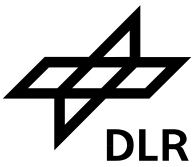

Automatisierte Analyse von Logdateien mit Methoden des maschinellen Lernens

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Engineering

in der Studienrichtung Informationstechnik

von

David Scholz

Bearbeitungszeitraum:	27.06.2016 bis 19.09.2016
Kurs, Matrikelnummer:	MA-TINF13ITIN, 1366944
Unternehmen:	Deutsches Zentrum für Luft- und Raumfahrt e. V.
Abteilung, Standort	Intelligente und verteilte Systeme, Köln
Betreuung durch:	Prof. Dr. Rainer Colgen
	Dr. rer. nat. Brigitte Boden

Eidesstattliche Erklärung

Gemäß §5 (3) der „Studien- und Prüfungsordnung DHBW Technik“ vom 22. September 2011.

Ich habe die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Köln, den 13. September 2016

Zusammenfassung

Moderne Softwaresysteme protokollieren ihre Aktivitäten in sogenannten Log-Dateien. Diese sind im Fehlerfall oft der einzige Anhaltspunkt für die Fehlersuche. Das manuelle Durchsuchen der Log-Dateien durch den Menschen ist aufgrund deren Größe und Komplexität mühselig und fehleranfällig.
Eine Log-Datei besteht aus sogenannten Log-Events, die bestimmte Ereignisse innerhalb des Softwaresystems repräsentieren. Mehrere Log-Events können einen Zusammenhang aufweisen. Sie bilden dann eine sogenannte Log-Event-Sequenz.
Es existieren Algorithmen, die aus einzelnen Log-Events Muster generieren, um Log-Dateien automatisiert analysieren zu können. Dabei beschränken sich die aktuellen Algorithmen auf einzelne Log-Events ohne dabei einen Zusammenhang zwischen mehreren Log-Events in Betracht zu ziehen.

In dieser Arbeit wird ein Algorithmus entworfen und implementiert, welcher automatisiert aus mehreren logisch zusammengehörigen Log-Events, den Log-Event-Sequenzen, Muster erzeugen kann. Hierbei werden Methoden des überwachten maschinellen Lernens eingesetzt, um anhand von speziell aufbereiteten Trainingsdaten Muster zu generieren. Der Algorithmus kann dazu genutzt werden Log-Dateien nach Anomalien und Fehlern zu durchsuchen. Er fasst die analysierten Log-Dateien in einem für den Menschen leicht verständlichen Bericht zusammen.
Der Algorithmus wird anhand von Log-Dateien der verteilten Integrationsumgebung Remote Component Environment (RCE) erprobt. RCE wird am Deutschen Zentrum für Luft- und Raumfahrt e.V. (DLR) entwickelt.
Abschließend werden mögliche Erweiterungen des Algorithmus für die Zukunft diskutiert.

Abstract

Modern software systems log their activities in so-called log files. The written log file often is the only reference point for debugging in case of an error. Manually identifying the cause of the problem by humans is time-consuming, laborious and error-prone, due to size and complexity of the log files.
Log data consists of so-called log events which represent specific incidents in the software system. Several log events may have a common context. They form a so-called log event sequence.
There are algorithms available which detect patterns from log events, for the automated analysis of log files. However, the current algorithms are limited to single events without considering a common context between several events.

The objective of the thesis is the design and implementation of an algorithm, which generates patterns autonomously from log event sequences. The algorithm uses methods of supervised machine learning to generate patterns from specially prepared training data. The algorithm can be used to determine anomalies and errors in log files. The analysis results are summarized in a report which is easily understandable by humans.
The algorithm will be tested based on log files from the distributed integration environment Remote Component Environment (RCE).
Furthermore, possible extensions of the algorithm will be discussed.

Inhaltsverzeichnis

Abkürzungsverzeichnis

RCE	Remote Component Environment
DHBW	Duale Hochschule Baden-Württemberg
DLR	Deutsches Zentrum für Luft- und Raumfahrt e.V.
RCP	Rich Client Platform
GUI	Graphical User Interface
JSON	JavaScript Object Notation
XML	Extensible Markup Language
JDK	Java Development Kit

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Algorithmenverzeichnis

Vorwort

Bei der vorliegenden Arbeit handelt es sich um meine Bachelorarbeit zur Erlangung des akademischen Grades „Bachelor of Engineering“ in der Studienrichtung „Informationstechnik“. Mein Studium habe ich an der Dualen Hochschule Baden-Württemberg (DHBW) in Mannheim in Kooperation mit dem Deutschen Zentrum für Luft- und Raumfahrt e.V. (DLR) absolviert. Während meiner Praxisphasen habe ich beim DLR in Köln in der Abteilung Intelligente und Verteilte Systeme in der Einrichtung Simulations- und Softwaretechnik gearbeitet.
Das DLR ist das nationale und führende Forschungszentrum der Bundesrepublik Deutschland für Luft- und Raumfahrt. Neben diesen namensgebenden Einsatzgebieten liefert das DLR, eingebunden in nationale und internationale Kooperationen mit verschiedensten Forschungseinrichtungen und Industrieunternehmen, maßgebliche neue Erkenntnisse in den Bereichen Verkehr, Sicherheit und Energietechnik. Für das 1907 gegründete Forschungszentrum arbeiten annähernd 8000 Mitarbeiter, welche sich auf insgesamt 16 verschiedene Standorte in ganz Deutschland verteilen. Darüber hinaus unterhält das DLR Büros in Paris, Brüssel, Tokyo und Washington D.C. (vgl. [Luf15], Absatz 4).
Die Abteilung Intelligente und Verteilte Systeme entwickelt innerhalb der Einrichtung Simulations- und Softwaretechnik Individualsoftware für DLR interne Institute und forscht in wissenschaftlichen Projekten im Bereich Softwaretechnologie. Sie entwickelt softwaretechnische Lösungen für die Luft- und Raumfahrt sowie verteilte Anwendungen für die multidisziplinäre Zusammenarbeit, Eignungsdiagnostik und Telemedizin. Bei letzteren nimmt die Nutzung mobiler Hardware und Software eine große Rolle ein.
Eine große Sparte in der Forschungsarbeit nimmt die Entwicklung der verteilten Integrationsumgebung Remote Component Environment (RCE) ein. Die vorliegende Arbeit ist vor dem Hintergrund von RCE entstanden.

1 Einleitung

Moderne Softwaresysteme protokollieren in der Regel ihre Aktivitäten während ihrer Ausführung in sogenannten Log-Dateien. Diese Aufzeichnungen können genutzt werden, um aufgetretene Ereignisse zu einem späteren Zeitpunkt nachvollziehen zu können.
Von besonderem Interesse werden die geschriebenen Log-Dateien, wenn ein Fehler im System auftritt und beispielsweise ein Systemadministrator den Fehler beheben muss. Die zum Zeitpunkt des aufgetretenen Fehlers geschriebenen Log-Zeilen sind häufig der einzige Anhaltspunkt für die Fehlersuche.
Darüber hinaus können Log-Dateien Informationen über die Infrastruktur des Systems liefern. Dazu gehören Informationen, die Rückschlüsse auf die Performanz des Systems erlauben, wie beispielsweise Speicher- und CPU-Auslastung während bestimmter Aktivitäten.
Im Allgemeinen ist eine Log-Datei eine Textdatei, in welcher jedes neu auftretende Ereignis von dem System automatisiert an das Dokument angehängt wird.
Eine Log-Zeile beginnt mit einem Zeitstempel und endet in der Regel mit dem nächsten auftretenden Zeitstempel. Man spricht von einem Log-Event (vgl. [Val01], Seite 7). Ein Log-Event besteht aus einem statischen und einem variablen Anteil. Dieser kann sich zur Laufzeit des Systems stetig ändern (vgl. [NT10], Abschnitt 2). Das Schreiben der Log-Dateien übernimmt ein sogenannter „Logger“.

1.1 Motivation

Während der Ausführung eines Softwaresystems entstehen in der Regel sehr große Log-Dateien. Diese nach dem Auftreten eines Fehlers manuell zu analysieren ist zeitaufwendig und fehleranfällig. Aus diesem Grund wird ein Programm benötigt,

welches Log-Dateien automatisiert analysiert. Die aus der Analyse resultierenden Ergebnisse sollen für den Menschen in verständlicher Form in einem Bericht zusammengefasst werden. Dies soll auftretende Probleme schneller für den Menschen erfassbar machen.

Eine besondere Herausforderung stellt das Analysieren von Log-Dateien in einem verteilten System dar. In einem verteilten System schreiben die einzelnen beteiligten Instanzen lokal Log-Dateien, die jedoch untereinander korrelieren können. Folglich muss ein Programm zur Analyse der Daten die notwendige Infrastruktur zur Verfügung stellen, um verteilte Ereignisse zu erkennen.
Die Motivation dieser Arbeit ist es, Log-Dateien der verteilten Integrationsumgebung Remote Component Environment (RCE) automatisiert zu analysieren.

1.2 Ziel der Arbeit

Es soll ein Programm entworfen und implementiert werden, das Log-Dateien automatisiert analysiert. Die von dem Programm durchgeführte Analyse soll aus maschinell gelernten Mustern eine Ausgabe erzeugen. Der Algorithmus soll sich an der Methodik des *überwachten Lernens* orientieren, d.h. ein spezifiziertes Modell wird mit Lerndaten trainiert. Im Anschluss soll der Algorithmus das trainierte Modell nutzen, um unbekannte Log-Dateien analysieren zu können.
Die gelernten Muster sollen den variablen Teil innerhalb eines Log-Events abstrahieren, um ähnliche Log-Events erkennen zu können.
Ein Beispiel für ein Log-Event ist in Quelltext 1.1 dargestellt.

```
1 2016-08-10 15:35:45,125 DEBUG - de.rcenvironment.core.shutdown - BundleEvent
      STARTING - de.rcenvironment.core.shutdown
```

Quelltext 1.1: Beispiel eines Log-Events

In Quelltext 1.1 sind die Ausdrücke „DEBUG “ und „BundleEvent STARTING“ statisch. Der variable Anteil „de.rcenvironment.core.shutdown “ muss in den Lerndaten kenntlich gemacht werden. Der Algorithmus soll schließlich das Muster aus Quelltext

1.2 erzeugen (vgl. [NT10], Seite 2, Absatz 2)[1].

```
1 DEBUG - (.*) - BundleEvent STARTING - (.*)
```

Quelltext 1.2: Beispiel eines erzeugten Musters

Neben der Mustererkennung von einzelnen Log-Events soll der Algorithmus aus mehreren logisch zusammengehörigen Log-Events Muster erkennen können. Die Schwierigkeit ist dadurch gegeben, dass die einzelnen Log-Events in der Log-Datei verteilt sein können und nicht zusammen auftreten müssen.
Der Algorithmus soll mithilfe von Log-Dateien aus RCE erprobt werden.

1.3 Gliederung der Arbeit

Die vorliegende Arbeit teilt sich grob in vier Bereiche auf: Die für das Bearbeiten der Aufgabe notwendigen Grundlagen, die Beschreibung des Entwurfs und der Implementierung des Algorithmus, ein abschließendes Fazit sowie ein Ausblick auf mögliche Erweiterungen.

Im Kapitel *Grundlagen* wird im Abschnitt *Remote Component Environment* die verteilte Integrationsumgebung RCE kurz vorgestellt. Dabei wird Bezug auf die während der Ausführung eines sogenannten „Workflows“ entstehenden Log-Dateien genommen. Im Anschluss wird der Aufbau einer Log-Datei im Allgemeinen sowie für RCE im Speziellen erläutert.
Im Abschnitt *Log-Analyse* werden Techniken für das Erkennen von Anomalien bzw. Fehlern in Log-Events vorgestellt. Die Analyse von Anomalien stellt eine der praktischen Anwendungen der Mustererkennung dar.
Anschließend werden im Abschnitt *Machine Learning für Log-Mining* zunächst allgemeine Grundbegriffe wie „Überwachtes“ bzw. „Unüberwachtes“ Lernen erläutert. Der zu entwickelnde Algorithmus lässt sich als ein Machine-Learning-Verfahren klassifizieren. Folglich ist vor Beginn der Entwicklung das Grundlagenwissen in diesem Bereich notwendig.
Besonderer Bezug wird auf die *Clusteranalyse* genommen, da es sich bei dieser

[1] (.*) stellt einen regulären Ausdruck dar, welcher beliebige Zeichen mit beliebiger Wiederholung erkennt.

um einen der populärsten Vertreter der unüberwachten Lernmethoden im Bereich Event-Mining handelt. Ein spezielles Clustering-Verfahren für das automatisierte Generieren von Mustern wird anschließend näher betrachtet.

Im Kapitel *Entwurf und Modellierung* wird der Algorithmus in seiner Struktur vorgestellt. Es werden zunächst die Anforderungen an den Algorithmus im Abschnitt *Anforderungsanalyse* genauer betrachtet.
Die Struktur der notwendigen Daten für das Training sowie die anschließende Analyse wird im Abschnitt *Das Datenmodell* entwickelt und diskutiert Das Datenmodell ist das Fundament des Algorithmus und wird in all seinen Facetten diskutiert und anhand eines Beispiels erläutert.
Anschließend wird kurz im Abschnitt *Ein- und Ausgabe der Daten* beschrieben wie die notwendigen Daten dem Algorithmus zur Verfügung gestellt bzw. die Ergebnisse persistiert werden.
Im Abschnitt *Beispiel einer Analyse* wird die geplante Umsetzung des Algorithmus anhand einer exemplarischen Log-Datei dargestellt, um die Vorgehensweise und die Anforderungen noch einmal zu verdeutlichen und mit einem Beispiel zu hinterlegen.

Das Kapitel *Automatisierte Mustererkennung in Log-Dateien* stellt den Algorithmus vor. Es wird mit dem Training (Abschnitt *Das Training*) begonnen. Das Training ist für die Mustergenerierung zuständig.
Der Abschnitt *Die Mustererkennung* stellt die Anwendung der gelernten Muster dar. Es werden auftretende Schwierigkeiten diskutiert und Lösungen präsentiert.

Abschließend werden die Ergebnisse im Kapitel *Fazit und Ausblick* zusammengefasst. Dabei wird besonderer Bezug auf Probleme bezüglich der Analyse von verteilten Log-Dateien genommen. Mögliche Erweiterungen werden abschließend diskutiert.

2 Grundlagen

Im folgenden Kapitel werden die Grundlagen, die für das Lösen der Problemstellung notwendig sind, näher betrachtet. Begonnen wird mit einer kurzen Vorstellung der verteilten Integrationsumgebung Remote Component Environment (RCE), da diese Arbeit vor dem Hintergrund dieses Systems verfasst worden ist. Von besonderer Bedeutung ist der Aufbau von Log-Events in Log-Dateien aus RCE.
Anschließend werden Log-Events und Log-Event-Sequenzen formal definiert sowie die klassischen Event-Mining-Techniken kurz vorgestellt. Darüber hinaus wird der Aufbau von Log-Dateien diskutiert sowie deren Analyse nach aktuellem Stand der Technik erläutert. Abschließend wird der grundlegende Hintergrund des maschinellen Lernens dargelegt, welcher eine gesonderte Rolle in der automatisierten Analyse großer Datenmengen einnimmt. Ein Algorithmus für das automatisierte Erzeugen von Mustern aus Log-Events wird vorgestellt. Dieser verwendet ein Attribut-basiertes Clustering-Verfahren, welches keine spezielle Distanzfunktion verwendet.

2.1 Remote Component Environment

In der modernen Wissenschaft gewinnt die rechnergestützte Lösung von Problemen immer mehr an Bedeutung. Dabei entwickeln die jeweiligen Expertenteams mathematische Modelle, um natur- und ingenieurwissenschaftliche Zusammenhänge zu analysieren. Die Modelle werden mithilfe von Programmen auf ein Problem angewandt. Sie sind jedoch häufig auf einen bestimmten Teilbereich beschränkt.
In der Praxis ist es notwendig, diese verschiedenen Teilbereiche miteinander zu verknüpfen, um ein größeres Problem lösen zu können. Dies macht die Zusammenarbeit verschiedener wissenschaftlicher Disziplinen notwendig, was neue Herausforderungen an die Softwareentwicklung stellt. Die von den jeweiligen Expertenteams entwickelten

Programme für die Modellierung des jeweiligen Teilbereichs müssen folglich miteinander verknüpft werden, um ein größeres Modell zu erzeugen und anwenden zu können. Beispielsweise werden im Deutschen Zentrum für Luft- und Raumfahrt e.V. (DLR) Flugzeuge entworfen. Dies macht die Zusammenarbeit mehrerer Disziplinen, beispielsweise der Aerodynamik oder der Antriebstechnik, notwendig. Die dort entwickelten Programme führen numerische Simulationen durch, um bestimmte Flugzeugeigenschaften zu berechnen. Die entstehenden Daten müssen untereinander ausgetauscht werden, um ein Flugzeug in all seinen Facetten entwerfen zu können.
Die Verknüpfung der verschiedenen Programme kann eine verteilte Integrationsumgebung übernehmen. Diese ermöglicht die Kommunikation von Programmen untereinander. Die Programme können auf physisch getrennten Rechnern Daten über ein Netzwerk miteinander austauschen.
Die Abteilung Verteilte und Intelligente Systeme des Instituts Simulations- und Softwaretechnik entwickelt für diesen Zweck seit 2006 das Softwaresystem Remote Component Environment (RCE). In RCE können beliebige externe Programme in einem sogenannten „Workflow“ miteinander verknüpft werden. Ein Workflow kann verteilt sein, d.h. er kann aus beliebig vielen RCE-Instanzen bestehen. Der Output der einen Instanz kann der Input einer anderen Instanz sein. Der entstehende Datenfluss wird von RCE gesteuert.
RCE implementiert für die Ausführung eines verteilten Workflows ein dynamisches Peer-to-Peer Netzwerk, d.h. alle Knoten im Netzwerk sind gleichberechtigt. Es existiert keine zentrale Verwaltungsinstanz.
RCE ist in der Programmiersprache Java geschrieben und basiert auf der Eclipse Rich Client Platform (RCP).

RCE Logging

Jede RCE Instanz schreibt während ihrer Ausführung kontinuierlich Log-Events in eine Log-Datei und nutzt dabei das Logging-Framework „Log4j“ der Apache Software Foundation.

Ein klassisches RCE-Log-Event orientiert sich an dem Muster aus Quelltext 2.1.

```
1 yyyy-dd-MM HH:mm:ss,SSS LOG_LEVEL - MESSAGE
```

Quelltext 2.1: RCE Log-Event Muster

Jedes Log-Event beginnt mit seinem Zeitstempel, gefolgt von dem für Log4j typischen „Log-Level“. Der nachfolgende Text (im obigen Beispiel „MESSAGE“) ist der für das Event spezifische Inhalt und stellt eine Aktivität innerhalb einer RCE-Instanz dar. Wichtig für die spätere Analyse von RCE-Log-Events ist die Tatsache, dass ein Log-Event nicht zwangsläufig durch einen Zeilenumbruch beendet werden muss. Es wird durch ein erneutes Auftreten eines Zeitstempels determiniert.

2.2 Log-Events und Log-Event-Sequenzen

Als „Event-Mining“ wird eine Ansammlung von Techniken und Algorithmen bezeichnet, die aus großen Datenmengen nützliche Informationen extrahieren.
Im folgenden wird der Begriff „Log-Event“ definiert und bezüglich klassischer Event-Mining-Techniken näher untersucht.
Ein Log-Event beginnt im Allgemeinen mit dem Zeitpunkt seines Auftretens. Dieser wird in Form eines Zeitstempels vom Logger an den Anfang des Events geschrieben. Jedes Log-Event entstammt einer Menge von Log-Events E. Jedes Log-Event besteht aus einem statischen und einem variablen Anteil.
Eine Abfolge von geordneten Log-Events wird *Log-Event-Sequenz* genannt. Eine Log-Event-Sequenz beinhaltet einen Start und Endzeitpunkt, sowie eine geordnete Menge von Log-Events. Eine Log-Event-Sequenz ist folglich ein Tripel bestehend aus einer Menge $d \subseteq E$, einem Startzeitstempel t_{start} und einem Endzeitstempel t_{end}. Daraus folgt für die Menge von Event-Sequenzen S:

$$S = \{(d_1, t_{start_1}, t_{start_2}), (d_2, t_{start_2}, t_{end_2}), \cdots, (d_n, t_{start_n}, t_{end_n})\} \tag{1}$$

Der Startzeitstempel einer Sequenz wird dem ersten Event der geordneten Menge d entnommen. Analog dazu verfährt man mit dem Endzeitstempel (vgl. [Vro10], Seite 6-8).
Eine Log-Datei ist schließlich definiert als eine Menge, welche aus einer geordneten

Abfolge von Event-Sequenzen besteht.
Ein Beispiel für eine Log-Event-Sequenz ist in Quelltext 2.2 dargestellt.

```
1 2016-08-23 15:35:45,592 DEBUG - de.rcenvironment.components.script.common -
      BundleEvent STARTING - de.rcenvironment.components.script.common
2 2016-08-23 15:35:45,592 DEBUG - de.rcenvironment.components.script.common -
      BundleEvent STARTED - de.rcenvironment.components.script.common
```

Quelltext 2.2: Beispiel einer Log-Event-Sequenz

Die in Quelltext 2.2 dargestellten Log-Events bilden eine Log-Event-Sequenz. Die Dauer dieser Sequenz ergibt sich aus dem Zeitstempel des ersten Events und dem Zeitstempel des zweiten Events.
In verteilten Systemen kann das Extrahieren der Zeitstempel von Sequenzen ein gesondertes Problem darstellen, da die dazugehörigen Events verteilt sein können. Die Events können dabei in verschiedenen Instanzen entstehen, die nicht zwangsläufig synchronisiert sind.

Log-Event-Sequenzen in großen Datenmengen automatisiert zu analysieren gewinnt immer mehr an Bedeutung. An dieser Stelle werden einige klassische Event-Mining-Algorithmen kurz vorgestellt.
Wie bereits oben erläutert, ist eine Event-Sequenz eine geordnete Menge von Log-Events mit dazugehörigen Start- und Endzeitpunkt. Aus diesen Sequenzen lassen sich Muster extrahieren. Diese Muster werden auch *sequentielle Muster* (engl. sequentiell patterns) genannt (vgl. [Vro10], Seite 23).
Um einen Algorithmus zu entwickeln, der aus Log-Event-Sequenzen automatisiert Muster generieren kann, ist es notwendig die generelle Struktur von Log-Dateien näher zu betrachten. Für das Format bzw. die inhaltliche Syntax existiert kein allgemeiner Standard (vgl. [Val01], Seite 7, Absatz 2), jedoch lassen sich einige Gemeinsamkeiten feststellen, die für die Mustergenerierung bzw. für die spätere Mustererkennung von besonderer Bedeutung ist.

Der genutzte Logger protokolliert die Events in einem fest definierten Output-Format in einem Textdokument. Beim Start des Softwaresystems ist dieses Dokument leer bzw. enthält Log-Events aus vorherigen Durchläufen. Der Logger hängt neu auftretende Log-Events an die Log-Datei an. Jedes Event beginnt mit einem Zeitstempel. Folglich sind diese zeitlich geordnet (vgl. [Kre13], Abschnitt 1). Es wird niemals

bestehender Inhalt verändert oder gelöscht (vgl. [And98], Seite 1, Abschnitt 3). Kommt es zu einem Fehler können die Log-Events vor und nach dessen Auftreten für eine Fehleranalyse genutzt werden (vgl. [KP99], Seite 125, Abschnitt „Write a log file“).

Ein Logger bietet üblicherweise die Möglichkeit an, den Grad der Ausführlichkeit, „verbosity level“ genannt, genauer zu definieren. Es existieren in der Regel die Level:

- Info-Level, das „normale“ Events protokolliert.
- Warning-Level, welches Warnungen protokolliert. Warnungen sind nicht zwangsläufig Fehler, vielmehr handelt es sich um vom Softwaresystem erkannte Anomalien.
- Error-Level, das Fehler-Events protokolliert.
- Debug-Level, protokolliert üblicherweise unkritische Events, konsumiert aber ebenfalls Info- und Error-Events.

Trotz der Tatsache, dass Logging eine etablierte Technik in der heutigen Softwareentwicklung ist, gibt es in der Praxis keinen allgemein anerkannten Standard für den Aufbau von Log-Dateien. (vgl. [YPZ12], Seite 1, Absatz 1 und Seite 2, Absatz 1). Es lässt sich lediglich feststellen, dass ein Log-Event in der Regel aus einem statischen und einem variablen Anteil zusammengesetzt wird. Folglich loggt das System zur Laufzeit Events mit wechselnden variablen Anteilen aus einem Repertoire an fest definierten Log-Events.
Wie in [Xu+09] beschrieben, entscheidet der statische Anteil eines Log-Events über den Event-Typ.

```
2016-08-10 15:35:45,125 DEBUG - de.rcenvironment.core.shutdown - BundleEvent
    STARTED - de.rcenvironment.core.shutdown
```

Quelltext 2.3: Log-Event mit Event-Typ als statischer Anteil

Das in Quelltext 2.3 dargestellte Event beginnt mit seinem Zeitstempel und enthält die statischen Anteile „DEBUG“, welches das Log4j spezifische Log-Level darstellt, und „BundleEvent“, welches ein systemspezifisches Event repräsentiert. Folglich lässt sich das Event mithilfe der statischen Anteile dem Typ „BundleEvent“ zuordnen.

Die Log-Event-Variablen lassen sich in zwei Kategorien einteilen: Identifier und State-Variablen. Identifier (engl. für „Kennung“, kurz auch ID) stellen eine eindeutige Kennzeichnung für ein Objekt innerhalb eines Log-Events dar. Im obigen Beispiel wäre die Log-Event-Variable „de.rcenvironment.core.shutdown“ die Kennung für das gleichnamige Objekt, welches von dem Softwaresystem manipuliert worden ist (vgl. [Xu+09], Seite 3, Abschnitt 2). Die State-Variable wäre „STARTED“ und stellt den aktuellen Zustand des manipulieren Objekts dar[1].

2.3 Log-Analyse

Aufgrund der enormen Anzahl von Log-Events erweist sich eine manuelle Analyse durch den Menschen als komplex und fehleranfällig. Aus diesem Grund steigt die Bedeutung von rechnergestützten Analysetechniken für das Aufdecken von Fehlern in einem Softwaresystem (vgl. [Zwi14], Seite 3).
Die dabei entwickelten Algorithmen lassen sich im Allgemeinen in zwei größere Teilbereiche aufspalten: in die *Anomaly Detection* und die *Fault Detection*. Ersteres versucht Log-Events bzw. Log-Event-Sequenzen zu finden, welche nicht der vorher aufgestellten Menge an Mustern zugeordnet werden können. Zweiteres sucht gezielt nach Sequenzen, welche sich durch aufgestellte Fehler-Muster als Fehler klassifizieren lassen.
Beide Bereiche werden im folgenden genauer beschrieben.

[1] In diesem Fall handelt es sich um ein Log-Event aus RCE. Das Bundle „de.rcenvironment.core.shutdown“ ist vom System gestartet worden.

2.3.1 Anomaly Detection

Verfahren, die sich unter den Sammelbegriff *Anomaly Detection* zusammenfassen lassen, behandeln das Problem Muster in Log-Dateien zu finden, welche nicht zu einem vorher definierten, erwarteten Verhalten passen. Diese Muster werden oft als *Ausreißer* (*Outlier*) bezeichnet und werden in der Praxis dazu genutzt, Fehler in Softwaresystemen zu isolieren (vgl. [CBK09], Seite 1, Abschnitt 1). Beispielsweise könnte eine Log-Event-Sequenz, die während der Kommunikation zwischen zwei RCE-Instanzen als Anomalie erkannt wird, den Indikator für einen Verbindungszusammenbruch darstellen.
Das Erkennen von Anomalien ist verwandt mit dem Erkennen von Rauschen in Datensätzen und deren Entfernen vor der eigentlichen Analyse. Als Rauschen werden Datenpunkte innerhalb einer Punktwolke bezeichnet, welche beispielsweise als Messfehler klassifiziert werden können und somit für eine spätere Analyse nicht betrachtet werden. Folglich unterscheidet sich das Erkennen von Rauschen und Anomalien einzig in ihrer Semantik. Anomalien versucht man gezielt aufzudecken, um diese für eine Analyse näher zu berücksichtigen, während Rauschen in Datensätzen in der Regel gezielt für eine spätere Analyse entfernt wird.

Anomalien lassen sich in drei verschiedene Bereiche einteilen (vgl. [CBK09], Seite 7):

- **Punkt-Anomalien**, stellen die einfachste Kategorie dar und beschäftigen sich mit dem Erkennen von einzelnen Ausreißern, beispielsweise dem Erkennen von einzelnen Log-Events oder Log-Event-Sequenzen, welche sich nicht in die Menge an aufgestellten Mustern einordnen lassen.
- **Kontext-Anomalien**, sind Anomalien, welche nur als solche gelten, wenn sie in einem bestimmten Kontext erkannt werden. Der Kontext muss dabei im Datensatz vorhanden sein und kenntlich gemacht werden. Ein Kontext könnte beispielsweise die Zeitdauer einer Log-Event-Sequenz sein. Diese lässt sich aus dem Start- und Endezeitpunkt berechnen. Eine Log-Event-Sequenz könnte beispielsweise erst dann als Anomalie gelten, wenn ihre Dauer eine gewisse Zeit überschreitet. Es können demnach Fälle auftreten, bei denen ein identischer Datensatz in einem Kontext als Anomalie und in einem anderen Kontext als

„normal“ gilt. Folglich stellt die Behandlung dieses beschriebenen Falls ein algorithmisches Entscheidungsproblem dar.

- Als **Mengen-Anomalien** gelten Datensätze, welche nur in einer bestimmten Verbindung mit anderen Datensätzen eine Anomalie darstellen. Gegeben seien beispielsweise zwei Log-Event-Sequenzen, welche einzeln betrachtet als normal gelten. Treten beide Sequenzen jedoch zusammen auf, stellen sie eine Anomalie dar.

An dieser Stelle ist anzumerken, dass Punkt-Anomalien in jedem Datensatz auftreten können, während Mengen-Anomalien nur in Datensätzen auftreten, bei denen ein kausaler Zusammenhang vorhanden ist. Dies ist besonders für das Erkennen von Log-Event-Anomalien in verteilten Systemen von Bedeutung.

Algorithmen für das Erkennen von Anomalien in Log-Dateien lassen sich in drei Kategorien einteilen (vgl. [CBK09], Seite 10-11):

- **Supervised anomaly detection**, in welcher zwei Klassen erzeugt werden: „Anomalie“ und „keine Anomalie“. Anschließend werden Muster mithilfe von Trainingsdaten extrahiert und jeweils einer der beiden Kategorien zugeordnet. Die Zuordnung sowie das Extrahieren der Muster erfordert das aktive eingreifen eines Menschen. Es handelt sich um ein *überwachtes*(engl. *supervised*) Verfahren.
- **Semi-Supervised anomaly detection**, erzeugt nur Muster für die Klasse „keine Anomalie“. Jeder Datensatz (beispielsweise jedes Log-Event bzw. jede Log-Event-Sequenz), welcher nicht von einem Muster aus der entstandenen Menge erkannt wird, gilt folglich als Anomalie.
- **Unsupervised anomaly detection**, führen die in den anderen Kategorien genannten Schritte durch ohne dabei speziell aufbereitete Trainingsdaten zu benötigen. Es wird häufig die Annahme getroffen, dass normale Log-Event-Sequenzen häufiger auftreten als anormale.

Nach der Analyse eines unbekannten Datensatzes erzeugen die Algorithmen einen klassifizierten Datensatz, welcher entweder jedes Log-Event bzw. jede Log-Event-Sequenz einer Klasse zuordnet oder eine Wahrscheinlichkeit angibt, mit welcher

dieses in die entsprechende Klasse einzuteilen ist (vgl. [CBK09], Seite 11, Abschnitt 3).

2.3.2 Fault Detection

Verfahren, die sich unter dem Sammelbegriff *Fault Detection* zusammenfassen lassen, sind denen der *Anomaly Detection* ähnlich. Sie erzeugen Muster, die als Fehler klassifiziert sind. Folglich werden Datensätze, die von diesen erzeugten Mustern erkannt werden, ebenfalls als Fehler klassifiziert. Diese Vorgehensweise ist problematisch, da alle möglichen Fehlerfälle vorher bekannt sein müssen, um diese erkennen zu können. Folglich werden beispielsweise nicht erkannte Log-Events bzw. Log-Event-Sequenzen als fehlerfrei klassifiziert. In der Praxis wird aus diesem Grund häufiger die *Anomaly Detection* genutzt bzw. mit der *Fault Detection* kombiniert.

2.4 Machine Learning für Log-Mining

Der Begriff „Maschinelles Lernen“ bezieht sich häufig auf das automatisierte Erkennen von Mustern in Daten. Im weiter gefassten Sinne handelt es sich um Algorithmen, die aus Input-Daten Wissen extrahieren. Sie „lernen“ eine Expertise durch Erfahrung. Die gegebenen Input-Daten werden Trainingsdaten genannt und der vom Lern-Algorithmus erzeugte Output repräsentiert die erlangte Expertise (vgl. [SB14], „Preface“ und Seite 19, Abschnitt 1).
Maschinelles Lernen wird vor allem dann benötigt, wenn die zu lösenden Probleme eine gewisse Komplexität übersteigen bzw. ein hohes Maß der Anpassung an neue Gegebenheiten von dem implementierten Programm erwartet wird. Beide Aspekte treffen in der Regel auf die Analyse von Log-Dateien zu. Auf der einen Seite ist das Erkennen von Mustern komplex, auf der anderen Seite können sich Log-Events in der Praxis jederzeit geringfügig ändern. Dies erfordert ein hohes Maß der eigenständigen Anpassung der Analysealgorithmen, da ansonsten jede Änderung innerhalb der Log-Events eine manuelle Anpassung durch den Menschen erforderlich machen würde. In der Regel bieten Lern-Algorithmen ein hohes Maß der Anpassung für die Umgebung, für die sie implementiert worden sind (vgl. [SB14], Seite 22, Abschnitt 3).

Im Allgemeinen wird zwischen **überwachtem** und **unüberwachtem** Lernen unterschieden. Zunächst wird an dieser Stelle das überwachte Lernen näher erläutert. Im Anschluss wird auf das unüberwachte Lernen eingegangen. Dabei wird besonderer Bezug auf die Clusteranalyse genommen.

Beim überwachten Lernen extrahiert der Algorithmus die gewünschte Expertise anhand von gezielt formatierten Trainingsdaten. Diese Trainingsdaten müssen von einem geschulten Nutzer, dem „Lehrer“, dem Algorithmus zur Verfügung gestellt werden. Möchte man beispielsweise, dass das Programm automatisiert Log-Events den richtigen Log-Event-Typen zuordnet, könnte ein überwachter Ansatz entsprechend gekennzeichnete Daten (beispielsweise mit einem Label versehen) zur Verfügung stellen. Aus diesen Daten extrahiert der Lern-Algorithmus abstrakte Muster, die für die Klassifizierung von unbekannten Log-Events genutzt werden können. Der Algorithmus hätte folglich durch die Trainings-Daten gelernt, welches Log-Events zu welchen Typen gehören.
Beim unüberwachten Lernen existiert keine Unterscheidung zwischen Lern- und Testdaten.

Darüber hinaus wird zwischen **aktivem** und **passivem** Lernen unterschieden. Ein aktiver Lern-Algorithmus tritt mit seiner Umgebung zur Laufzeit in Interaktion. Er verändert beispielsweise gegebene Lerndaten. Ein passiver Lern-Algorithmus nimmt dagegen die Rolle eines reinen Beobachters ein. Er verändert die von der Umgebung bereitgestellten Daten nicht. Beispielsweise wäre es passiv, wenn der Algorithmus darauf warten würde, dass der „Lehrer“ Log-Events mit gekennzeichneten Log-Event-Typen zur Verfügung stellt, um die gewünschte Expertise zu extrahieren. In einer aktiven Umgebung könnte der Algorithmus selbstständig versuchen Log-Event-Typen aus unbekannten Daten zu extrahieren und dem Lehrer zur Verifikation vorzuschlagen (vgl. [SB14], Seite 23).

2.4.1 Clusteranalyse

Muster durch Methoden des überwachten Lernens in großen Log-Dateien zu erzeugen, ist mühselig und zeitaufwendig. Darüber hinaus wird bei jeder Änderung eine geschul-

te Person benötigt, welche die gelernten Muster durch weiteres Training ergänzt. In der Praxis nehmen aus diesem Grund unüberwachte Lernverfahren eine große Rolle ein. An dieser Stelle wird die Clusteranalyse allgemein vorgstellt, um anschließend einen Clustering-Algorithmus für das Generieren von Mustern aus Log-Events zu diskutieren.

Bei der Clusteranalyse handelt es sich um ein unüberwachtes Lernverfahren. Die Clusteranalyse fasst einzelne Objekte in Gruppen, sogenannten Clustern, zusammen. Es werden folgende Bedingungen gestellt (vgl. [Sch12], Seite 23. Siehe auch [SV11], Seite 4.):

- Objekte innerhalb desselben Clusters sollen sich so ähnlich wie möglich sein (hohe Intracluster-Homogenität).
- Objekte in unterschiedlichen Clustern sollen sich möglichst unterscheiden (niedrige Intercluster-Homogenität).

Objekte, welche in keines der erzeugten Cluster passen, werden als Ausreißer bzw. Outlier bezeichnet (vgl. [Vaa03], Seite 2, Abschnitt 2). Die Clusteranalyse versucht eine unstrukturierte Menge von Objekten einer Gruppe von Clustern zuzuordnen und „nimmt damit vor allem eine Vorbereitungsfunktion für weitergehende Analysen ein, welche eine exakte Gruppenzuordnung als Prämisse haben.“([SV11], Seite 1, Abschnitt 2).
Im Allgemeinen wird zwischen deterministischen und nicht-deterministischen Verfahren unterschieden[2]. Im deterministischen Fall bedeutetet das, dass man jedes Objekt eindeutig einem Cluster zuordnen kann.
Für die Einteilung der Objekte in Cluster wird eine Distanzfunktion für die Unterscheidung der Objekte benötigt. Die Wahl der Distanzfunktion ist dabei nicht trivial[3].
Ist eine passende Distanzfunktion aufgestellt worden, wird diese dafür genutzt die Objekte den entsprechenden Clustern zuzuordnen.

[2]Die nicht-deterministischen Verfahren werden hier nicht näher betrachtet.
[3]Für genauere Details siehe [SV11], Seiten 4, 6, 9, 10 und 15.

Im Allgemeinen wird zwischen hierarchischen und nicht-hierarchischen Verfahren unterschieden.

Bei den hierarchischen Verfahren handelt es sich um Verfahren, welche Objekte mit der geringsten Distanz zueinander zusammenfassen. Man unterscheidet zwischen zwei Verfahrenstypen (vgl. [Sch09], Seite 18):

- Dem **agglomerativen** Clusterverfahren, bei welchem jedes Objekt zu Beginn ein eigenes Cluster bildet. Ein Bottom-Up-Algorithmus fasst die Cluster in einer Analyse zusammen bis ein einzelnes Cluster bestehen bleibt. Die Menge der Start-Cluster wird mit jeder Iteration verringert. Folglich verringert sich schrittweise die Homogenität (vgl. [Ger09], Folie 3). Nach jeder Iteration müssen die Distanzen neu berechnet werden. Die Einteilung der Objekte in gröbere Cluster ist nicht reversibel (vgl. [Sch09], Folie 19).
- Dem **divisiven** Clusterverfahren, bei welchem alle vorhandenen Objekte zu Beginn ein gemeinsames Cluster bilden. Ein Top-Down-Algorithmus zerlegt die vorhandenen Objekte in kleinere Cluster bis jedes Objekt einem eigenen Cluster zugeordnet wird (vgl. [Pre08], Kapitel „Divisive clustering“). Folglich erhöht sich schrittweise die Homogenität (vgl. [Ger09], Folie3).

Für das agglomerative Verfahren benötigt man spezielle Algorithmen. Diese berechnen die Distanzen zwischen den Clustern. Die Distanz zwischen zwei Clustern entscheidet, ob diese zusammengefasst werden. Die Clusterfusionierung für den Fall, dass jedes Cluster lediglich ein Objekt enthält, ist trivial. Man nutzt die aufgestellte Distanzfunktion. Für den Fall von n Objekten mit $n > 1$ pro Cluster C_k nutzt man sogenannte Fusionierungsalgorithmen.
Beispiel: Sei Γ_v die Menge an Clustern, dann bildet für $v = 0$ die Menge $\Gamma_0 = \{\{x_1\}, \cdots, \{x_n\}\}$ die Start-Cluster für die Objektmenge $\Omega = \{x_1, \cdots, x_n\}$. Gegeben sei ein Distanzmaß D. Man erhält eine neue Clustermenge Γ_v mit $v \geq 1$, indem D minimal wird. Diese Schritte werden wiederholt bis ein Abbruch- bzw. Zielkriterium erreicht ist.
Die verschiedenen Fusionierungsalgorithmen unterscheiden sich durch die Wahl der Distanzfunktion D (vgl. [Ger09], Folie 5, 7). Bei der folgenden Aufzählung werden

immer zwei Cluster C_r, C_s betrachtet[4].

- **Single-Linkage**:
$$D(C_r, C_s) = \min_{x_i \in C_r, x_k \in C_s} d(x_i, x_k) \tag{2}$$

- **Complete-Linkage**:
$$D(C_r, C_s) = \max_{x_i \in C_r, x_k \in C_s} d(x_i, x_k) \tag{3}$$

- **Average-Linkage**: Sei n_r die Anzahl der Objekte in C_r und n_s die Anzahl der Objekte in C_s, dann ist die Distanzfunktion gegeben durch:
$$D(C_r, C_s) = \frac{1}{n_r n_s} \sum_{x_i \in C_r} \sum_{x_k \in C_s} d(x_i, x_k) \tag{4}$$

Es ist im Allgemeinen üblich die „Fusionierungsschritte hierarchisch-agglomerativer Verfahren [...] durch ein Dendrogramm zu visualisieren“([SV11], Seite 21, Abschnitt 1). Bei einem Dendrogramm handelt es sich um ein Baumdiagramm, bei welchem „auf jeder Stufe jeweils zwei Cluster [...] vereinigt werden“([Spr]). Ein beispielhaftes Dendrogramm ist in Abbildung 2.1 dargestellt.
Auf der y-Achse ist die Distanz dargestellt. Folglich ist bei jedem Fusionierungsschritt die Ähnlichkeit der fusionierten Objekte zueinander visuell sichtbar.

Bei den nicht-hierarchischen Verfahren wird im Gegensatz zu den hierarchischen Verfahren eine Startgruppierung, d.h. eine Menge von statischen Clustern, vorgegeben. Diese Startgruppierung wird zufällig erzeugt. Anschließend werden die Objekte in die entsprechenden Cluster verschoben. Diese Schritte werden wiederholt, bis ein Abbruchkriterium erreicht wurde. Dieses Abbruchkriterium bzw. Zielkriterium ist in der Regel die Minimierung der Varianz innerhalb der Cluster. Die Einteilung der Objekte in die verschiedenen Cluster ist dabei reversibel (vgl. [Sch09], Seite 19). Der Nachteil bei den Nicht-Hierarchischen Verfahren ist der hohe sich ergebende Arbeitsaufwand pro Objekt (vgl. [Ort03], Folie 12-13).

[4] An dieser Stelle wird sich auf drei Verfahren beschränkt, um einzig die generelle dahinter stehende Idee zu vermitteln. Für weitere Details siehe [Ger09], Folie 7-10; [Sch09], Folie 20-22; [Ort03], Seite 3-4; [SV11], Seite 18-21.
[5] Quelle: [Spr]

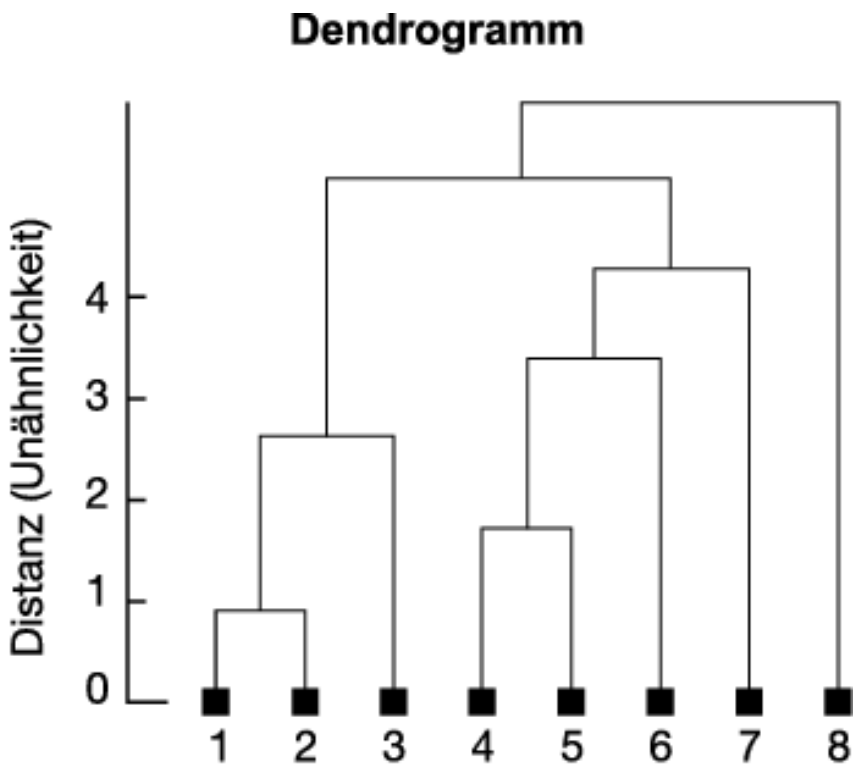

Abbildung 2.1: Beispiel für ein Dendrogramm[5]

2.4.2 Clustering-Algorithmus für Mustergenerierung aus Log-Events nach Risto Vaarandi

Im folgenden Abschnitt wird ein Clustering-Algorithmus vorgestellt, welcher speziell für das Generieren von Mustern in Log-Dateien entwickelt worden ist. Der Algorithmus verwendet dabei keine Distanzfunktion. Beschrieben und entwickelt wurde der Algorithmus von Risto Vaarandi in [Vaa03].

In dem Algorithmus wird ein Datenraum erzeugt, in welchem sich beliebig viele Datenpunkte befinden. Jeder Datenpunkt besitzt eine gewisse Anzahl an Attributen, welche sich in verschiedene Kategorien einteilen lassen. Ein Datenpunkt repräsentiert ein Log-Event und jedes Wort entspricht einem Attribut. Ein Unterraum des Datenraums wird *Region* genannt, wenn in ihm Datenpunkte vorhanden sind, bei denen mindestens ein Attribut übereinstimmt. Man nennt die identischen Attribute *fix*. Existiert genau ein fixes Attribut in dem Unterraum, heißt die Region 1-*Region*.

Ein Beispiel für Datenpunkte eines Unterraums sind in Quelltext 2.4 dargestellt.

```
1 DEBUG - de.rcenvironment.components.script.common - BundleEvent STARTED - de.
      rcenvironment.components.script.common
2 DEBUG - de.rcenvironment.core.scripting - BundleEvent STARTED - de.rcenvironment.
      core.scripting
3 DEBUG - de.rcenvironment.core.datamodel - BundleEvent STARTED - de.rcenvironment.
      core.datamodel
4 DEBUG - de.rcenvironment.core.component.integration - BundleEvent STARTED - de.
      rcenvironment.core.component.integration
```

Quelltext 2.4: Beispiel für Datenpunkte eines Unterraums

Der Algorithmus erzeugt die 1-Regionen mit den fixen Attributen {*DEBUG*}, {*BundleEvent*}, {-} und {*STARTED*}. Tauchen die fixen Attribute innerhalb des Unterraums N mal auf, spricht man von einer *dichten* 1-Region. Der Parameter N wird dabei vom Nutzer festgelegt und ist dem dargestellten Beispiel entsprechend mit $N = 4$ initialisiert.

Die Ausführung des Algorithmus besteht aus drei Schritten:

1. Die **Datenzusammenfassung**, welche alle dichten 1-Regionen innerhalb des Datenraums identifiziert.

2. Das **Clustering**. Der Algorithmus erzeugt die Cluster-Kandidaten, in dem er jedes Log-Event einzeln verarbeitet. Findet der Algorithmus ein Log-Event, welches zu einer oder mehreren dichten 1-Regionen gehört, dann wird ein neues Cluster erzeugt. Die Cluster-Kandidaten werden dabei in einer Tabelle gespeichert. Die Tabelle ist zu ihrem Initialisierungszeitpunkt leer. Jeder Cluster-Kandidat innerhalb dieser Tabelle enthält einen Zähler, welcher inkrementiert wird, sobald ein identisches Log-Event gefunden wurde.

3. Die **Mustererzeugung**, bei welcher der Algorithmus abschließend die erzeugte Tabelle untersucht. Dabei werden alle Cluster-Kandidaten, welche einen Zähler haben, der mindestens der Zahl N entspricht, zu einem Cluster geformt. Anschließend wird mithilfe der fixen Attribute ein Muster erzeugt. Dem Beispiel aus Quelltext 2.4 entsprechend würde sich folgendes Muster ergeben: *DEBUG - (.*) - BundleEvent STARTED - (.*)*, da die fixen Attribute {*DEBUG*}, {*BundleEvent*}, {-} und {*STARTED*} nach dem Parameter N als dicht gelten.

Es lässt sich feststellen, dass die fixen und dichten Attribute den statischen Anteil innerhalb eines Log-Events darstellen. Die fixen aber nicht dichten Attribute bilden den variablen Anteil.
Mithilfe des vorgestellten Clustering-Verfahrens besteht die Möglichkeit automatisiert Muster aus Log-Events zu generieren. Der Algorithmus ist in der Programmiersprache C implementiert worden. Das Programm wird Simple Logfile Clustering Tool (SLCT) genannt und ist frei verfügbar (vgl. [Vaa03]).

3 Entwurf und Modellierung

In den folgenden Abschnitten wird der Algorithmus für die automatisierte Mustererkennung entworfen und das notwendige Datenmodell erstellt. Es werden zunächst die genauen Anforderungen analysiert. Die Datenstruktur wird benötigt, um große Trainingsdaten effizient zu erstellen und die Ergebnisse des Trainings in einem Modell persistieren zu können.
Anschließend wird beschrieben, wie die Trainingsdaten sowie das aus diesen erzeugte Modell dem Algorithmus innerhalb des Hauptspeichers zur Verfügung gestellt werden.

3.1 Anforderungsanalyse

Es soll ein Algorithmus entworfen werden, der mithilfe der Methodik des überwachten Lernens Muster erzeugt. Diese Muster sollen genutzt werden, um unbekannte Log-Dateien zu analysieren und in einem für den Menschen verständlichen Bericht zusammenzufassen.
Während des überwachten Trainings soll der Algorithmus aus vorgegebenen Log-Event-Sequenzen Muster erzeugen. Wie bereits festgestellt benötigt der Algorithmus hierfür Informationen über die variablen und statischen Anteile innerhalb der Log-Event-Sequenzen. Die variablen Anteile werden durch Platzhalter ersetzt.

Sei beispielsweise die Log-Event-Sequenz aus Quelltext 3.1 gegeben.

```
1 2016-09-01 13:31:27,592 DEBUG - de.rcenvironment.components.script.common -
      BundleEvent STARTING - de.rcenvironment.components.script.common
2 2016-09-01 13:31:27,592 DEBUG - de.rcenvironment.components.script.common -
      BundleEvent STARTED - de.rcenvironment.components.script.common
```

Quelltext 3.1: Beispiel einer Log-Event-Sequenz für das Training

Der Algorithmus soll aus der Trainings-Sequenz (Quelltext 3.1) ein Muster, dargestellt in Quelltext 3.2, erzeugen[1].

```
1 DEBUG - (.*) - BundleEvent STARTING - (.*)
2 DEBUG - (.*) - BundleEvent STARTED - (.*)
```

Quelltext 3.2: Beispiel eines erzeugten Musters aus einer Trainings-Sequenz

Um dieses Muster erzeugen zu können benötigt der Algorithmus Informationen über den variablen und den statischen Anteil innerhalb der Trainings-Sequenzen. Es genügt den variablen Anteil „de.rcenvironment.components.script.common“ kenntlich zu machen. Aus erkannten Log-Event-Sequenzen soll eine komprimierte und für den Menschen verständliche Ausgabe erzeugt werden. Innerhalb dieser Ausgabe kann der variable Anteil markiert werden. Bei der späteren Mustererkennung für unbekannte Log-Event-Sequenzen wird der jeweils entsprechende variable Anteil in die Ausgabe in der markierten Stelle ersetzt.
Der grobe Ablauf soll sich folgendermaßen einteilen:

1. Der Trainer stellt Trainings-Sequenzen mit entsprechender Ausgabe dem Algorithmus zur Verfügung. Innerhalb dieser Ausgabe wird der variable Anteil der dazugehörigen Trainings-Sequenz kenntlich gemacht.

2. Der Algorithmus erzeugt ein Muster mithilfe der vom Trainer zur Verfügung gestellten Trainingsdaten.

3. Die entsprechenden Log-Event-Sequenzen werden mit Hilfe der gelernten Muster erkannt.

4. Die spezifizierte Ausgabe wird erzeugt, in dem für den Platzhalter innerhalb der Ausgabe der jeweilige variable Anteil der erkannten Sequenzen eingesetzt wird.

[1] Die Zeitstempel werden beim Erzeugen der Muster sowie beim späteren Musterabgleich mit unbekannten Log-Event-Sequenzen zugunsten der Komplexität nicht berücksichtigt.

Nachdem der Lernalgorithmus die Muster aus den Lerndaten generiert hat, sollen diese in einem Modell zwischengespeichert werden. Dieses Modell soll anschließend dazu genutzt werden, um die gelernten Muster in Log-Dateien zu erkennen und die für jedes erzeugte Muster definierte Ausgabe zu generieren. Eine besondere Herausforderung bei der Erkennung stellen Log-Event-Sequenzen dar, deren Log-Events nicht direkt aufeinander folgen.
In Quelltext 3.3 ist ein Beispiel für eine Log-Datei gegeben.

```
1 2016-09-01 13:31:27,592 DEBUG - de.rcenvironment.components.script.common -
      BundleEvent STARTING - de.rcenvironment.components.script.common
2 2016-09-01 13:31:27,592 DEBUG - de.rcenvironment.components.script.execution -
      BundleEvent [unknown:512] - de.rcenvironment.components.script.execution
3 2016-09-01 13:31:27,592 DEBUG - de.rcenvironment.components.script.common -
      BundleEvent STARTED - de.rcenvironment.components.script.common
```

Quelltext 3.3: Über mehrere Zeilen verteilte Log-Event-Sequenzen

Der zu entwerfende Algorithmus soll das Muster aus Quelltext 3.2 erkennen, auch wenn andere Log-Events zwischenzeitlich auftreten. Darüber hinaus soll der Zeitraum, d.h. der Start- und Endzeitpunk, der erkannten Log-Event-Sequenzen innerhalb der Ausgabe erhalten bleiben. Dies macht eine besondere Behandlung der Zeitstempel notwendig. Mithilfe des Algorithmus soll es damit möglich sein, gegebene Log-Dateien zu komprimieren sowie die Log-Dateien in eine Struktur zu überführen, die es erlaubt Anomalien in Log-Event-Sequenzen kenntlich zu machen.

Der Algorithmus soll entworfen und in der Programmiersprache Java implementiert werden. Es wird zunächst eine passende Datenstruktur für Trainingsdaten sowie die trainierten Muster mit der dazugehörigen Ausgabe entworfen. Anschließend wird der Algorithmus implementiert. Für die spätere Nutzung der implementierten Funktionalitäten wird eine grafische Benutzeroberfläche (graphical user interface (GUI)) entwickelt. Das Programm wird mithilfe von Log-Dateien aus RCE erprobt.

3.2 Das Datenmodell

Der Algorithmus benötigt für die Erzeugung der Muster Log-Event-Sequenzen, in denen der variable Anteil kenntlich gemacht ist. Die Markierung des variablen Anteils erfolgt innerhalb der definierten Ausgabe. Sei innerhalb einer Log-Event-Sequenz beispielsweise der Ausdruck „de.rcenvironment.components.script.common“ variabel, dann könnte dieser innerhalb der Ausgabe speziell kenntlich gemacht werden. Dies ist in Quelltext 3.4 dargestellt.

```
1 ${de.rcenvironment.components.script.common}
```

Quelltext 3.4: Beispiel eines markierten variablen Anteils innerhalb der Ausgabe

Der Ausdruck wird mit einem $ eingeleitet und befindet sich anschließend innerhalb der geschweiften Klammern. Dies erleichtert die Verarbeitung von variablen Ausdrücken mit Leerzeichen.

Wie bereits in den Grundlagen erläutert, besteht ein Log-Event aus einer Log-Zeile, die mit dem Beginn eines neuen Zeitstempels, der ein neues Log-Event einläutet, endet. Eine Log-Zeile besteht aus einem statischen und variablen Anteil. Mehrere logisch zusammengehörige Log-Events werden Log-Event-Sequenz bzw. an dieser Stelle Log-Event-Block genannt. Ein Log-Event-Block enthält folglich eine geordnete Liste von Log-Events. Die Beziehung zwischen Log-Events und Log-Event-Blocks ist in Abbildung 3.1 in Form eines Klassen-Diagramms dargestellt. Ein Log-Event wird aufgeteilt in Inhalt und Zeitstempel. Diese Trennung ist essentiell für die spätere Mustererkennung, da der Zeitstempel zwar semantisch einen variablen Anteil darstellt, aber nicht in Form eines Platzhalters in den erzeugten Mustern auftreten soll. Der dahinterstehende Grund wurde in der Anforderungsanalyse bereits kurz angeschnitten. Log-Events müssen nicht aufeinander folgen. Die zeitliche Korrelation der vermeintlich zusammengehörigen Log-Events spielt jedoch spätestens bei der Betrachtung von verteilten Log-Dateien eine große Rolle. Folglich wird bereits zu diesem Zeitpunkt eine Trennung vorgenommen, um den Algorithmus bezüglich einer genaueren Betrachtung der Zeitstempel erweiterbar zu machen. An dieser Stelle bedeutet dieser Schritt lediglich, dass die Zeitstempel für die Musterkennung zunächst nicht berücksichtigt werden.

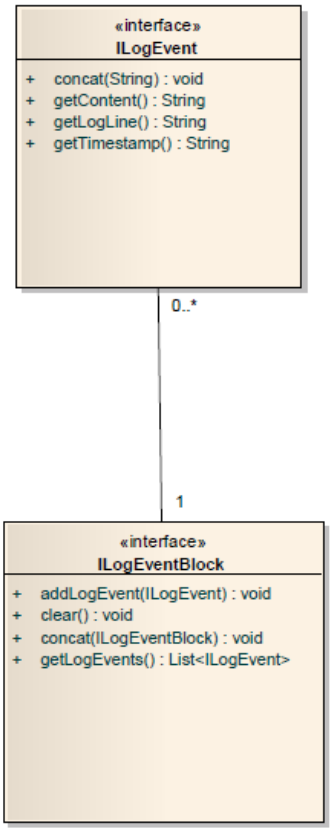

Abbildung 3.1: Beziehung zwischen Log-Event und Log-Event-Block

Für das Training müssen Log-Event-Blocks mit ihren Log-Events sowie die dazugehörige Ausgabe in einem angemessenen Format dem Algorithmus zur Verfügung gestellt werden. Im Allgemeinen handelt es sich bei den Trainingsdaten um eine Abbildung, welche einen Log-Event-Block auf einen anderen Log-Event-Block, die zu erzeugende Ausgabe, abbildet. Es handelt sich folglich um ein Key-Value-Paar. Als Datenformat wird XML (Extensible Markup Language) verwendet, da Log-Event-Blocks, die aus einer größeren Anzahl an Log-Events bestehen, sich in XML leichter darstellen lassen als in alternativen Formaten wie beispielsweise JSON (JavaScript Object Notation).

Ein exemplarischer Trainingsdatensatz ist in Quelltext 3.5 dargestellt.

```
<?xml version="1.0"?>
<trainingdata>
    <data>
        <input>
            <inputline>2016-06-29 08:44:13,119 DEBUG - de.rcenvironment.core.utils.
    common - BundleEvent STARTING - de.rcenvironment.core.utils.common</inputline>
            <inputline>2016-06-29 08:44:13,119 DEBUG - de.rcenvironment.core.utils.
    common - BundleEvent STARTED - de.rcenvironment.core.utils.common</inputline>
        </input>
        <output>
            <outputline>Bundle ${de.rcenvironment.core.utils.common} ist gestartet
    </outputline>
        </output>
    </data>
</trainingdata>
```

Quelltext 3.5: Trainingsdatensatz in XML-Format

Die Entität *inputline* stellt eine Log-Zeile bzw. ein Log-Event dar. Die Menge an Log-Events, die einen gemeinsamen Log-Event-Block bildet, wird unter der Entität *input*, die dazugehörige Ausgabe analog unter der Entität *output* zusammengefasst. Ein Datenpunkt, d.h. ein Log-Event-Block mit dazugehörigen Output, beginnt mit der Entität *data*.

Der Trainingsalgorithmus soll folglich Daten in dem oben dargestellten Format verarbeiten und aus diesen Muster erzeugen. Für die erzeugten Muster, die entsprechende Ausgabe und die dazugehörigen Metainformationen wird ein spezielles Datenformat erstellt: das *Pattern-Model*.
Das Pattern-Model ist in Abbildung 3.2 dargestellt. Die Quintessenz dieser Datenstruktur sind die *Log-Pattern*, welche aus einer geordneten Liste von *Pattern* bestehen. Ein Pattern stellt ein Muster, d.h. in der Regel einen regulären Ausdruck, dar. Dieses Muster abstrahiert jeweils ein Log-Event. Folglich bildet die Menge an erkannten Log-Events innerhalb der *Log-Pattern* einen Log-Event-Block.
Darüber hinaus benötigt der Algorithmus für die spätere Erzeugung der Ausgabe eine Zuordnung von den Platzhaltern innerhalb der Muster zu den variablen Anteilen innerhalb der Ausgaben.

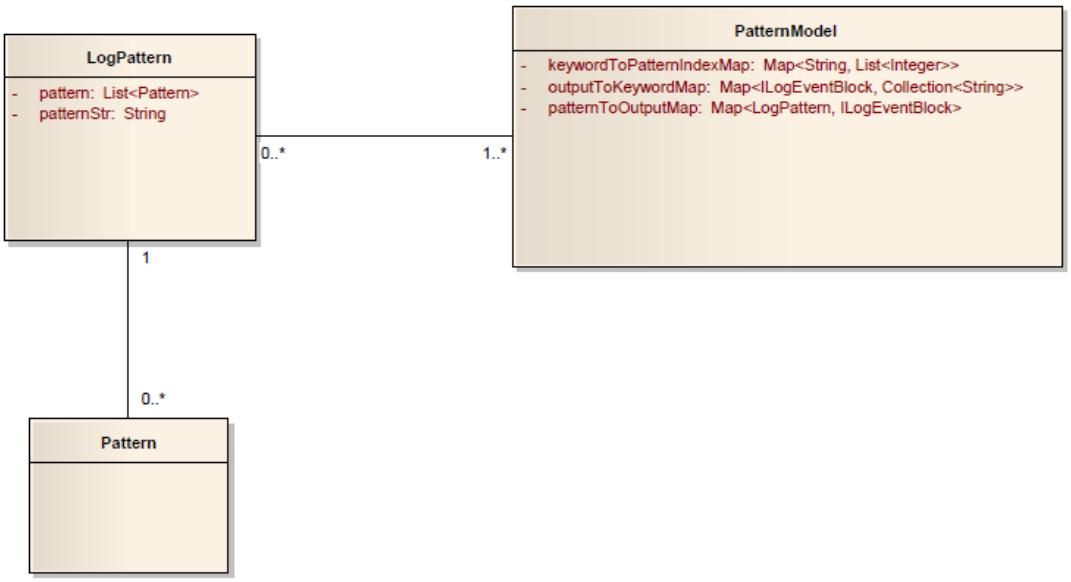

Abbildung 3.2: Das Pattern-Model

In Quelltext 3.6 ist ein Beispiel für ein Muster gegeben.

```
1 DEBUG - (.*) - BundleEvent STARTING - (.*)
2 DEBUG - (.*) - BundleEvent STARTED - (.*)
```

Quelltext 3.6: Beispiel für ein vereinfachtes Muster

In Quelltext 3.7 findet sich eine exemplarische Ausgabe.

```
1 Bundle ${de.rcenvironment.core.utils.common} startet.
```

Quelltext 3.7: Beispiel für eine vereinfachte Ausgabe

Der in der Ausgabe kenntlich gemachte variable Ausdruck *de.rcenvironment.core.utils.common* wird Schlüsselwort genannt.

Es soll beispielsweise das Log-Event aus Quelltext 3.8 erkannt werden.

```
1 DEBUG - de.rcenvironment.core.shutdown - BundleEvent STARTING - de.rcenvironment.
      core.shutdown
2 DEBUG - de.rcenvironment.core.shutdown - BundleEvent STARTED - de.rcenvironment.
      core.shutdown
```

Quelltext 3.8: Beispiel für ein zu erkennendes Log-Event

Der Algorithmus soll die Ausgabe aus Quelltext 3.9 erzeugen.

```
1 Bundle de.rcenvironment.core.shutdown ist gestarted.
```

Quelltext 3.9: Beispiel für eine erzeugte Ausgabe

Folglich benötigt der Algorithmus die Information welcher von den Platzhaltern „(.∗)“ erkannten Ausdrücke, anstelle welchen Schlüsselworts in der Ausgabe eingesetzt werden soll. Für die Lösung dieses Problems werden die Platzhalter im erzeugten Pattern-Model mit einem Index $n \in \mathbb{N}$ versehen. Diese Indizierung ermöglicht eine eindeutige Identifizierung der Platzhalter innerhalb des Musters. Die Schlüsselwörter werden anschließend den jeweiligen Positionen zugeordnet. Für das obige Beispiel wird das Schlüsselwort *de.rcenvironment.core.utils.common* den Indizes $0, 1, 2$ und 3 zugeordnet. Dieser Zusammenhang ist in Tabelle 3.1 dargestellt.

Schlüsselwort	Positionen
de.rcenvironment.core.utils.common	$0, 1, 2, 3$

Tabelle 3.1: Zuordnung von Schlüsselwort zu Indizes

Bei der Mustererkennung kann der Algorithmus anschließend eine Zuordnung der Schlüsselwörter mithilfe der Indizes zu den entsprechenden variablen Ausdrücken vornehmen. Für das Beispiel aus Quelltext 3.8 ist diese Zuordnung in Tabelle 3.2 dargestellt.

Schlüsselwort	Variabler Ausdruck
de.rcenvironment.core.utils.common	de.rcenvironment.core.shutdown

Tabelle 3.2: Zuordnung von Schlüsselwort zu variablen Ausdruck

Die Platzhalter (.∗) mit den Indizes $0, 1, 2$ und 3 können folglich den richtigen Schlüsselwörtern zugeordnet werden und der Algorithmus kann in der Ausgabe die in Quelltext 3.9 dargestellte Ersetzung vornehmen.
Die ermittelten Daten werden in dem vorgestellten Pattern-Model in einem speziellen XML-Format persistiert.

Für das obige Beispiel ist die XML-Datei in Quelltext 3.10 dargestellt.

```
<?xml version="1.0" ?>
<patternModel>
    <data>
        <pattern>
            <patternline>DEBUG - (.*) - BundleEvent STARTING - (.*)</patternline>
            <patternline>DEBUG - (.*) - BundleEvent STARTED - (.*)</patternline>
        </pattern>
        <output>
            <outputline>Bundle ${de.rcenvironment.core.utils.common} ist gestartet
    </outputline>
        </output>
        <keywords>
            <keyword key="de.rcenvironment.core.utils.common">0;1;2;3</keyword>
        </keywords>
    </data>
</patternModel>
```

Quelltext 3.10: Beispiel eines Pattern-Models im XML-Format

Der Beginn eines Datensatzes, d.h. ein Muster mit allen Metainformationen, wird mit der Entität *data* begonnen. Das Muster, das Log-Pattern, beginnt mit der Entität *pattern*. Jedes Pattern in einem Log-Pattern wird durch die Entität *patternline* gekennzeichnet. Im obigen Beispiel besteht das Log-Pattern aus zwei Pattern, d.h. die zu erkennende Log-Event-Sequenz besteht aus zwei Log-Events. Die zugeordnete Ausgabe beginnt mit der Entität *output*. Sie kann ebenfalls aus mehreren Log-Events bestehen, die durch die Entität *outputline* gekennzeichnet werden. Die Zuordnung der Schlüsselwörter erfolgt unter der Entität *keywords*. Jedes Schlüsselwort enthält in einer Entität als Schlüssel das Schlüsselwort und als Attribut eine geordnete Menge an Indizes, welche durch ein Semikolon getrennt werden. Im obigen Beispiel wäre das einzige Schlüsselwort „de.rcenvironment.core.utils.common“ mit den Indizes $0, 1, 2$ und 3.
Es besteht theoretisch die Möglichkeit ein Pattern-Model nach dem vorgestellten Schema manuell zu entwerfen, jedoch ist dies fehleranfällig. Aus diesem Grund soll das Pattern-Model in der Praxis durch den Algorithmus mithilfe der spezifizierten Lerndaten erzeugt werden.

3.3 Die Ein- und Ausgabe der Daten

Das zu implementierende Programm benötigt mehrere Module für das Einlesen und Schreiben der verschiedenen Daten. Dazu gehören Log-Dateien, Trainingsdaten sowie das Pattern-Model.
Eine Log-Datei ist eine einfache Text-Datei. Dabei wird jedes Log-Event einzeln in den Speicher geladen, um Speicherüberläufe zu verhindern, die bei sehr großen Log-Dateien auftreten könnten.
Für die bereits vorgestellten Trainingsdaten bzw. das Pattern-Model sind eigene XML-Reader, welche die Daten in den Speicher lesen, bzw. XML-Writer, welche die Daten persistent auf die Festplatte schreiben, implementiert worden.

3.4 Beispiel einer Analyse

Die Analyse von Log-Dateien wird mithilfe des gelernten Pattern-Models durchgeführt. Es wird zunächst von einer einzigen Log-Datei ausgegangen. Die Analyse bildet aus dieser einen Log-Datei einen einzigen Log-Event-Block und versucht diesen mithilfe der gelernten Muster in feinere Log-Event-Blocks zu zerlegen. Die Log-Datei wird nach den Mustern durchsucht. Jedes gefundene Muster erzeugt einen neuen Log-Event-Block. Für diesen wird am Ende der Analyse die zugehörige Ausgabe erzeugt. Der Algorithmus generiert folglich aus einem einzigen großen Log-Event-Block eine geordnete Liste an kleineren Log-Event-Blocks. Im folgenden wird der Algorithmus anhand einer exemplarischen Log-Datei mit alle seinen Facetten angewandt.

Ein Beispiel für eine Log-Datei ist in Quelltext 3.11 dargestellt.

```
1 2016-03-29 09:46:12,360 DEBUG - de.rcenvironment.core.communication.connection.impl
      .ConnectionSetupImpl - Message channel c1s-a0a53cd8c3a54e3fbed06ac2eb691398
      established for connection setup 1
2 2016-03-29 09:46:12,375 DEBUG - de.rcenvironment.core.communication.connection.impl
      .ConnectionSetupImpl - Processing event CONNECT_ATTEMPT_SUCCESSFUL (#1, Channel
       c1s-a0a53cd8c3a54e3fbed06ac2eb691398 (ESTABLISHED)) while in state CONNECTING
3 2016-03-29 09:46:12,375 INFO  - de.rcenvironment.core.communication.connection.impl
      .ConnectionSetupImpl - Network connection established: "rce-test02-win7.sc.kp.
      dlr.de:21000"
4 2016-03-29 09:46:12,375 DEBUG - de.rcenvironment.core.utils.common - BundleEvent
     STARTING - de.rcenvironment.core.utils.common
5 2016-03-29 09:46:12,375 DEBUG - de.rcenvironment.core.communication.connection.impl
      .ConnectionSetupImpl - Connection setup rce-test02-win7.sc.kp.dlr.de:21000
      changed state from CONNECTING to CONNECTED
```

Quelltext 3.11: Beispiel einer Log-Datei für die Analyse

Die Zeilen $1, 2, 3$ und 5 aus dem Quelltext 3.11 sind typisch für einen erfolgreichen Verbindungsaufbau von einer Instanz zu einer anderen Instanz in der verteilten Integrationsumgebung RCE. Aus diesen Zeilen sollen zunächst Lerndaten erzeugt werden. Mithilfe der Lerndaten soll der Algorithmus ein Pattern-Model generieren. Dieses soll genutzt werden, um die obige Datei zu analysieren. Eine zusätzliche Komplexität ergibt sich durch Zeile 4, welche dem Muster nicht zugehörig ist. Die zum Muster gehörenden Log-Events folgen nicht unmittelbar aufeinander. Der Algorithmus soll das Muster dennoch erkennen.
Eine exemplarische Ausgabe ist in Quelltext 3.12 definiert.

```
1 Connection for setup ${1} to ${rce-test02-win7.sc.kp.dlr.de:21000} on channel ${c1s
     -a0a53cd8c3a54e3fbed06ac2eb691398} successfully established
```

Quelltext 3.12: Beispiel einer Ausgabe für die Analyse

Die Ausdrücke „1“, „rce-test02-win7.sc.kp.dlr.de:21000“ und „c1s-a0a53cd8c3a54e3fbed06ac2eb691398“ sind als variabel in der obigen Ausgabe markiert.

Der notwendige Lerndatensatz ist in Quelltext 3.13 dargestellt.

```
<?xml version="1.0"?>
<trainingdata>
    <data>
        <input>
            <inputline>2016-03-29 09:46:12,360 DEBUG - de.rcenvironment.core.communication.connection.impl.ConnectionSetupImpl - Message channel c1s-a0a53cd8c3a54e3fbed06ac2eb691398 established for connection setup 1</inputline>
            <inputline>2016-03-29 09:46:12,375 DEBUG - de.rcenvironment.core.communication.connection.impl.ConnectionSetupImpl - Processing event CONNECT_ATTEMPT_SUCCESSFUL (#1, Channel c1s-a0a53cd8c3a54e3fbed06ac2eb691398 (ESTABLISHED)) while in state CONNECTING</inputline>
            <inputline>2016-03-29 09:46:12,375 INFO  - de.rcenvironment.core.communication.connection.impl.ConnectionSetupImpl - Network connection established: "rce-test02-win7.sc.kp.dlr.de:21000"</inputline>
            <inputline>2016-03-29 09:46:12,375 DEBUG - de.rcenvironment.core.communication.connection.impl.ConnectionSetupImpl - Connection setup rce-test02-win7.sc.kp.dlr.de:21000 changed state from CONNECTING to CONNECTED</inputline>
        </input>
        <output>
            <outputline>Connection for setup ${1} to ${rce-test02-win7.sc.kp.dlr.de:21000} on channel ${c1s-a0a53cd8c3a54e3fbed06ac2eb691398} successfully established</outputline>
        </output>
    </data>
</trainingdata>
```

Quelltext 3.13: Beispiel eines Lerndatensatzes für die Analyse

Der Algorithmus ersetzt die variabel markierten Ausdrücke durch den regulären Ausdruck (.∗). Das vom Algorithmus erzeugte Muster ist in Quelltext 3.14 dargestellt.

```
1 DEBUG - de.rcenvironment.core.communication.connection.impl.ConnectionSetupImpl - Message channel (.*) established for connection setup (.*)
2 DEBUG - de.rcenvironment.core.communication.connection.impl.ConnectionSetupImpl - Processing event CONNECT_ATTEMPT_SUCCESSFUL (#(.*), Channel (.*) (ESTABLISHED)) while in state CONNECTING
3 INFO  - de.rcenvironment.core.communication.connection.impl.ConnectionSetupImpl - Network connection established: "(.*)"
4 DEBUG - de.rcenvironment.core.communication.connection.impl.ConnectionSetupImpl - Connection setup (.*) changed state from CONNECTING to CONNECTED
```

Quelltext 3.14: Beispiel eines Musters für die Analyse

Das obige Muster soll während der Analyse die Zeilen $1, 2, 3$ und 5 in der obigen Log-Datei als einen Log-Event-Block erkennen und aus diesen einen neuen Log-Event-Block bilden. Aus diesem neuen Block soll anschließend die hinterlegte Ausgabe erzeugt werden. In der Ausgabe werden die Schlüsselwörter durch die jeweiligen

variablen Ausdrücke ersetzt. Sie komprimiert in der Regel den eigentlichen Inhalt der Log-Events in eine für den Menschen verständlichen Form.

Zusätzlich zu dem Muster werden die Schlüsselwörter mit den entsprechenden Indizes versehen. Die Indizes stellen die Positionen der regulären Ausdrücke (.*) innerhalb des erzeugten Musters dar. Das Pattern-Model ist in Quelltext 3.15 dargestellt.

```
<?xml version="1.0" ?>
<patternModel>
    <data>
        <pattern>
            <patternline>DEBUG - de.rcenvironment.core.communication.connection.impl.ConnectionSetupImpl - Message channel (.*) established for connection setup (.*)</patternline>
            <patternline>DEBUG - de.rcenvironment.core.communication.connection.impl.ConnectionSetupImpl - Processing event CONNECT_ATTEMPT_SUCCESSFUL (#(.*), Channel (.*) (ESTABLISHED)) while in state CONNECTING</patternline>
            <patternline>INFO  - de.rcenvironment.core.communication.connection.impl.ConnectionSetupImpl - Network connection established: "(.*)"</patternline>
            <patternline>DEBUG - de.rcenvironment.core.communication.connection.impl.ConnectionSetupImpl - Connection setup (.*) changed state from CONNECTING to CONNECTED</patternline>
        </pattern>
        <output>
            <outputline>Connection for setup ${1} to ${rce-test02-win7.sc.kp.dlr.de:21000} on channel ${c1s-a0a53cd8c3a54e3fbed06ac2eb691398} successfully established</outputline>
        </output>
        <keywords>
            <keyword key="c1s-a0a53cd8c3a54e3fbed06ac2eb691398">0;3</keyword>
            <keyword key="1">1;2</keyword>
            <keyword key="rce-test02-win7.sc.kp.dlr.de:21000">4;5</keyword>
        </keywords>
    </data>
</patternModel>
```

Quelltext 3.15: Beispiel eines Pattern-Models für die Analyse

Die definierte Ausgabe fasst vier Log-Events eines Log-Event-Blocks in einem einzigen Log-Event zusammen. In diesem Fall müssen die Zeitstempel, die Rückschluss auf die Zeitdauer des Log-Event-Blocks erlauben, ebenfalls in einer Zeile komprimiert werden, um diese Information nicht zu verlieren. Der Algorithmus soll den Startzeitstempel, d.h. der Zeitstempel des ersten auftretenden Log-Events innerhalb des Log-Event-Blocks, und Endzeitstempel, d.h. der Zeitstempel des letzten auftretenden Log-Events innerhalb des Log-Event-Blocks, in der Form *Startzeitstempel - Endzeitstempel* in

einer Zeile komprimieren. Darüber hinaus werden die Schlüsselwörter durch die von den Mustern erkannten variablen Ausdrücke in der Ausgabe ersetzt.
Die erzeugte Ausgabe für das obige Beispiel ist in Quelltext 3.16 dargestellt.

```
1 2016-03-29 09:46:12,360 - 2016-03-29 09:46:12,375 Connection for setup 1 to rce-
      test02-win7.sc.kp.dlr.de:21000 on channel c1s-a0a53cd8c3a54e3fbed06ac2eb691398
      successfully established
```

Quelltext 3.16: Beispiel für eine komprimierte Ausgabe mit komprimierten Zeitstempeln

Es wurden vier zusammengehörige Log-Events in einem neuen Log-Event mit ihrer Zeitdauer komprimiert. Wichtig ist dabei, dass die Komprimierung von der in den Lerndaten definierten Ausgabe abhängt. Folglich ist der „Lehrer“ hierfür zuständig. Neben der Komprimierung soll der Algorithmus die Daten in eine Struktur überführen, die es ermöglicht Anomalien in Log-Dateien zu erkennen. Neben der Vorstellung des Algorithmus werden diese Möglichkeiten im folgenden genauer diskutiert.

4 Automatisierte Mustererkennung in Log-Dateien

In den folgenden Abschnitten wird der Algorithmus entworfen und näher diskutiert. Der Algorithmus teilt sich in zwei Bereiche auf:

- Das **Training**, welches ein überwachtes Lernverfahren implementieren soll. Der Lehrer soll, dem bereits entwickelten Datenmodell entsprechend, Lerndaten erzeugen, welche dem Trainings-Algorithmus als Input zur Verfügung gestellt werden können. Der Algorithmus soll daraufhin das Pattern-Model erzeugen.
- Die **Analyse**, welche die im Training gelernten Muster in Log-Dateien mithilfe des gelernten Pattern-Models identifizieren soll. Dabei sollen die Log-Events nach den gelernten Mustern durchsucht und die dazugehörige Ausgabe erzeugt werden. Die Ergebnisse werden anschließend in einem komprimierten und für den Menschen leicht erfassbaren Bericht persisitiert.

Der Algorithmus wird in Form von Pseudocode vorgestellt. Das Programm wurde in der Programmiersprache Java implementiert.

4.1 Das Training

Das Training teilt sich in drei Teilbereiche auf. Dies ermöglicht eine logische Trennung der notwendigen Arbeitsschritte und erleichtert Wartung, Pflege sowie eventuelle Fehlersuche innerhalb der Codebasis.

Die Bereiche sind:

1. Das **Preprocessing** erhält als Inputparameter zwei Log-Event-Blocks. Einer der Blocks stellt dabei den Block dar, der von dem zu erzeugenden Muster erkannt werden soll. Der andere Block repräsentiert die dazugehörige Ausgabe mit markierten variablen Anteil. Das Format muss dabei dem bereits vorgestellten Datenmodell entsprechen. Es werden die Schlüsselwörter, d.h. die variablen Ausdrücke, aus dem Ausgabe-Event-Block extrahiert und zwischengespeichert.

2. Das **Training** erzeugt, mithilfe der im Preprocessing extrahierten Schlüsselwörter, das Muster und speichert dieses mit den notwendigen Metainformationen in dem Pattern-Model. Zu den Metainformationen gehört insbesondere die Zuordnung von Platzhalterposition zu jeweils einem Schlüsselwort, ohne welche eine spätere Ausgabenerzeugung nicht möglich ist.

3. Das **Postprocessing** gibt nicht mehr benötigte Ressourcen für die nächste Iteration frei.

Im **Preprocessing** werden die Schlüsselwörter mithilfe des regulären Ausdrucks $\${(.*?)}$ extrahiert. Der reguläre Ausdruck erkennt Zeichenketten, die mit einem $ beginnen und sich zwischen geschweiften Klammern befinden. .*? erkennt dabei null oder mehr Zeichen. Die runden Klammern stellen spezielle Metazeichen dar. Ausdrücke, die zwischen diesen erkannt werden, bilden eine sogenannte Gruppe. Dies erleichtert eine spätere Unterteilung der gefundenen Schlüsselwörter.

Das Extrahieren ist in Algorithmus 1 dargestellt.

```
   Data : ILogEventBlock b
   Result : List<String> result
1  for all event ∈ b do
2  |  keyword := search ${(.*?)} in event;
3  |  if keyword ≠ null then
4  |  |  put keyword in result;
5  |  end
6  end
```

Algorithmus 1 : Das Extrahieren der Schlüsselwörter

Die Methode erhält als Inputparameter einen Log-Event-Block, die vom Nutzer spezifizierte Ausgabe mit den markierten variablen Anteilen, bestehend aus n Log-Events und erzeugt als Ausgabe eine Liste an Schlüsselwörtern. Mithilfe des vorgestellten regulären Ausdrucks wird jedes Log-Event nach markierten Schlüsselwörtern durchsucht und in einer Liste gespeichert und steht damit für das Training zur Verfügung.

Die Mustergenerierung

Während des **Trainings** soll aus einem übergebenen Log-Event-Block ein Muster erzeugt werden. Hierfür nutzt der Algorithmus die im Preprocessing extrahierten Schlüsselwörter. Diese werden für die Mustergenerierung durch den regulären Ausdruck $(.*)$ ersetzt. Jedes ersetzte Schlüsselwort wird mit einem Index $i \in \mathbb{N}$ versehen und dem Pattern-Model hinzugefügt. Nach der Mustergenerierung wird das Pattern-Model in einer XML-Datei persistiert.

Die Mustergenerierung ist in Algorithmus 2 dargestellt.

```
   Data : ILogEventBlock input
   Result : PatternModel model
 1 position := 0;
 2 model := PatternModel;
 3 pattern := null;
 4 for all event ∈ input do
 5 |   Collection<Token> tokens := tokenize event;
 6 |   for all token ∈ tokens do
 7 |   |   if token matches keyword then
 8 |   |   |   token := (.*);
 9 |   |   |   add token with position to model;
10 |   |   |   increment position;
11 |   |   end
12 |   |   pattern := pattern + token;
13 |   end
14 |   add pattern to LogPattern;
15 end
16 add LogPattern to model;
```

Algorithmus 2 : Die Mustergenerierung

Als Eingabeparameter erhält der Algorithmus einen Log-Event-Block. Aus diesen soll das Muster erzeugt werden. Als Ausgabe wird das Pattern-Model erzeugt, welches nach Abschluss des Trainings in der genannten XML-Datei persistiert wird. In den Zeilen $1 - 3$ werden die notwendigen Ressourcen vor der ersten Iteration initialisiert. Dazu gehört der Index (*position*) der Platzhalter, welcher mit 0 initialisiert wird, das Pattern-Model sowie das zu erzeugende Pattern je Log-Event innerhalb des Log-Event-Blocks. Es wird anschließend über alle Log-Events iteriert (Zeile 4). Jedes Log-Event wird mithilfe der im Preprocessing extrahierten Schlüsselwörter in einzelne Token zerlegt (Zeile 5).

```
1 DEBUG - de.rcenvironment.core.shutdown - BundleEvent STARTING - de.rcenvironment.
      core.shutdown
```

Quelltext 4.1: Beispiel für ein Log-Event für das Extrahieren der Schlüsselwörter

Das Schlüsselwort aus Quelltext 4.1 *de.rcenvironment.core.shutdown* würde zu den Tokens {*DEBUG* -}, {- *BundleEvent STARTING* -} und {*de.rcenvironment.core.shutdown*} führen.
Anschließend wird über die erzeugten Tokens iteriert (Zeile 6) und geprüft, ob es sich bei diesen um ein Schlüsselwörter handelt (Zeile 7). Ist dies der Fall wird das Schlüsselwort durch den regulären Ausdruck $(.*)$ ersetzt und mit dem entsprechenden Index dem Pattern-Model hinzugefügt (Zeilen $8-9$). Anschließend wird der Index für nachfolgende Iterationen inkrementiert (Zeile 10). Je Log-Event des Log-Event-Blocks wird ein Pattern erzeugt und in dem entsprechenden Log-Pattern gespeichert (Zeile 14). Abschließend wird das Log-Pattern dem Pattern-Model hinzugefügt (Zeile 16).

4.2 Die Mustererkennung

Die Mustererkennung erhält als Inputparameter eine Log-Datei. Diese bildet einen einzigen Log-Event-Block. Der Algorithmus sucht in diesem Log-Event-Block nach den gelernten Mustern und erzeugt für gefundene Muster einen neuen Log-Event-Block. Es entsteht eine geordnete Liste an Log-Event-Blocks.

Die Mustererkennung ist in Algorithmus 3 dargestellt.

```
Data : Log ∈ ILogEventBlock
Result : List<ILogEventBlock>
1  for all event ∈ Log do
2      isMatch := false;
3      for all Log-Pattern ∈ PatternModel do
4          for all pattern ∈ Log-Pattern do
5              if event matches pattern then
6                  isMatch := true;
7                  hasScope := false;
8                  for all scope ∈ Pattern-Scopes do
9                      if event is in scope then
10                         hasScope := true;
11                         put event in scope;
12                         break;
13                     end
14                 end
15                 if hasScope ≠ true then
16                     open new scope s;
17                     put event in s;
18                     break;
19                 end
20                 if isMatch = true then
21                     break;
22                 end
23             end
24         end
25         if isMatch = true then
26             break;
27         end
28     end
29 end
```

Algorithmus 3 : Die Mustererkennung in einer Log-Datei

Der Algorithmus iteriert über die Menge der Log-Events des übergebenen Log-Event-Blocks (Zeile 1). In Zeile 2 wird eine boolesche Variable *isMatch* mit *false* initialisiert. Diese wird an späterer Stelle dazu genutzt, um kenntlich zu machen, dass ein Muster erkannt worden ist. Für jedes Log-Event wird geprüft, ob dieses von einem gelernten Log-Pattern erkannt wird. Die gelernten Log-Pattern befinden sich in dem Pattern-Model, welches während des Trainings erzeugt worden ist. Ein Log-Pattern besteht aus einer geordneten Liste von Pattern. Jedes Pattern repräsentiert ein Log-Event. Für die Erkennung eines Log-Events muss folglich jedes Pattern innerhalb jedes Log-Patterns geprüft werden. Dies wird durch zwei ineinander verschachtelte For-Schleifen umgesetzt. In der äußeren Schleife (Zeile 3) wird über alle Log-Pattern des Pattern-Models iteriert. Anschließend werden alle Pattern innerhalb dieser Log-Pattern auf einen Treffer geprüft. Dies übernimmt die innere Schleife in Zeile 4. Das Pattern stellt einen regulären Ausdruck dar, in welchem der variable Teil des Log-Events durch den Ausdruck $(.*)$ ersetzt worden ist. $(.*)$ erkennt beliebige Zeichen mit beliebiger Wiederholung.
Die regulären Ausdrücke sollten im Pattern-Model aus Gründen der Performanz bereits kompiliert vorliegen.
Wird ein Log-Event von einem Pattern erkannt, wird der Variablen *isMatch* der Wert *true* zugewiesen (Zeile 6). Anschließend wird eine weitere boolesche Variable *hasScope* mit *false* initialisiert.
In der Anforderungsanalyse ist erläutert worden, dass das Erkennen von Log-Events, die nicht unmittelbar aufeinander folgen, ein Problem darstellt. Dieses Problem ist an dieser Stelle erkennbar, da jedes Log-Event zunächst einzeln betrachtet werden muss bzw. einzeln von einem Pattern innerhalb eines Log-Pattern erkannt wird. Das zu erkennende Muster setzt sich jedoch aus beliebig vielen Pattern zusammen, welche nicht direkt aufeinander folgen müssen. Folglich kann zwar ein Log-Event von einem Pattern erkannt werden, jedoch ist das dazugehörige Log-Pattern unvollständig.
Der naive Ansatz für die Lösung dieses Problems wäre es für jedes erkannte Log-Event erneut alle Log-Events innerhalb der Log-Datei zu durchsuchen und alle dazugehörigen Log-Events auf das entsprechende Muster, d.h. Log-Pattern, abzugleichen. Dies würde die Komplexität jedoch deutlich steigern. Eine bessere Lösung wäre es jedes Log-Event einer Art Klassifizierung zu unterziehen, d.h. eine Zuordnung zu dem passenden Log-Pattern innerhalb des Pattern-Models vorzunehmen. Wenn ein Log-Event folglich von einem Pattern innerhalb eines Log-Patterns erkannt wird,

muss dieses Log-Event diesem Log-Pattern zugeordnet werden. Darüber hinaus muss das Pattern innerhalb des Log-Patterns als „besetzt“ kenntlich gemacht werden, d.h. der Algorithmus muss wissen, dass für dieses Pattern bereits ein erkanntes Log-Event existiert. Vor der Erzeugung der Ausgabe, die den Log-Pattern zugeordnet sind, muss geprüft werden, ob alle dazugehörigen Pattern ein passendes Log-Event erkannt haben. Für diesen Zweck wird ein Datentyp eingeführt: der *Pattern-Scope*. Es wird im Hintergrund zunächst eine geordnete Liste von Pattern-Scopes angelegt. Diese Liste ist zum Initialisierungszeitpunkt leer. Für jedes von einem Pattern erkannte Log-Event wird ein neuer Pattern-Scope erzeugt. Diesem Scope wird anschließend das erkannte Log-Event hinzugefügt. Darüber hinaus wird eine Referenz auf das Log-Pattern des Patterns, welches das Log-Event erkannt hat, ebenfalls dem Scope hinzugefügt. Zusätzlich wird die Information hinterlegt, welches Pattern von dem hinzugefügten Log-Event besetzt wird[1].

Für das Prinzip der Pattern-Scopes müssen für jedes von einem Pattern erkannte Log-Event alle bereits existierenden Scopes auf Übereinstimmung geprüft werden, d.h. es muss geprüft werden, ob ein erkanntes Log-Event zu einem bereits bestehenden Scope gehört. Hierfür wird über die geordnete Liste an Scopes iteriert (Zeile 8). Ist dieses Event in einem der Scopes wird der Variablen *hasScope* der Wert *true* zugeordnet (Zeile 10). Das Log-Event wird dem Scope hinzugefügt (Zeile 11) und die Schleife kann verlassen werden (Zeile 12). Wird kein passender Scope gefunden, wird ein neuer Scope geöffnet (Zeile 16) und das Log-Event wird diesem hinzugefügt (Zeile 17). Abschließend wird vor dem Ende jeder Iteration mithilfe des booleschen Ausdrucks *isMatch* geprüft, ob für dieses Log-Event innerhalb der bestehenden Iteration ein Treffer erzielt worden ist (Zeile 20). Ist dies der Fall kann die Schleife verlassen werden.
An dieser Stelle sei anzumerken, dass es Log-Events geben kann, die von mehreren unterschiedlichen Log-Pattern erkannt werden könnten. In seiner jetzigen Form akzeptiert der Algorithmus den ersten gefundenen Treffer. Die anderen möglicherweise existierenden Pattern werden nicht weiter berücksichtigt. Bei ersten Tests anhand von Log-Dateien aus RCE ist in der Praxis dieses Problem nicht aufgetreten.

[1] Eine genaue Beschreibung der Pattern-Scopes erfolgt im Kapitel *Die Pattern-Scopes*.

4.2.1 Die Pattern-Scopes

Die Notwendigkeit der Pattern-Scopes ist durch die Verteilung der Log-Events der Log-Event-Sequenzen innerhalb der Log-Datei gegeben. Ein Pattern-Scope enthält eine geordnete Liste an Log-Events sowie ein Log-Pattern. Die Liste ist zum Initialisierungszeitpunkt leer und wird mit den Log-Events, die von den Pattern des Log-Patterns erkannt werden, gefüllt. Ein Pattern-Scope kann zwei verschiedene Zustände annehmen:

1. Ein Pattern-Scope ist **geöffnet**, wenn ein Log-Event von einem Pattern eines Log-Patterns erkannt wird. Für jedes erkannte Log-Event wird folglich ein neuer Pattern-Scope geöffnet, es sei denn das Log-Event wird als Teil eines bereits geöffneten Pattern-Scopes erkannt. In diesem Fall wird das erkannte Log-Event der geordneten Liste des bereits geöffneten Pattern-Scopes hinzugefügt.
2. Ein Pattern-Scope ist **geschlossen**, wenn die Anzahl der Log-Events innerhalb der geordneten Liste mit der Anzahl an Patterns des dazugehörigen Log-Patterns übereinstimmt. Alle Patterns gelten als besetzt.

Ein Beispiel für eine Log-Event-Sequenz ist in Quelltext 4.2 dargestellt.

```
1 2016-09-01 13:31:27,592 DEBUG - de.rcenvironment.components.script.common -
      BundleEvent STARTING - de.rcenvironment.components.script.common
2 2016-09-01 13:31:27,592 DEBUG - de.rcenvironment.core.shutdown - BundleEvent
      STARTED - de.rcenvironment.core.shutdown
3 2016-09-01 13:31:27,592 DEBUG - de.rcenvironment.components.script.common -
      BundleEvent STARTED - de.rcenvironment.components.script.common
```

Quelltext 4.2: Beispiel einer Log-Event-Sequenz für das Eröffnen eines neuen Pattern-Scopes

Die zu erkennende Log-Event-Sequenz setzt sich aus den Log-Events aus den Zeilen 1 und 3 zusammen. Das Log-Event in Zeile 2 ist nahezu identisch mit dem Log-Event in Zeile 3. Sie unterscheiden sich durch den variablen Ausdruck. Die algorithmische Problematik ist dadurch gegeben, dass der Algorithmus erkennen muss, dass das Log-Event aus Zeile 1 mit dem Event aus Zeile 3 und nicht mit dem Event aus Zeile 2 eine Log-Event-Sequenz bildet.
Das Muster, welches die obige Log-Event-Sequenz erkennen würde, kann diese Problematik alleine nicht lösen, da das Log-Event aus Zeile 2 zuerst auftritt. Das

Konzept hinter den Pattern-Scopes löst dieses Problem. Der Ablauf für das obige Beispiel wäre dabei folgendermaßen:

1. Das Log-Event in Zeile 1 wird von dem ersten Pattern des Log-Patterns, das die gesamte Log-Event-Sequenz erkennen soll, erkannt. Es wird ein neuer Pattern-Scope geöffnet. Diesem Pattern-Scope wird das entsprechende Log-Pattern zugeordnet und das Log-Event wird in die geordnete Liste aufgenommen. Das erste Pattern des Log-Patterns gilt als besetzt.

2. Das Log-Event in Zeile 2 wird von dem zweiten Pattern des Log-Patterns erkannt. Es wird geprüft, ob das Log-Event dem bereits geöffneten Pattern-Scope zugehörig ist. Hierfür wird überprüft, ob im bereits besetzten Pattern variable Ausdrücke vorhanden sind, welche ebenfalls in dem neu erkannten Log-Event vorhanden sein sollten. Dies geschieht mithilfe der Zuordnung der Schlüsselwörter zu den Positionen. Im obigen Beispiel handelt es sich auf den Positionen $0-3$ um den gleichen variablen Ausdruck. Dieser ist im ersten Schritt bereits mit dem Ausdruck $a =$*de.rcenvironment.components.script.common* besetzt worden. Der Ausdruck a stimmt nicht mit *de.rcenvironment.core.shutdown* überein. Folglich befindet sich das Log-Event aus Zeile 2 nicht im selben Pattern-Scope wie das Event aus Zeile 1.

3. Für das Log-Event aus Zeile 3 wird analog verfahren. Die variablen Ausdrücke stimmen dieses mal überein. Das Log-Event wird der geordneten Liste des Pattern-Scopes hinzugefügt.

4. Der Pattern-Scope wird geschlossen, da alle Pattern des Log-Patterns innerhalb des Pattern-Scopes als besetzt gelten.

Die Überprüfung, ob ein Log-Event zu einem geöffneten Pattern-Scope gehört, stellt eine besondere Herausforderung dar.

Für die Lösung nutzt die Implementierung vier verschiedene Hashtabellen:

- Die Hashtabelle *keywordPosition* ordnet den Schlüsselwörtern ihre Positionen innerhalb des Log-Patterns zu.
- Die Hashtabelle *patternPosition* ordnet den Pattern des Log-Patterns, welches dem Pattern-Scope zugeordnet ist, eine Menge an Positionen zu. Es handelt sich um die Positionen der Platzhalter $(.*)$.
- Die Hashtabelle *positionVariable* ordnet die Positionen den erkannten variablen Ausdrücken zu.
- Die Hashtabelle *keywordReplacement* ordnet den Schlüsselwörtern die erkannten variablen Ausdrücke zu.

Das Muster, welches die oben dargestellte Log-Event-Sequenz erkennt, enthält vier variable Ausdrücke. Die Lerndaten, aus denen dieses Muster generiert wurde, könnte beispielsweise den variablen Ausdruck *de.rcenvironment.components.cpacs.writer.gui* enthalten haben. Dieser wurde in den Lerndaten als Schlüsselwort markiert. Die entstehende Hashtabelle *keywordPosition* ist in Tabelle 4.1 dargestellt.

Schlüsselwort	Positionen
de.rcenvironment.components.cpacs.writer.gui	$0, 1, 2, 3$

Tabelle 4.1: Beispiel einer Hashtabelle für die Zuordnung von Schlüsselwort zu Positionen

Die Hashtabelle *patternPosition* ist in Tabelle 4.2 dargestellt.

Pattern	Positionen
DEBUG - (.*) - BundleEvent STARTING - (.*)	$0, 1$
DEBUG - (.*) - BundleEvent STARTED - (.*)	$2, 3$

Tabelle 4.2: Beispiel einer Hashtabelle für die Zuordnung von Pattern zu Positionen

Das Log-Event in Zeile 1 des Quelltexts 4.2 wird von dem ersten Pattern des Log-Patterns erkannt. Es wird ein neuer Pattern-Scope geöffnet. Diesem Scope wird das entsprechende Log-Pattern zugeordnet. Mithilfe des Pattern-Models wird die in Tabelle 4.1 dargestellte Hashtabelle *keywordPosition* erzeugt. Darüber hinaus wird

die Hashtabelle *positionVariable* aus dem ersten Log-Event generiert (siehe Tabelle 4.3).

Position	Variabler Ausdruck
0	de.rcenvironment.components.script.common
1	de.rcenvironment.components.script.common

Tabelle 4.3: Beispiel einer Hashtabelle für die Zuordnung von Indizes zu variablen Ausdrücken

Folglich wird beim Eröffnen eines Pattern-Scopes die Hashtabelle *keywordReplacement* initialisiert und gefüllt. Der sich ergebende Inhalt für *keywordReplacement* ist in Tabelle 4.4 dargestellt.

Schlüsselwort	Variabler Ausdruck
de.rcenvironment.components.cpacs.writer.gui	de.rcenvironment.components.script .common

Tabelle 4.4: Beispiel einer Hashtabelle für die Zuordnung von Schlüsselwort zu erkannten variablen Ausdruck

Mithilfe der Hashtabelle *keywordPosition*, *positionVariable*,*patternPosition* und *keywordReplacement* ergibt sich die Implementierung aus Algorithmus 4 für die Überprüfung, ob ein Log-Event Teil eines geöffneten Pattern-Scopes ist.

```
   Data : e ∈ Log-Event, Pattern p which matches e,Log-Pattern L_p ∈
          Pattern-Scope s, keywordPosition, positionVariable,
          keywordReplacement, patternPosition
   Result : true, if e is part of s. false otherwise
 1 if s is closed then
 2 |  return false;
 3 end
 4 for all pattern p_i ∈ L_p do
 5 |  if p = p_i then
 6 |  |  if patternPosition contains p_i then
 7 |  |  |  patternPositions := patternPosition.get(p_i);
 8 |  |  |  Map<Keyword, Replacement> temp := create new empty hashmap;
 9 |  |  |  for all entries t ∈ keywordPosition do
10 |  |  |  |  for all positions pos ∈ t do
11 |  |  |  |  |  if keywordReplacement contains keyword ∈ t then
12 |  |  |  |  |  |  replacement := keywordReplacement.get( keyword ∈ t);
13 |  |  |  |  |  |  if replacement ≠ positionVariable.get(pos) then
14 |  |  |  |  |  |  |  return false;
15 |  |  |  |  |  |  end
16 |  |  |  |  |  end
17 |  |  |  |  |  else
18 |  |  |  |  |  |  map keyword ∈ t to positionVariable.get(pos) in temp;
19 |  |  |  |  |  end
20 |  |  |  |  end
21 |  |  |  end
22 |  |  end
23 |  |  else
24 |  |  |  return false;
25 |  |  end
26 |  end
27 |  if temp is not empty then
28 |  |  concat temp with keywordReplacement;
29 |  end
30 |  return true;
31 end
```

Algorithmus 4 : Überprüfung auf Zugehörigkeit zu einem Pattern-Scope

Der Algorithmus 4 definiert als Rückgabewert einen booleschen Ausdruck. Dieser ist *false*, falls das übergeben Log-Event e nicht Teil des Pattern-Scopes s ist. Es wird mit der Überprüfung begonnen, ob s geschlossen ist (Zeile 1). Ist dies der Fall wird *false* zurückgegeben (Zeile 2), da folglich alle Pattern des Log-Pattern L_p besetzt sind. Anschließend wird geprüft, welches Pattern $p_i \in L_p$ für die Erkennung des Log-Events e zuständig war (Zeile $4-5$). Mithilfe der Hashtabelle *patternPosition* werden die Positionen des Patterns p_i in der Liste *patternPositions* zwischengespeichert (Zeile 7). Vorher wird geprüft, ob der Schlüssel p_i in der Hashtabelle *patternPosition* vorhanden ist (Zeile 6). Ist dies nicht der Fall, kann direkt *false* zurückgegeben werden (Zeile 24), da das Pattern nicht in dem Log-Pattern vorhanden ist[2]. In Zeile 8 wird eine neue Hashtabelle *temp* erzeugt. Diese wird später dazu genutzt eine Zuordnung zwischen Schlüsselwörtern und variablen Ausdrücken herzustellen, falls eine solche Zuordnung nicht bereits existiert.
Anschließend wird über die Einträge t aus *keywordPosition* iteriert (Zeile 9). Jeder Schlüssel aus t repräsentiert ein Schlüsselwort und jeder Wert aus t repräsentiert eine Liste an Positionen, welche die Indizes dieser Schlüsselwörter repräsentiert. Es wird über alle Positionen aus *keywordPosition* iteriert (Zeile 10). Anschließend wird geprüft, ob für die Schlüsselwörter aus *keywordPosition* bereits variable Ausdrücke hinterlegt sind. Dies wird mithilfe der Hashtabelle *keywordReplacement* umgesetzt (Zeile 11). Ist dies für keines der Schlüsselwörter der Fall, kann das gefundene Log-Event dem Pattern-Scope s hinzugefügt werden. Es werden die Schlüsselwörter aus t in der Hashtabelle *temp* den entsprechenden variablen Ausdrücken zugeordnet. Diese sind in $positionVariable$ hinterlegt (Zeile 18). Abschließend wird der Inhalt von *temp* mit *keywordReplacement* für nachfolgende Iterationen vereinigt (Zeile 28). Es kann *true* zurückgegeben werden.
Enthält hingegen *keywordReplacement* eines der Schlüsselwörter aus t, wird überprüft, ob der variable Ausdruck des Log-Events e an dieser Position mit dem zu diesem Schlüsselwort hinterlegten variablen Ausdruck übereinstimmt. Die variablen Ausdrücke von e sind in $positionVariable$ hinterlegt. Mithilfe der Positionen *pos* können diese abgefragt werden (Zeile 12). Die den Schlüsselwörtern bereits zugeordneten variablen Ausdrücke sind in *keywordReplacement* hinterlegt. In Zeile 13 erfolgt die Überprüfung auf Übereinstimmung. Stimmen die variablen Ausdrücke

[2]Dieser Fall sollte in der Praxis nicht auftreten. Die Überprüfung stellt eine zusätzliche Absicherung dar.

nicht überein wird *false* zurückgegeben. Stimmen alle variablen Ausdrücke überein, wird *true* zurückgegeben (Zeile 30).

Dem obigen Beispiel entsprechend würde Zeile 2 des Quelltexts 4.2 nicht dem geöffneten Pattern-Scope zugeordnet werden. Die entstehende Hashtabelle $positionVariable$ ist in Tabelle 4.5 dargestellt.

Position	Variabler Ausdruck
2	de.rcenvironment.core.shutdown
3	de.rcenvironment.core.shutdown

Tabelle 4.5: Beispiel einer Hashtabelle für das Erkennen von nicht zugehörigen Log-Events

Der geöffnete Pattern-Scope hat in der Hashtabelle $keywordReplacement$ dem Schlüsselwort
de.rcenvironment.components.cpacs.writer.gui den variablen Ausdruck
de.rcenvironment.components.script.common zugeordnet. In der Hashtabelle $keywordPosition$ sind die entsprechenden Positionen dieser Zuordnung hinterlegt. Die Positionen 2 und 3 müssen folglich mit dem variablen Ausdruck *de.rcenvironment.components.script.common* besetzt werden. Folglich wird anhand der Hashtabelle 4.5 erkannt, dass Zeile 2 des Quelltexts 4.2 nicht Teil des geöffneten Pattern-Scopes ist.

4.2.2 Die Analyse

Die Analyse erfolgt nach der Erzeugung der Pattern-Scopes. Mit Analyse ist die Erzeugung der Ausgaben gemeint, welche für jedes Log-Pattern hinterlegt sind. Im Allgemeinen handelt es sich bei diesem Schritt um eine mögliche Komprimierung der Log-Event-Blocks. Die Analyse erzeugt als Ausgabe folglich die Zusammenfassung einer Log-Datei. Mögliche weitere Analyseschritte werden nach der Vorstellung des für die Analyse zuständigen Algorithmus näher diskutiert.

Die Analyse der Pattern-Scopes ist in Algorithmus 5 dargestellt.

```
   Data : Log ∈ ILogEventBlock
 1 for all event ∈ Log do
 2     if event has scope then
 3         if scope is closed then
 4             write output created by scope;
 5         end
 6         else
 7             write events ∈ scope;
 8         end
 9     end
10     else
11         write event;
12     end
13 end
```

Algorithmus 5 : Die Analyse der Pattern-Scopes

Es wird erneut über alle Log-Events innerhalb der Log-Datei iteriert (Zeile 1)[3]. Anschließend wird für jedes Log-Event geprüft, ob dieses einem Pattern-Scope zugeordnet worden ist (Zeile 2). Ist dies der Fall, wird geprüft, ob der Pattern-Scope geschlossen wurde, d.h. ob alle Pattern innerhalb des Scopes besetzt wurden (Zeile 3). Ist der Scope geschlossen, wird die Ausgabe erzeugt (Zeile 4), die dem zugehörigen Log-Pattern im Pattern-Model zugeordnet worden ist. An dieser Stelle findet die Komprimierung der Log-Events statt, wobei diese maßgeblich durch die vom Lehrer spezifizierte Ausgabe gegeben ist.
Ist der Scope nicht geschlossen, d.h. es wurden nicht alle Pattern des dazugehörigen Log-Pattern besetzt, werden die bis zu diesem Zeitpunkt aufgenommenen Log-Events in der korrekten zeitlichen Reihenfolge unverändert persistiert (Zeile 7). Befindet sich ein Log-Event in keinem Scope, d.h. es wurde von keinem der gelernten Muster erkannt, wird dieses ebenfalls unverändert persistiert (Zeile 11).

[3]Dabei handelt es sich um die gleiche Datei, die der Mustererkennung unterzogen worden ist.

4.2.3 Weitere Analysemöglichkeiten

Während der Durchsuchung der erzeugten Pattern-Scopes könnten mehrere weitere Analysemöglichkeiten eingebunden werden. Beispielsweise könnte bereits ein nicht geschlossener Pattern-Scope auf eine mögliche Anomalie hindeuten, da ein logischer Log-Event-Block nicht vollständig ist. Dies ist jedoch abhängig von der Spezifizierung der Trainingsdaten und eine mögliche Ausnutzung für die Fehlererkennung ist somit abhängig von der Person, die diese Trainingsdaten erzeugt hat.
Darüber hinaus könnten an dieser Stelle Methoden der *Anomaly Detection* bzw. *Fault Detection* angewandt werden. Geht man von vollständigen Lerndaten aus, d.h. man geht davon aus, dass jedes Log-Event von einem Muster erkannt wird, dann stellen alle nicht erkannten Muster in dieser Analyse eine Anomalie dar. Eine Erweiterung, die einen Pattern-Scope zusätzlich mit einem Kontext ausstattet, ist in der jetzigen Implementierung möglich. Ein solcher Kontext könnte beispielsweise durch einen bestimmten maximalen Abstand der Zeitstempel zwischen den im Scope enthaltenen Pattern dargestellt werden.
Eine *Fault Detection* lässt sich durch eine gezielte Definition der Trainingsdaten ebenfalls umsetzen. Folglich könnte man spezielle Muster definieren, welche bekannte Fehlermuster darstellen. Ein Schema der folgenden Art wäre denkbar: tritt Log-Event A gefolgt von Log-Event B auf handelt es sich um Fehler X.
Zusammenfassend lässt sich sagen, dass mithilfe des oben dargestellten Algorithmus eine Komprimierung, eine *Anomaly Detection* sowie *Fault Detection* möglich ist.

4.3 Der Algorithmus im Überblick

Im Folgenden wird der vorgestellte Algorithmus in seinem Gesamtbild dargestellt und die grundlegenden Funktionen noch einmal zusammengefasst. Der Algorithmus unterteilt sich in die folgenden Abschnitte:

1. Die Erzeugung der **Lerndaten** wird durch einen Menschen vorgenommen. Die Daten stellen eine Zuordnung von einer Log-Event-Sequenz e_{in} zu einer anderen Log-Event-Sequenz e_{out} dar und sollen ein Muster m erzeugen. Es handelt sich um eine Abbildung $l : e_{in} \rightarrow e_{out}$. Die Sequenz e_{in} stellt eine Sequenz dar, die von dem Muster m erkannt werden soll. Sie stellt ein Beispiel aus einer

Log-Datei dar. Die Sequenz e_{out} stellt die Ausgabesequenz dar, die anstelle von den von m erkannten Log-Event-Sequenzen erzeugt werden soll. In e_{out} müssn die variablen Anteile der Log-Events, aus denen sich die Log-Event-Sequenz e_{in} zusammensetzt, speziell markiert werden. Die Markierung eines variablen Anteils x_v richtet sich nach dem Schema $\$\{x_v\}$. Die Lerndaten können vom „Lehrer“ über eine speziell formatierte XML-Datei definiert werden.

2. Die **Mustergenerierung** erzeugt aus der Log-Event-Sequenz e_{in} das Muster m. In m wird der variable Anteil der Log-Event-Sequenzen durch den regulären Ausdruck $(.*)$ ersetzt. Dieser erkennt beliebige Zeichen mit beliebiger Wiederholung. Die Mustergenerierung erzeugt ein internes Pattern-Model. Dieses ordnet dem erzeugten Muster m die Ausgabe e_{out} zu. Die Abbildung $l : e_{in} \rightarrow e_{out}$ aus den Lerndaten wird folglich in eine Abbildung $f : m \rightarrow e_{out}$ überführt. Neben der Abbildung f speichert das Pattern-Model zusätzliche Metainformationen, wie die Zuordnung der Platzhalter $(.*)$ zu den variablen Ausdrücken innerhalb der Ausgabe e_{out}. Dies ermöglicht eine spätere Ersetzung der Platzhalter durch die erkannten variablen Ausdrücke der erkannten Log-Event-Sequenzen.

3. Die **Mustererkennung** durchsucht eine gegebene Log-Datei l_d nach den gelernten Mustern. Wird ein Log-Event von einem Muster erkannt, so wird ein neuer Pattern-Scope geöffnet. Ein Pattern-Scope besteht aus n Mustern, die jeweils ein Log-Event der zu erkennenden Log-Event-Sequenz darstellen. Ein Scope gilt als geschlossen, sobald alle n Muster innerhalb dieses Scopes besetzt sind.

4. Die **Analyse** erzeugt aus allen geschlossenen Pattern-Scopes die entsprechende Ausgabe e_{out}. Durch eine gezielte Definition der Ausgabe e_{out} können die dazugehörige Log-Event-Sequenz komprimiert werden. Darüber hinaus besteht an dieser Stelle die Möglichkeit Anomalien bzw. konkrete Fehler innerhalb des Systems mithilfe der Log-Datei aufzudecken. Die Analyseergebnisse werden in einer Datei persistiert.

Mögliche Erweiterungen

Bei der Entwicklung des oben zusammengefassten Algorithmus ist dieser an einer einzelnen Log-Datei erprobt worden, jedoch vor dem Hintergrund Log-Dateien in der verteilten Integrationsumgebung RCE zu analysieren. In einem verteilten System erzeugt jede Instanz innerhalb dieses Systems ihre eigene Log-Datei. Folglich macht dies die Betrachtung mehrerer Dateien notwendig. Diese Anforderung ist bei der Implementierung des Algorithmus berücksichtigt worden. Prinzipiell würde die Möglichkeit bestehen mehrere Log-Dateien zu einer für den Algorithmus internen Log-Datei zu verschmelzen, d.h. es würde theoretisch die Möglichkeit bestehen verteilte Log-Event-Sequenzen zu erkennen. In der Praxis wäre ein solches Vorgehen jedoch aus verschiedenen Gründen problematisch. Zum einen würde die Information verloren gehen, welches Log-Event einer erkannten Log-Event-Sequenz zu welcher Instanz gehört, zum anderen müsste zwischen lokalen und nicht-lokalen Mustern unterschieden werden. Es könnte ansonsten vorkommen, dass identische Log-Events aus verschiedenen Instanzen ungewollt demselben Muster zugeordnet werden. Tritt dieser Fall auf, würde das Log-Event, welches zuerst auftritt, dem entsprechenden Scope zugeordnet werden. In einer verteilten Umgebung, insbesondere in RCE, kann man jedoch nicht davon ausgehen, dass die Zeitstempel der verschiedenen Instanzen synchronisiert sind. Dieses Problem ist noch offen und ungelöst.
Eine bessere Lösung wäre es, den Algorithmus dahingehend zu erweitern, dass die Analyse-Methode anstelle eines einzigen Log-Event-Blocks, der die zu analysierende Log-Datei darstellt, eine Menge an Tupeln als Inputparameter übergeben bekommt. Ein Tupel würde eine Zuordnung von Instanz zu Log-Event-Block darstellen und könnte folglich in der anschließenden Analyse berücksichtigt werden.
Die Grundsteine für diese Überlegung sind bereits gelegt worden, so besteht die Möglichkeit einer Log-Event-Sequenz eine *Entität*, d.h. eine Instanz, zuzuordnen. Darüber hinaus sind Methoden implementiert worden, die es erlauben Log-Event-Sequenzen mit anderen Log-Event-Sequenzen zu verschmelzen. Dies ermöglicht eine vereinfachte Kombination verteilter Log-Events. Darüber hinaus sind entsprechende Schnittstellen für eine erweiterte Analyse implementiert worden, die es dem Entwickler ermöglichen die Analyse der Pattern-Scopes durch neue Module zu ergänzen.

5 Fazit und Ausblick

Im Rahmen dieser Arbeit ist ein Algorithmus entwickelt worden, der mit Methoden des maschinellen Lernens, Muster für die automatisierte Analyse von Log-Dateien generiert. Die erzeugten Muster wurden dafür genutzt, Log-Dateien der verteilten Integrationsumgebung RCE, die im DLR entwickelt wird, zu analysieren. Die Analyseergebnisse sind in einem für den Menschen leicht verständlichen Bericht zusammengefasst worden. Hintergrund dieser Problemstellung war es das im Fehlerfall manuelle Durchsuchen großer und komplexer Log-Dateien durch ein automatisiertes Verfahren zu ersetzen.

Eine Log-Datei besteht aus einer geordneten Menge an Log-Events. Ein Log-Event stellt eine Aktivität innerhalb des Softwaresystems dar, von welchem diese protokolliert worden ist. Ein Log-Event setzt sich aus einem variablen und einem statischen Anteil zusammen und beginnt mit einem Zeitstempel, d.h. mit dem Zeitpunkt seines Auftretens. Dabei lässt sich beobachten, dass die Vereinigung mehrerer Log-Events einen Zusammenhang bilden kann. Diese logische Vereinigung wird Log-Event-Sequenz genannt. Die zu einer Log-Event-Sequenz zugehörigen Log-Events können in der Log-Datei verteilt sein und müssen nicht zusammen auftreten. Aus diesem Grund lag bei der Entwicklung des Algorithmus ein besonderer Fokus auf der Generierung und Erkennung von Mustern in Log-Event-Sequenzen.

Bevor die Entwicklung begonnen wurde, sind zunächst aktuelle Algorithmen, die sich dem Themenbereich der Mustererkennung in Log-Dateien zuordnen lassen, untersucht worden, um eine mögliche Überschneidung der Anforderungen zu evaluieren. Die aktuellen Algorithmen beschränken sich auf das Generieren von Mustern aus einzelnen Log-Events ohne dabei einen übergeordneten Zusammenhang zwischen mehreren Log-Events zu untersuchen. Die gefundenen Verfahren nutzen Methoden

des unüberwachten maschinellen Lernens, insbesondere das Clustering.
Die Anforderungen der Aufgabenstellung verlangten jedoch das Generieren von Mustern aus Log-Event-Sequenzen. Aus diesem Grund ist die Entscheidung getroffen worden einen Algorithmus zu entwickeln, der mithilfe des überwachten maschinellen Lernens aus einem Trainingsdatensatz Muster generiert. Die erzeugten Muster wurden anschließend dafür genutzt Log-Dateien aus RCE zu analysieren. Die Analyse umfasste zunächst eine Komprimierung, d.h. eine Zusammenfassung der von den Mustern erkannten Log-Event-Sequenzen in eine verkürzte und für den Menschen verständlichere Form.
Es ist zunächst eine passende Datenstruktur entworfen worden, welche dem Algorithmus als Input für die Mustergenerierung dient. Die Datenstruktur setzt sich aus einer Menge an Log-Events, einer Log-Event-Sequenz, zusammen. Diese Log-Event-Sequenz wird einer neuen Log-Event-Sequenz zugeordnet. Die neue Log-Event-Sequenz stellt dabei die komprimierte Ausgabe dar. Die Trainingsdaten werden im XML-Format definiert.
Die aus diesen Log-Event-Sequenzen generierten Muster sollten ähnliche Log-Event-Sequenzen, die sich in ihrem variablen Anteil unterscheiden, erkennen. Der Algorithmus ersetzt dafür mithilfe der Lerndaten die variablen Anteile der Sequenzen mit einem speziellen regulären Ausdruck, der als Platzhalter dient. Hierfür müssen die variablen Ausdrücke in den Lerndaten speziell kenntlich gemacht werden. Dies wird in der dazugehörigen Ausgabe umgesetzt. Folglich müssen alle variablen Ausdrücke innerhalb der komprimierten Ausgabe erneut auftreten.
Die anschließende Analyse erhält eine Log-Datei als Eingabeparameter. Eine Log-Datei stellt zu Beginn eine einzige große Log-Event-Sequenz dar und wird im Verlauf der Analyse in feinere Strukturen zerlegt. Für ein erkanntes Muster wird die entsprechende Ausgabe erzeugt und in einer neuen Log-Datei persistiert. Neben der Komprimierung lassen sich mithilfe gezielt gelernter Muster die Log-Dateien auf Anomalien untersuchen. Dabei lassen sich prinzipiell alle Konzepte, die in Kapitel 2.3.1 vorgestellt worden sind, umsetzen.
Der entworfene Algorithmus ist in der Programmiersprache Java implementiert worden. Für die Nutzung ist eine grafische Benutzeroberfläche (GUI) entwickelt worden. Der Nutzer kann über die GUI die Lerndaten erzeugen bzw. bereits vorhandene Lerndaten in einer vorliegenden XML-Datei dem Algorithmus zur Verfügung stellen.

Darüber hinaus können Log-Dateien mit dem trainierten Modell über die GUI analysiert werden.

Abschließend lässt sich sagen, dass der Algorithmus umgesetzt worden ist und dabei alle Anforderungen erfüllt. Für die Zukunft ist eine Überarbeitung bzw. Erweiterung der Lerndaten denkbar. Der Nutzer ist gezwungen alle variablen Ausdrücke innerhalb der Ausgabe zu markieren. Dies ist umständlich und kann für gewisse Sachverhalte unerwünscht sein. Eine gesonderte Markierung der variablen Ausdrücke außerhalb der spezifizierten Ausgabe ist denkbar. Der Lehrer könnte beispielsweise gesondert eine Liste der variablen Ausdrücke übergeben. Dieser Ansatz könnte in der vorhanden GUI leicht ergänzt werden. Darüber hinaus ist das übergeordnete Ziel der Analyse von mehreren verteilten Log-Dateien offen. Es verbleibt die Klärung, inwieweit der Algorithmus mit der Problematik der nicht synchronisierten Zeitstempel umgehen soll.
Im Allgemeinen scheint eine Zuordnung von Log-Events zu entsprechender Log-Datei denkbar und sinnvoll zu sein, da dem Algorithmus folglich die Information zur Verfügung stehen würde welches Log-Event in welcher Instanz des verteilten Systems aufgetreten ist. Diese Information könnte dafür genutzt werden einen kausalen Zusammenhang zwischen verteilten Log-Events herzustellen. Darüber hinaus müsste in dem entwickelten Programm ein weiteres Modul implementiert werden, welches diese Anforderungen umsetzt.

Der Algorithmus benötigt in seiner jetzigen Form für die Analyse eine geschulte Person, welche die Lerndaten erzeugt. Diese Vorgehensweise ist zeitaufwendig und fehleranfällig, da die kontinuierlich wachsenden Log-Dateien stetig und manuell auf mögliche neue Muster untersucht werden müssen, um eine Erkennung von Anomalien möglich zu machen. Besser wäre ein automatisiertes Vorschlagen neuer Muster durch den Algorithmus. Hierfür ist die Verwendung des in 2.4.2 vorgestellten Clustering-Algorithmus denkbar. Dieser beschränkt sich jedoch auf das Generieren von Mustern aus einzelnen Log-Events. Eine Erweiterung des Algorithmus auf das automatisierte Generieren von Mustern aus Log-Event-Sequenzen ist ohne speziell aufbereitete Lerndaten nicht trivial. Eine vorstellbare Lösung wäre es die erzeugten Muster aus den einzelnen Log-Events von einer geschulten Person kombinieren zu lassen, sodass diese kombinierten Muster die entsprechenden Log-Event-Sequenzen

erkennen könnten. Hierfür ist eine Ergänzung des bestehenden Lern-Algorithmus denkbar.
Darüber hinaus sollte die Mustererkennung gegen kleinere Änderungen innerhalb des Log-Formats robust sein. Hierfür wären Erweiterungen in Form von distanzbasierten Clustering-Verfahren denkbar. Diese könnten die Distanz der Log-Events zu einem möglichen zutreffenden Muster angeben. Befindet sich diese Distanz innerhalb eines bestimmten festgelegten Wertebereich, könnten sich leicht abgeänderte Log-Events trotzdem dem richtigen Muster zuordnen lassen.
Diese Ergänzung würde die Mustergenerierung folglich deutlich vereinfachen.

Literatur

[And98] James H. Andrews. „A Framework for Log File Analysis“. London, Ontario, Canada N6A 5B7: University of Western Ontario, Dept. of Computer Science, 1998.

[CBK09] Varun Chandola, Arindam Banerjee und Vipin Kumar. „Anomaly Detection : A Survey“. In: *ACM Computing Surveys* (2009).

[Ger09] Jan Gertheiss. *Cluster-Analyse - Vorlesung Statistik IV für Studierende mit Nebenfach Statistik*. Institut für Statistik, LMU München, 2009.

[KP99] Brian W. Kernighan und Rob Pike. *The Practice of Programming*. One Jacob Way, Massachusetts 01867: Addison Wesley Longman, Inc., 1999.

[Kre13] Jay Kreps. *The Log: What every software engineer should know about real-time data's unifying abstraction*. 2013. URL: `https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying` (Einsichtnahme: 18. 08. 2016).

[Luf15] Das Deutsche Zentrum für Luft- und Raumfahrt e.V. *Das DLR im Überblick*. 2015. URL: `http://www.dlr.de/dlr/desktopdefault.aspx/tabid-10443/637_read-251/#/gallery/8570` (Einsichtnahme: 17. 08. 2016).

[NT10] Meiyappan Nagappan und Mladen A. Vouk Test. „Abstracting Log Lines to Log Event Types for Mining Software System Logs“. In: *7th Working Conference on Mining Software* (2010).

[Ort03] Carina Ortseifen. *Einführung in die Cluster-Analyse mit SPSS*. 2003. URL: `https://www.urz.uni-heidelberg.de/imperia/md/content/urz/programme/statistik/spss-treff/2003-07-11.pdf` (Einsichtnahme: 22. 08. 2016).

[Pre08] Cambridge University Press. *Hierarchical clustering*. 2008. URL: `http://nlp.stanford.edu/IR-book/html/htmledition/hierarchical-clustering-1.html` (Einsichtnahme: 23. 08. 2016).

[Sch09] Thomas Schäfer. *Methodenlehre II - Clusteranalyse*. 2009. URL: `https://www.tu-chemnitz.de/hsw/psychologie/professuren/method/homepages/ts/methodenlehre/meth11.pdf` (Einsichtnahme: 22. 08. 2016).

[Sch12] Friedhelm Schwenker. *Data Mining*. Institut für Neuroinformatik, Universität Ulm, 2012.

[SB14] Shai Shalev-Shwartz und Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

[Spr] Gabler Wirtschaftslexikon Springer Gabler Verlag. *Dendrogramm*. URL: `http://wirtschaftslexikon.gabler.de/Archiv/2564/dendrogramm-v9.html` (Einsichtnahme: 23. 08. 2016).

[SV11] Prof. Petra Stein und Sven Vollnhals. *Grundlagen clusteranalytischer Verfahren*. Universität Duisburg-Essen, Institut für Soziologie, 2011.

[Vaa03] Risto Vaarandi. „A Data Clustering Algorithm for Mining Patterns From Event Logs“. In: *Proceedings of the 2003 IEEE Workshop on IP Operations and Management* (2003).

[Val01] Jan Valdman. „Log File Analysis“. Diss. Univerzitni 8, 30614 Pilsen, Czech Republic: University of West Bohemia in Pilsen, 07/2001.

[Vro10] Katerina Vrotsou. „Everyday mining - Exploring sequences in event-based data“. Diss. Department of Science und Technology, Linköping University, SE-601 74 Norrköping, Sweden: Linköping University, Department of Science und Technology, 2010.

[Xu+09] Wei Xu u. a. „Detecting Large-Scale System Problems by Minng Console Logs“. In: *SOSP '09' Proceedings of the ACM SIGOPS 22nd symposium on Operating system principles* (2009).

[YPZ12] Ding Yuan, Soyeon Park und Yuanyuan Zhou. „Characterizing Logging Practices in Open-Source Software“. University of California und University of Illinois, 2012.

[Zwi14] Tim Zwietasch. „Detecting Anomalies in System Log Files using Machine Learning Techniques“. Bachelorarbeit. University of Stuttgart, 2014.

A Anhang

A.1 Die Grafische Benutzeroberfläche

Für die Nutzung des Programms ist eine grafische Benutzeroberfläche (GUI) implementiert worden. Es wurde hierfür das Framework *JavaFX* verwendet. Dieses ist seit Java 8 Teil des JDK. In Abbildung A.1 ist das Hauptfenster der GUI dargestellt. Im linken Teil der GUI befindet sich der Verzeichnisbaum des Arbeitsverzeichnisses. In diesem wird die XML-Datei für das Pattern-Model sowie die Ergebnisdatei erzeugt. Die Trainingsdaten können in die GUI geladen werden. Dies geschieht über das Dropdown-Menü „File“. Es können bestehende Pattern-Models, Trainingsdaten sowie Log-Dateien für die Analyse geladen werden. Die Trainingsdaten können ebenfalls in dem Tab „Training“ spezifiziert werden. In dem Textfeld „Input“ werden die Log-Events definiert, aus denen das Muster erzeugt werden soll. In dem Textfeld „Output“ wird die dazugehörige Ausgabe angegeben. In dieser werden die Schlüsselwörter angegeben. Mithilfe des Buttons „Train“ wird das Pattern-Model trainiert. Das erzeugte Muster mit dazugehöriger Ausgabe erscheint anschließend in der unten angezeigten Tabelle. Befindet sich bereits ein trainiertes Pattern-Model innerhalb des Arbeitsverzeichnisses, wird dieses nach Applikationsstart in der Tabelle angezeigt. Es wird automatisch während des Trainings ergänzt. Mithilfe des Buttons „Save“ werden die Trainingsergebnisse in dem Pattern-Model in der XML-Datei persistiert. Das Training ist in Abbildung A.2 dargestellt. Über das Dropdown-Menü „Window“ kann ein neuer Tab für die Analyse geöffnet werden. Anschließend kann eine Log-Datei in die GUI geladen werden und mithilfe des Buttons „Analyze“ wird diese Datei analysiert. Die Analyse ist in Abbildung A.3 dargestellt. Für die Darstellung der Log-Dateien handelt es sich um eine Tabelle mit zwei Spalten. Eine Spalte wird für die Zeitstempel genutzt während die andere Spalte für die Log-Events reserviert

ist. Die angesprochene Komprimierung der Zeitstempel ist in Abbildung A.3 in der ersten Zeile dargestellt.

Abbildung A.1: Das Hauptfenster der GUI

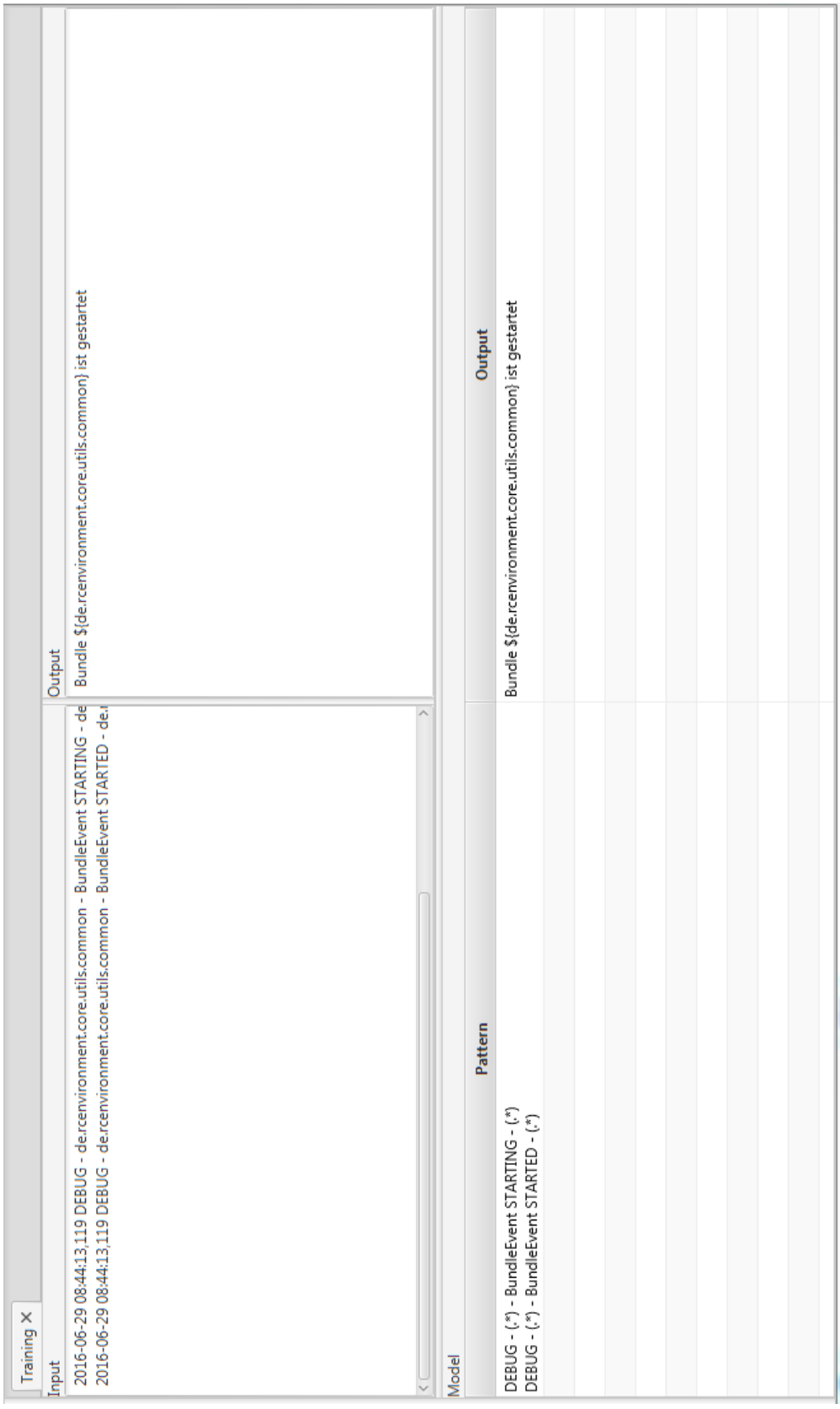

Abbildung A.2: Das Training innerhalb der GUI

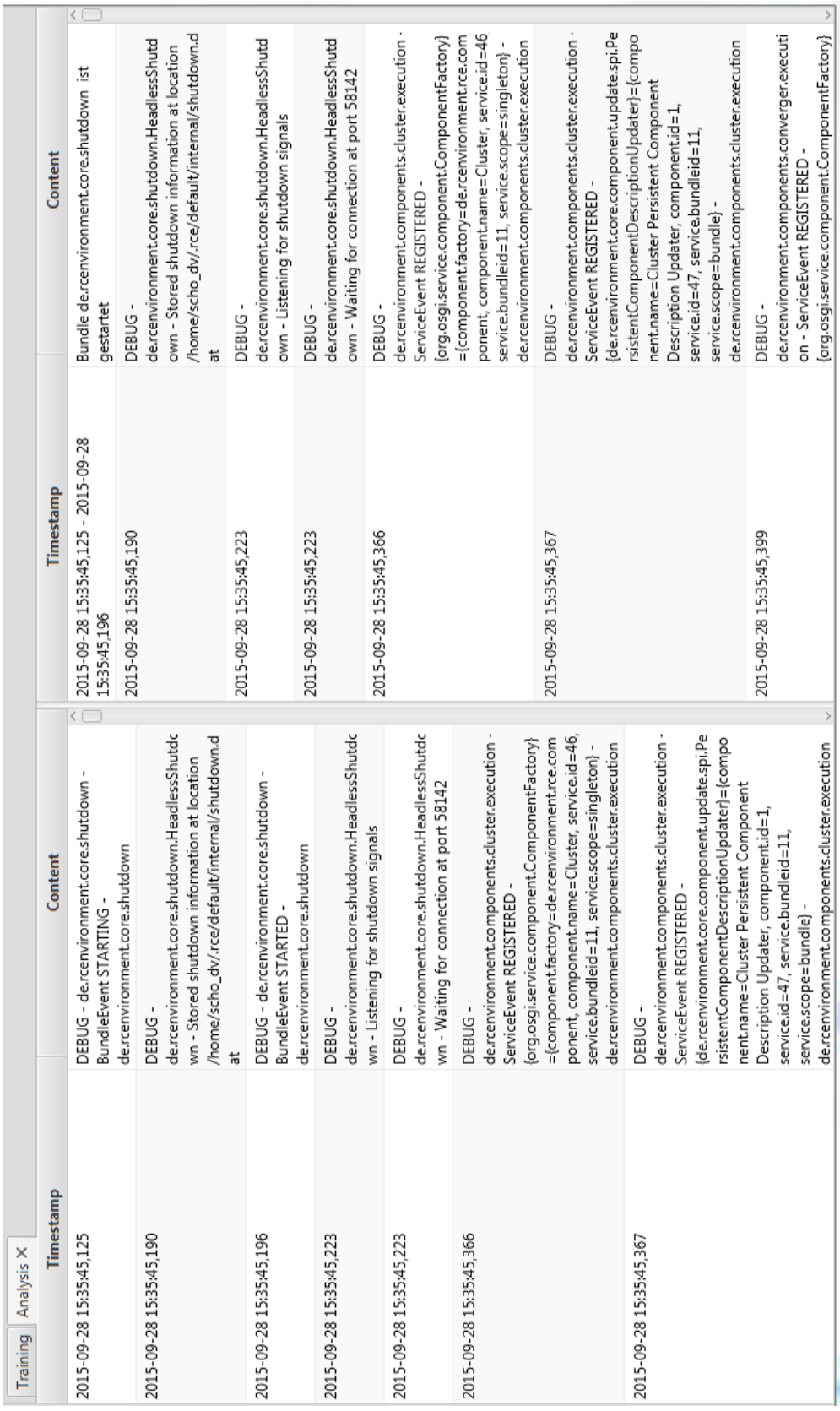

Abbildung A.3: Die Analyse innerhalb der GUI