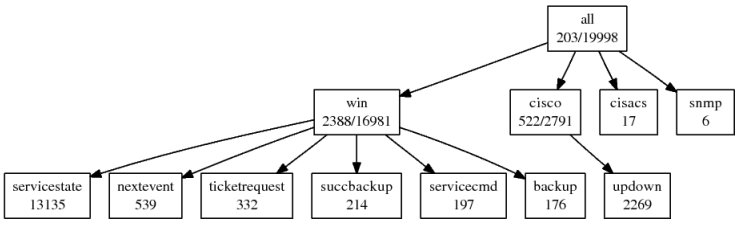

Fig. 1: Hierarchy of Parser Modules

At this stage, reviewing the groups reveals, that the *win* group with 2.388 matching events and 847 patterns is the most inhomogeneous one. All others are homogeneous already. The used file clearly has a strong dominance of Windows events. In log files with more diverse events, we would find more formats of different vendors and logging facilities at the top layer.

While creating the parsers for this example, we also noticed patterns of syntax elements like lists of key/value pairs or the dotted–decimal notation of IP addresses. These patterns can be handled by parser modules, which are used orthogonally to the tree structure. For example, we find a list pattern in many Windows messages. In the two patterns below, the lists follow the word $SYSTEM$:.

```
Security: 9: ...SYSTEM: login attempt from: ME
   account: lock workstation: Y9 error code: 9xC9A
Security: 9: ...SYSTEM: user logout:
   user name: Y9 domain: Y9 login type: 9
```

The list pattern has the form "$a1$: $v1$ $a2$: $v2$...", where a_i and v_i are key/value pairs. After defining a parser for this format, we have implemented a very generic transformation rule for extracting all such lists from Windows event messages.

4 Temporal Data Mining Workflow

In this section, we will describe our declarative framework for temporal data mining. For the data reduction phase (preprocessing), declarative programming in PROLOG was very suitable, since the methods can be flexibly adapted and extended. Our approach also allows flexible options for analysis, considering, e.g., the inclusion of background knowledge. In a prototypical implementation, we have also formulated the central temporal data mining algorithms in PROLOG; since the performance was no problem, we have postponed a possible reimplementation in a standard procedural language. In Section 4.4 we also apply text mining approaches for reducing the number of words.

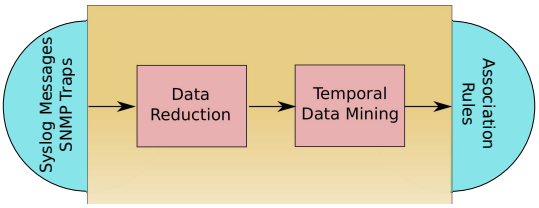

Fig. 2: Temporal Data Mining Workflow

We chose to store the events uniformly in XML. In the case of parsed events, several properties are available. Unparsed events reveal only the basic values like the timestamp, the text message, and the sender address.

4.1 Combining Sequence Analysis and Message Analysis

The event sequence analysis of [5, 7] requires to assign an event type to each event. The event types could be identified by integers, and in the case of text messages, each type would describe a set of messages, i.e. the type describes a pattern. So, defining event types, requires approximately as much work as implementing the parser functions. The presented approach combines sequence and message analysis. Instead of a simple event type, we assign each event several key/value pairs. The set of these pairs is called *event data*, and in the case of a single pair or value we also call it *event type* in the following.

For unparsed text messages, we follow the approach of SLCT [12] and split the message into words. Every word is a value and the key is the word's position in the message. For parsed messages more meaningful data, like the event types of [5, 7], can be added instead. We define a selection predicate using only the words of each text message. It selects the required attributes $timestamp$ and $message$ with expressions of the PROLOG–based XML query language FNQUERY [11], and then splits the message into words. At last, the words are enumerated in their order. Then we consider each position/word pair as an event on its own by inheriting the timestamp of the event; we call these events *subevents*. Because every original event had a unique timestamp, we can still reconstruct the event from its subevents. Subevents will show similarly strong correlations in a temporal analysis as were found in the non–temporal analysis of SLCT.

Since words describe the attributes of a message, temporal relations between attributes of events correspond to relations between subevents. Applying temporal analysis on the subevents will reveal these relations, although at no step manual classifiction of events was necessary. We create the event sequence with unique timestamps as a list of pairs. Each pair (T, E) consists of the event's timestamp T and the event data E; for convenience, we call such pairs *event pairs*. Moreover, an *episode* $\alpha = (P_1, \ldots, P_n)$ is a sequence of event pairs P_i.

4.2 Sliding Windows

The WINEPI algorithm is based on sliding a window over the event sequence. At each position of the window, it considers the set (or multiset) of visible events; the order of the events is irrelevant. For our analysis, we are only interested in the content of each window and not in its time bounds. A call $window(\mathit{Win})$ providing all possible instances of the sliding window in the variable Win as a list of events by backtracking would be nice, but has some deficiencies. Firstly, we are not interested in empty windows. These are unneeded for frequency analysis, and the total number of windows can be computed from the sequence's bounds. Secondly, depending on the distribution of the events, many consecutive windows share the same content. Furthermore, the number of windows – but not the number of different contents – depends heavily on the timestamps' resolution. To avoid these issues, we have implemented a slightly different call $window(\mathit{Win}, \mathit{Rep})$, which provides all windows in the variable Win. Several identical successive windows are bound only once to Win. Rep specifies the repetition of the window Win. The rest of the algorithm, i.e. the candidate generation and the controlling loop, is equivalent to the standard apriori algorithm. Per option $injective$, one can decide between injective and not injective episodes, i.e. an episode may contain the same event type several times.

Using these definitions, we can already apply the frequent episodes analysis on events. For a first analysis, a rather small window size of 10 seconds should be appropriate. After having filtered (cf. Section 4.6) the most frequent episodes with a small temporal extent, we can select larger window sizes. The minimum frequency is difficult to preselect, because a single episode can occur in several windows. For example, a single occurrence of an event is counted in 100 windows, if the timestamps are given in 10–ths of a second and the window size is 10 seconds. But an episode spanning 10 sec-

onds occurs in exactly one window. We decided to use a minimum frequency of 0.01 and selected appropriate settings in the following sections to obtain comparable results.

Using the words of the original text messages as event types, only three frequent injective and parallel episodes are found in the log file with 20.000 messages:

```
1-episodes: [ __ "stopped". ]-0.03799, [ __ "executed". ]-0.038002
2-episodes: [ __ "stopped". , __ "executed". ]-0.0376735
```

Here, __ stands for the text `Control Manager: 7036: Service "McAfee McShield" now is in the status`. Inspecting the frequencies of the 1– (single events) and 2–episodes (combinations of events) reveals, that the two start/stop messages of the antivirus software almost always occur nearby each other.

4.3 Episode Rules

Such temporal relations are described by episode rules, which are similar to association rules. We can apply the algorithm for association rules without modification.

Given the list of frequent itemsets (or frequent episodes), we search for an itemset A and a subset B of A, such that the *confidence* $conf = f(A)/f(B)$ is larger than a given minimum confidence threshold; here, $f(I)$ is the frequency of an itemset I. The result is an association rule $B \rightarrow A \setminus B$. If we apply the filtering on the set of frequent episodes, which we found above, then the following episode rules with a confidence above 0.99 are found. For brevity, we replaced some text passages with an underscore.

```
( __ "stopped". -> __ "executed". ):[0.99167, 0.0376735]
( __ "executed". -> __ "stopped". ):[0.99136, 0.0376735]
```

In the sequence of subevents we find too many and a lot of redundant episode rules. From a set of rules, all rules subsumed by other rules can be dropped. A rule $r : A \rightarrow B$ *subsumes* a rule $s : C \rightarrow D$, if $A \subseteq C$ and $D \subseteq B$; the frequency is not important here. In this case, rule s can be deduced from r, because of the trivial rules $C \rightarrow A$ and $B \rightarrow D$; this means, that s provides no additional information except the confidence, which is not already provided by r. We ignore the confidence here, because we are interested in the compliance with the rather high confidence threshold and not the exact value. Of course, other variations are thinkable.

4.4 Data Reduction based on Most Specific or Equivalent Words

Searching for frequent episodes in the sequence of subevents, i.e. on the words of the messages, produces a huge number of frequent episodes. Many of these episodes have no temporal extent, but occur in a single message, because the subevents of a message show a strong correlation. All subsets of a frequent message's words will be identified as frequent episodes. That is just too much to calculate and provides no new information. The main cause of this problem is the number of words per message: the 20.000 messages of the example file produce 259.735 words.

Therefore, in this section, we will present two methods for reducing the number of words or event data in general, before we apply frequent episode mining. Prior to any other reduction or analysis, we can drop all infrequent words, as they appear neither in frequent itemsets nor in frequent episodes. Dropping words with a frequency below 0.05 lowers the number of words from 259.735 down to 222.026.

Most Specific Words. Text messages are highly redundant, especially if they contain regular sentences. All redundant words can be dropped and the remaining words will suffice to identify the type of the message. Such relations are described by association rules treating the messages as transactions and the words as items. The interpretation of a rule $a \rightarrow b$ is that in a message containing the word a, we will also find the word b. If this rule has a high confidence, then we can drop the word b. Afterwards, b can also be reconstructed in all messages containing a.

The effect of this reduction is, that each message is described by as few words as possible. These words are the *most specific* ones, and unspecific words occuring in different types of messages are removed. For example, the following message is reduced to the three words/position pairs `(1,'%LINK')`, `(5,'FastEthernet0/26')` and `(9, up)`. From the first pair all remaining words can be derived.

```
%LINK-3-UPDOWN: Interface FastEthernet0/26, changed state to up
```

Applying this reduction on the log file lowers the word count from 222,026 to 20,793. Afterwards, a temporal analysis on remaining words can only discover relations between such very specific types of messages and more general patterns are lost.

Equivalent Words. Another approach is to find *equivalent words* in the messages. Of two equivalent words, only one has to be kept, the other does not carry any more information and can be dropped, or the two words can be merged into one *compound word*. Two words are equivalent, if they always occur together. In data mining, we can only determine such relations with a certain frequency, of course.

At first, we search for frequent wordsets in the messages, i.e. itemsets in transactions. We only consider frequent 1– and 2–itemsets. Similar to finding association rules, the next step is finding association equivalences with a high confidence. From a frequent 2–itemset $\{a, b\}$ we deduce the association equivalence $a \leftrightarrow b$ with the confidence given by: $conf(a \leftrightarrow b) = \min\{f(a)/f(b), f(b)/f(a)\}$; here, $f(i)$ is the frequency of an item i.

According to [12], applying the apriori algorithm for finding frequent sets of words in messages results in exponentially (i.e., for 2–itemsets: quadratically) many candidates, nearly all of which are infrequent. It is more appriopriate to deduce candidates from each message and check these candidates' frequency, which occur at least once. We can compute all k–itemsets with $k \leq MaxIt$ by applying the apriori algorithm only after the $MinIt$–th iteration. In order to calculate only the 1– and 2–itemsets without utilising the apriori algorithm, we set $MinIt = MaxIt = 2$. From the frequent 1– and 2–itemsets, we calculate all equivalences with high confidence by backtracking.

The 20.000 messages reveal about 480 word equivalences with a confidence of at least 0.95. Using these equivalences, the 222.026 frequent words can be reduced to 48.318 words. For example, the words of the antivirus message from above are combined into two groups. The first group describes, that the service "McAfee McShield" changed its state, and the second indicates the new state, the service changed to. The listing below shows the groups in detail. This example shows, that the grouping by association equivalences can be as accurate as a manual type definition.

```
[ (1, Control), (2, Manager), (3, 7036), (4, Service), (5, "McAfee),
  (6, McShield"), (7, now), (8, is), (9, in), (10, the), (11, status) ]
```

```
[ (12, "stopped") ]
[ (12, "executed") ]
```

In contrast to the reduction to the most specific words, which removed the words describing general message types, the reduction based on equivalent words keeps keywords for different degrees of abstraction, and more general relations can be discovered.

Frequent Episode Mining. With the number of words reduced to a more feasible number, we can apply the frequent episode algorithm and the rule discovery on the sequence of words or word groups. With the same settings as before (window size of 10 seconds, minimum frequency of 0.01 and injective episodes), we find three rules, which provide exactly the same information as the rules discovered before from the sequence of messages. For clarity we omit some words of the groups, which are already listed above:

```
( (12, "stopped") -> (12, "executed"), [(1, Control) __] ):0.99
( (12, "executed") -> (12, "stopped"), [(1, Control) __] ):0.99
( [(1, Control) __] -> (12, "stopped"), (12, "executed") ):0.98
```

We notice a deficiency of using parallel episodes to find relations of subevents. The rules do not describe the temporal extent of the affected events, and accordingly, the reference to a relation between occurrences of whole text messages is unclear.

Let $\{A\} \rightarrow \{B, C\}$ be the third rule with the actual words replaced by the variables A, B and C. Then we cannot tell, if this rule describes primarily three different occurrences of messages or only two, because the events may coincide with each other. A single message is not covered by this rule, because B and C exclude each other. In the case of two messages it is unclear, if A and B belong to a single message or instead A and C. These rules do not comprise any temporal extent, and especially, do not describe any ordering because we used parallel episodes. It is important to keep in mind, that the episodes discovered by the sliding window approach only identify frequent occurrences of events in a time span given by the window size. Actually, there can be several occurrences in a single window.

4.5 Minimal Occurrences of Episodes

For comparison, we discuss another approach using minimal occurrences and serial episodes based on the MINEPI algorithm in this section. Every episode α is now additionally augmented with the set μ_α of its minimal occurrences, i.e. the minimal intervals containing the episode.

1–Episodes. In analogy to the apriori algorithm, we begin by determinining all 1–episodes. The minimal occurrences of a 1–episode (A) are given by the occurrences of the event A. If A occurs at time S, then a minimal occurrence of (A) is the interval $[S, S]$. We can group all minimal occurrences of all 1–episodes.

Candidate Generation. From a set of frequent serial episodes, larger episodes can be created analogously to the apriori algorithm. Two serial episodes α and β can be concatenated to an episode γ, if the suffix δ of α is also a prefix of β: if $\alpha = \alpha' \cdot \delta$ and $\beta = \delta \cdot \beta'$, then $\gamma = \alpha' \cdot \delta \cdot \beta'$. According to [7], the minimal occurrences μ_γ of γ can

be calculated from μ_α and μ_β by merging pairs of intervals: Let $[S_A, E_A] \in \mu_\alpha$ and $[S_B, E_B] \in \mu_\beta$ be minimal occurrences of α and β respectively, such that $S_A < S_B$ and $E_A < E_B$. If there is no other, later minimal occurrence $[S'_A, E'_A] \in \mu_\alpha$, where $S_A < S'_A$, with the same property, then $[S_A, E_B]$ is a minimal occurrence of γ. For example, for the sequence $\langle(1, a), (2, b), (3, a), (4, c), (5, b), (6, c), (7, d)\rangle$ and the episodes $\alpha = (a, b, c)$ and $\beta = (b, c, d)$, we get $\mu_\alpha = \{[1, 4], [3, 6]\}$, $\mu_\beta = \{[5, 7]\}$, $\gamma = (a, b, c, d)$, and $\mu_\gamma = \{[3, 7]\}$.

We enforce the intervals $[S_A, E_A]$ and $[S_B, E_B]$ to be different, because we allow non–injective episodes. This can be seen in the following simple example. In the case of injective episodes, this restriction can be left out. E.g., in the sequence $\langle(1, a), (2, a)\rangle$ the episode $\alpha = \beta = (a)$ has the set $\mu_\alpha = \{[1, 2], [2, 3]\}$ of minimal occurrences, and the combination $\gamma = (a, a)$ has only a single minimal occurrence: $\mu_\gamma = \{[1, 3]\}$.

Beginning with the 1–episodes, we iteratively combine k–episodes to get candidate $(k + 1)$–episodes. From these candidates only the frequent ones are selected. In this process we find all frequent episodes.

Rules. As we use serial episodes and minimal occurrences in this section, we have to adapt the previous rule algorithm, which used parallel episodes and relative frequencies (based on windows). Different kinds of rules can be derived from serial episodes; we describe only the two most understandable kinds.

Forward rules have the form $\alpha \to \beta$, where α and β are serial episodes. Their interpretation is that if α has a minimal occurrence $[S, E]$, then the concatenation $\gamma = \alpha \cdot \beta$ has a minimal occurrence $[S, F]$ (with $E \leq F$). Less formally spoken, an occurrence of α is followed by an occurrence of β. The maximal extent of all occurrences is given by the same upper bound. That is, we do not use two different time bounds for each rule, but only one upper bound for the whole rule. This reduces the parameters for the algorithm, and nonetheless, rules with a smaller temporal extent are still found. *Backward rules* $\beta \leftarrow \alpha$ mean: if α has a minimal occurrence $[S, E]$, then the concatentation $\gamma = \beta \cdot \alpha$ has a minimal occurrence $[R, E]$ (with $R \leq S$). In other words, an occurrence of α is preceded by an occurrence of β. In both cases, the confidence of such rules r is easily calculated as $conf(r) = f(\gamma)/f(\alpha)$, as in the case of association rules.

Results. In the 20.000 lines of the example log file, we discover the same relation as before. The upper bound was set to 10 seconds, the minimum support to a $1/10$ of the word count (20,793), and the minimum confidence was set to 0.90.

```
(12, "stopped") -> (12, "executed")
(12, "stopped") -> (1, Control) __ (11, Status)
```

But this time, the rules are more expressive. We can interpret both rules together, because their left hand sides are identical. They describe that a message including the word *stopped* is usually followed by a message including the words of the right hand sides (`Control __ executed`) within 10 seconds. Again, we have omitted some words of the word groups described in Section 4.4.

Analysing the Minimal Occurrences. The minimal occurrences of the concatenation (i.e., γ from above) of a rule's left and right hand side indicate the rule's temporal expansion, which can actually be much smaller than the given upper bound.

Therefore, we analyse the lengths of the minimal occurrences. Several thousands of values can be visualised as a histogram. But a histogram does not necessarily reveal dense regions, if the values differ slightly as is the case with distances between events. It seems more appriopriate to apply a clustering algorithm on the length values. We selected the common *hierarchical clustering* method. Beginning with a cluster for each value, clusters too close to each other are merged repeatedly. For our experiments, we used the distance between the arithmetic means of two clusters to decide their proximity.

In the case of the above rule, the results are trivial. The two events occur in immediate succession, i.e. all minimal occurrences have a length of two 10–ths of a second. More interesting results have been found in a comprehensive case study, which we cannot present here due to the limited space; this will be part of an extended version of the paper.

4.6 Filter Rules for Event Messages

In order to discover additional correlations, it is reasonable to filter already known relations, which would clutter up the results unnecessarily. A first filter would replace the previously shown two messages of the antivirus software with a single *stop/start* message. Analysing the reduced event sequence will then reveal, that these messages repeat approximately every 5 minutes. Therefore, we need other filter rules, which are the combination of repeating events and the deletion of a rule's consequent.

For a replacement rule $replace(Episode, Replace, WinSize)$ the events matching the serial episode $Episode$ are removed from the sequence, and the replacement $Replace$ is inserted at the time of the first event. $WinSize$ determines the time span in which the episode has to occur. Moreover, each event type in the episode can match a more specific type. For example, the event type or event data X matches the data Y, where

```
X = [(1,'Service'), (2,'McAfee')],
Y = [(1,'Service'), (2,'McAfee'), (3,'stops')].
```

5 Conclusions

We have presented two selected algorithms for the discovery of episode rules and gave insights into the possible results, providing the basis for future research about these rules. As a next step, we are planning to integrate the presented solution into the network management solution StableNet, which covers fault, performance and configuration management in one product. StableNet has been used by large enterprise and telco customers for several years. The integration gives us the possibility to verify the results in large data networks. We are expecting that the discovery of rules will simplify the setup of fault management in StableNet dramatically. At the moment, rules have to be defined by an experienced user and refined later. We are expecting to simplify and speed up this process with the presented approach, such that we can react earlier to changes in the network.

The algorithms can be further extended to also support the discovery of serial episodes using window–based frequency and parallel episodes using minimal occurrences. Furthermore, we can consider complete or strictly closed episodes [13, 9]. The

combination of non–injective serial episodes and subevents is still an issue, because several subevents share the same timestamp. At the moment, we have solved this by enforcing the minimal occurrences to contain only one event per timestamp and accordingly only one event per message, but some interesting rules require message patterns consisting of several words.

While applying the filter rules, the event filter could at the same time determine the confidence and the outliers of each filter rule. The confidence would then indicate the quality of the filter rules and the incorporated knowledge with respect to the latest events. The outliers indicate particularly interesting events. Furthermore, causal analysis of event sequences, e.g., [3] complementing the association and correlation analysis, would be interesting as well, since it could directly provide more actionable knowledge.

References

1. A. Achar and S. Laxman and P. Sastry. *A Unified View of the Apriori–Based Algorithms for Frequent Episode Discovery.* Knowl. and Inf. Systems, 31 (2), pp. 223–250, 2012.
2. J. Chen and H. Jie and H. Hongxing and G. Williams and J. Huidong *Temporal Sequence Associations for Rare Events.* In: Advances in Knowledge Discovery and Data Mining, pp. 235–239, Springer, 2004.
3. E. Chuah, G. Lee, W. Tjhi, S. Kuo, T. Hung, J. Hammond, T. Minyard and J. C. Browne. *Establishing Hypothesis for Recurrent System Failures from Cluster Log Files.* Proc. 9th IEEE International Conference on Dependable, Autonomic and Secure Computing, pp. 15–22, 2011.
4. L. Pfleger de Aguiar and V. A. F. de Almeida and W. Meira. *Mining Redundant Industrial Alarm Occurrences with Association Rules Extraction and Complex Networks Modeling.* Journal of Computational Methods in Science and Engineering, 11, pp. 15–28, 2011.
5. M. Klemettinen, H. Mannila, and H. Toivonen. *Rule Discovery in Telecommunication Alarm Data.* Journal of Network and Systems Management, Vol. 7, No. 4, pp. 395–423, 1999.
6. S. Laxman and P. S. Sastry. *A Survey of Temporal Data Mining.* Sadhana, Academy: Proceedings in Engineering Sciences, Vol. 31, pp. 173–198, 2006.
7. H. Mannila, H. Toivonen, and A. I. Verkamo. *Discovery of Frequent Episodes in Event Sequences.* Data Mining and Knowledge Discovery 1, pp. 259–289, 1997.
8. N. Méger and C. Rigotti. *Constraint–Based Mining of Episode Rules and Optimal Window Sizes.* Proc. 8th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD), pp. 313–324, Springer, 2004.
9. N. Tatti and B. Cule *Mining Closed Strict Episodes.* Data Mining and Knowledge Discovery 25 (1), pp 34–66, 2012.
10. C. Schneiker, D. Seipel, W. Wegstein, and K. Prätor. *Declarative Parsing and Annotation of Electronic Dictionaries.* Proc. 6th International Workshop on Natural Language Processing and Cognitive Science (NLPCS), 2009.
11. D. Seipel. *Processing* XML*–Documents in* PROLOG. Proc. 17th Workshop on Logic Programmierung (WLP), 2002.
12. R. Vaarandi. *A Data Clustering Algorithm for Mining Patterns from Event Logs.* Proc. IEEE Workshop on IP Operations and Management, 2003.
13. J. Wu and L. Wan and Z. Xu. *Algorithms to Discover Complete Frequent Episodes in Sequences.* In: New Frontiers in Applied Data Mining, pp. 267–278, Springer, 2012.