Complex Event Processing am Beispiel von Esper

Martin Steinbach
Institut für Informatik
Universität Rostock
martin.steinbach@uni-rostock.de

ABSTRACT

Nachdem die Masseneuphorie über *CEP* vor einigen Jahren abgeflaut ist, tritt *CEP* mittlerweile als wichtiger Bestandteil der Analysearchitektur in den Vordergrund. Das ist nicht zuletzt der günstigen Sensortechnik und der Überwachung von Prozessen oder Märkten geschuldet. In all diesen Anwendungsgebieten werden Ereignisse erzeugt, und das oft in einem Ausmaß, dass eine Speicherung unmöglich wird oder der Informationsgehalt der Ereignisse schon nach kurzer Zeit sinkt. *Complex Event Processing* bietet eine Möglichkeit, diese Ereignisse annähernd in Echtzeit zu analysieren und zu korrelieren, um neues Wissen zu erzeugen und auf Basis dieser neuen Erkenntnisse zu reagieren. Die vorliegende Arbeit bietet eine kompakte Einführung in das Thema *Complex Event Processing* und deren zugrunde liegenden Anfragesystematiken. Hauptaugenmerk liegt jedoch auf der exemplarischen Veranschaulichung dieser Systematiken anhand der Anfragesprache *EPL* der *CEP*-Software Esper.

Keywords

EDA, CEP, EPL, Esper

1. EINLEITUNG

Jede Aktion in der IT-gestützten Welt erzeugt Informationen in Form von Daten. Dabei spielt es keine Rolle, ob ein Verweis in einem sozialen Netzwerk verwendet wird, ob ein Sensor einen Messwert meldet oder jemand eine Aktie kauft. Werden diese Daten betrachtet, scheint der Informationsgehalt gering und die Datenmenge überschaubar. Aus diesem Grund lässt sich aus diesen Einzelinformationen kaum eine nutzbringende Auskunft für ein Gesamtsystem konstruieren. In der Praxis relevante Fragestellungen wären zum Beispiel: Wie oft wird ein Verweis innerhalb einer Zeitspanne ausgelöst und zu welcher Tageszeit hauptsächlich? Wie viele Messwerte eines Sensors werden benötigt, um konkrete Aussagen zu einem Messobjekt machen zu können? Wann und welche Menge an Aktien eines Unternehmens werden über einen Zeitraum erworben beziehungsweise verkauft?

Um diese exemplarischen Fragen zu beantworten, steigt die Anzahl der benötigten Daten sehr schnell an, und eine Verarbeitung mit anschließender Analyse der Daten bedarf wesentlich mehr Aufwand und Zeit. Sollen zudem nicht nur historische Daten analysiert, sondern möglichst ohne Zeitverzögerung eine Antwort auf aktuell erhobene Daten gefunden werden, dann bietet sich die Technologie *Complex Event Processing (CEP)* an. Mithilfe der von *CEP* angebotenen Verfahren lassen sich riesige und aktuellste Datenmengen nahezu direkt verarbeiten. Im Gegensatz zu Datenbankmanagementsystemen, in denen auf einer endlichen Menge an Daten operiert wird, existieren auch für *CEP* fertige Softwarelösungen zur systematischen Analyse von massiven Datenströmen. Eine dieser *CEP-Engines* ist Esper. Laut [10] besitzt Esper eine sehr große Verbreitung, wird im kommerziellen Umfeld von namhaften Unternehmen verwendet und ist universell einsetzbar. Esper ist freie Software[1] und steht damit der Öffentlichkeit für jeden Zweck zur Verfügung.
Neben der Erklärung der Esper-eigenen Abfragesprache *EPL* werden zuvor die Grundlagen von *Complex Event Processing* erläutert. Darunter fällt die Klärung essentieller Begriffe, eine Einordnung von CEP und die Erklärung von Abfragesystematiken.

2. CEP AM BEISPIEL VON ESPER

Um die Mechanismen der Anfragesprache *EPL* zu verstehen, muss zuvor in Abschnitt 2.1 die Frage geklärt werden, wie sich *Complex Event Processing* eigentlich charakterisieren lässt und wie es in das System der Datenverarbeitung eingegliedert werden kann. Darüber hinaus werden in diesem Abschnitt auch grundlegende Begriffe des *CEP*-Kontextes geklärt. Kapitel 2.2 befasst sich mit allgemein gültigen Aussagen zu Anfragesprachen und stellt geläufige Anfragealgebren vor. Der sich anschließende Abschnitt 2.3 stellt die Funktionsweise von Esper aus Sicht eines Nutzers dar und listet in einem kurzen Quellcodefragment auf, wie Esper in Betrieb genommen werden kann. Der ausführlichste Teil, Abschnitt 2.4, zeigt exemplarisch anhand eines gegebenen Ereignisstromes, wie sich die Theorie der vorherigen Abschnitte in *EPL* realisieren lässt.

[1] General Public License v2

2.1 Einordnung von CEP

CEP kann als Bestandteil der *Event-Driven Architecture (EDA)* verstanden werden. *EDA* unterscheidet sich als Architekturstil grundlegend von anderen Stilen; so findet in *EDA* keine schrittweise Abarbeitung von vorher definierten Anweisungen statt, um Daten zu verarbeiten. Wie in [7] beschrieben, existieren stattdessen ereignisgesteuerte Komponenten, deren Interaktion untereinander ebenfalls über Ereignisse erfolgt. Damit ist der Begriff des Ereignisses ein elementarer und wird in Abschnitt 2.1.1 ausführlich behandelt. *CEP* kann wiederum als Komponente in einem *EDA*-System zum Einsatz kommen, wie in Abbildung 1 aus [3].

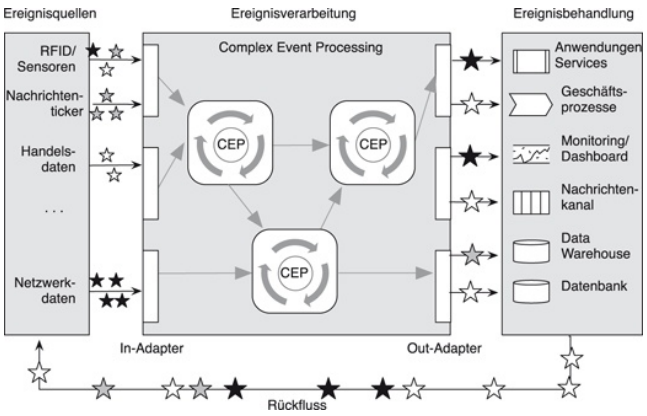

Figure 1: *EDA*-Struktur

CEP ist in erster Linie als Sammelbezeichnung für verschiedene Paradigmen und Techniken für die Analyse und Verarbeitung von Ereignissen zu verstehen. Ziel ist es, Wissen aus einer kontinuierlich nachströmenden Menge an Daten in Form von Ereignissen zu generieren. Dies geschieht zum Beispiel durch Korrelation oder Gruppierung von Ereignissen durch zuvor definierte Regeln. Dabei kann eine Vielzahl von Ereignisquellen (Abbildung 1) existieren, welche fortlaufend neue Ereignisse generieren. Trifft eine Regel auf eine Menge an Ereignissen zu, so wird daraus ein komplexes Ereignis ([7]) generiert, welches abermals als Ereignis für eine weitere *CEP*-Instanz dienen kann. Ein komplexes Ereignis wird auch verwendet, um verschiedenartige Meldungen zu generieren, Prozesse zu initiieren oder um als Grundlage für Visualisierungen zu dienen. In [3] wird ein Grundzyklus für ereignisgesteuerte Systeme identifiziert (Abbildung 2), der aus den drei Schritten Erkennen, Verarbeiten und Reagieren besteht.

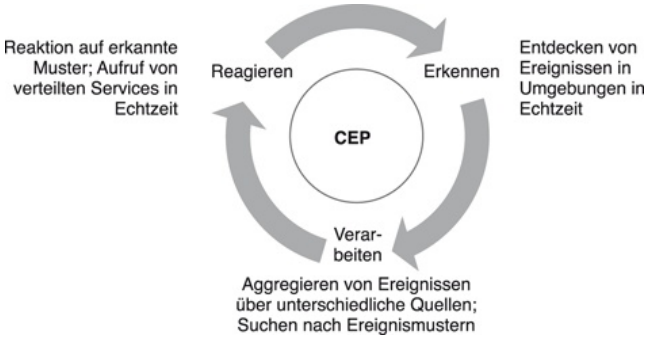

Figure 2: *CEP*-Zyklus

Den Beginn stellt dabei das *Erkennen* dar, relevante Informationen (zum Beispiel Messwerte) werden ohne Verzögerung als Ereignisse interpretiert. Während der *Verarbeitung* wird die Mustererkennung auf einem oder mehreren Ereignisströmen durchgeführt. Sobald Muster erkannt werden, wird mit Meldungen oder mit der Generierung komplexer Ereignisse reagiert.
Laut [4] gibt es zwei grundlegende Arten, um komplexe Ereignisse zu identifizieren. Entweder über bereits bekannte Muster, die durch Regeln in Ereignisanfragesprachen formuliert werden können, oder über unbekannte Muster. Die Detektierung von unbekannten Mustern benötigt allerdings Technologien wie *MachineLearning* und *DataMining*, daher wird in dieser Arbeit nicht darauf eingegangen.

2.1.1 Begriffsbestimmungen

Datenströme/Ereignisströme sind kontinuierliche, kleinteilige Datensätze in der zeitlichen Reihenfolge ihres Auftretens oder ihrer Messung. Die Daten weisen eine geringe Komplexität auf und beziehen sich nur auf ein Datum (zum Beispiel den Messwert eines Sensors oder den Kurs eines Wertpapiers.). Die Ströme sind endlos und meist hochfrequent und massiv, daher können die Daten nicht persistent gespeichert werden.
Jeder Datensatz in einem Datenstrom bildet ein eigenes ***Ereignis***. Im Allgemeinen kann laut [7] ein Ereignis alles sein was eintreten kann, wie zum Beispiel ein Erdbeben, eine Finanztransaktion, das Betätigen einer Taste oder die Oktoberrevolution. Im Speziellen ist ein Ereignis ein aufbereiteter Datensatz, welcher die Informationen eines Ereignisses beinhaltet und für die rechnergestützte Verarbeitung angepasst ist. Dazu ist es notwendig, dass zum Informationsgehalt des Ereignisses (den Kontextinformationen) auch eindeutige und strukturierte Metadaten erfasst werden. Exemplarisch kann ein Ereignis folgendermaßen aufgebaut sein ([5]):

Table 1: Ereignisaufbau

Metadaten	
Ereignistyp	`Kursänderung`
Ereignisquelle	`Frankfurt`
Zeitstempel	`2018-11-21 22:14:00`
ID	`98127634`
Kontextinformation	
`Name: GCME AG`	
`Einkaufkurs: 32.5`	
`Letzter Kurs: 40.8`	
`Differenzbetrag: 5.7`	
`Aktueller Kurs: 42.1`	

Die Felder *Ereignistyp* und *Ereignisquelle* sind nach [3] optional. Wie aus den Metainformationen ersichtlich wird, stehen alle Ereignisse in impliziter Beziehung zueinander.
Im Verarbeitungsschritt (Abbildung 2) werden Beziehungen zwischen Ereignissen gesucht; diese werden durch ***Ereignismuster*** beschrieben. Die Mustererkennung wird nur über ein bestimmtes Zeitintervall des Ereignisstromes ausgeführt und durch Ereignisanfragesprachen definiert. In [3] werden drei Arten von Ereignismustern unterschieden. Können Muster ausschließlich durch boolesche Operatoren der Aussagenlogik festgelegt werden, so fallen sie in die Kategorie der *einfachen Ereignismuster*. Werden hingegen speziellere Operatoren nötig, um zum Beispiel die Reihenfolge oder Zeitfenster in den Ereignissen auszudrücken, gehören sie zur Kategorie der *komplexen Ereignismuster*. Zur Kategorie der *abstrakten Ereignismuster* gehören die Muster, welche aus einem bereits erkannten Muster komplexe Ereignisse erzeugt haben, um diese auf einer höheren Abstraktionsebene wieder zur Verfügung zu stellen.
Ein ***komplexes Ereignis*** ist eine Menge von Ereignissen, die durch ein Ereignismuster beschrieben sind.
Ereignisregeln sind die syntaktische Abstraktion von Ereignismustern und werden mithilfe einer ***Anfragesprache*** erstellt; laut [3] existieren für diese Sprachen keine einheitlichen Standards. Daher gibt es eine Fülle an Sprachen, die aber alle einige Eigenschaften teilen. So bestehen in einer Anfragesprache formulierte Regeln aus einer Prämisse und einem Aktionsteil. Wenn das Muster beziehungsweise die Bedingung in der Prämisse erfüllt ist, wird der Aktionsteil ausgeführt. Anfragesprachen sind grundsätzlich deklarativ, es wird daher ein Modell beschrieben; demnach müssen keine Verfahren zur Mustererkennung implementiert werden. In [4] werden auch reaktive Regeln erwähnt; diese definieren, wie auf komplexe Ereignisse reagiert werden soll.

2.1.2 Beispielanwendungen

CEP kommt überall dort zum Einsatz, wo Daten möglichst in Echtzeit analysiert werden sollen, um Erkenntnisgewinn zu erlangen. Die Finanzbranche ist sicherlich einer der größten Sektoren, in dem *CEP* zum Einsatz kommt. Dabei muss es sich nicht um zeitnahe Analysen der Marktsituation oder automatisierten Handel handeln, auch die Erkennung von Kreditkartenmissbrauch wird mittels *CEP* automatisiert.
CEP wird zudem zur Überwachung von Industrieanlagen und im Automobilbereich eingesetzt, um schnell die Informationen einer Vielzahl von Sensoren zu interpretieren. Auch das CERN wäre ohne *CEP* nicht in der Lage, die Daten der verschiedenen Detektoren am LHC auszuwerten, denn eine Speicherung der Daten zur späteren Analyse ist aufgrund der Menge unmöglich.
Zu erwähnen ist auch die *DEBS*-Konferenz[2] und deren *Grand Challenges*, ein seit 2010 jährlich veröffentlichtes Problem, was auf Basis einer riesigen Datenmenge zu lösen ist.

2.2 Anfragesprachen

Mithilfe von Ereignisanfragesprachen lassen sich Daten aus einer Ereignisfolge, eine endliche Menge an Ereignissen innerhalb des endlosen Ereignisstromes, extrahieren, verdichten, zeitliche Zusammenhänge herstellen und Aktionen festlegen.

2.2.1 Definition von Ereignisregeln

In diesem Abschnitt wird eine Übersicht über die zur Verfügung stehenden Mengenalgebren in Anfragesprachen gegeben, dabei wird sich auf das Vorgehen in [3] berufen.
Seien A, B, C Ereignistypen und a, b, c zugehörige Ereignisinstanzen, wobei gilt $a \in A$, $b \in B$ und $c \in C$. Eine Ereignisfolge wird durch $a_1 a_2 a_3 b_1 c_1 b_2 1_4$ beschrieben.
Jede Ereignisalgebra enthält spezielle Operatoren, die auf Ereignisfolgen angewendet werden können und dabei zum Beispiel zeitliche, kausale oder fachliche Zusammenhänge zwischen Ereignissen beschreiben. In ***ereignistypbasierten Mustern*** legt der Sequenzoperator die zeitliche Reihenfolge des Auftretens von Ereignistypen fest. Zum Beispiel $A \rightarrow B$. Diese Regel akzeptiert die Ereignisfolge $c_1 a_1 a_2 c_2 b_1$. Des Weiteren existieren die booleschen Operatoren $\wedge, \vee$, die keine Reihenfolge des Auftretens beschreiben. Ebenso existiert die Negation: $\neg A$, mit deren Hilfe explizit das Nichteintreten eines Ereignisses geprüft werden kann. Der Negationsoperator kann nur auf eine endliche Untermenge des gesamten Ereignisstromes angewandt werden, zum Beispiel innerhalb eines Fensters oder einer Sequenz. So darf im folgenden Beispiel Ereignistyp B nicht zeitlich zwischen einem Ereignis des Typs A und C liegen: $A \rightarrow \neg B \rightarrow C$. Die Kombination dieser Operatoren ermöglicht es, komplexe Muster in Ereignisfolgen zu erkennen. Neben dem Ansatz, Mustererkennung auf Basis der Aussagenlogik durchzuführen, existieren auch andere Ansätze. Der Ende 2016 verabschiedete Standard [6] für die Abfragesprache SQL von relationalen Datenbanken beschreibt Muster mithilfe von regulären Ausdrücken. Eine kurze Übersicht zu dieser alternativen Möglichkeit, um Ereignismuster zu spezifizieren, wird in Kapitel 2.5 behandelt.

Eine weitere Algebra, welche von Anfragesprachen implementiert wird, ist die Algebra über ***Kontextbedingungen***. Mithilfe dieser Algebra ist es möglich, direkt auf Attribute der Ereignisinstanzen über den .-Operator zugreifen zu können. Damit besteht die Möglichkeit, die Attribute in Relation zu setzen und Rechenvorschriften zu definieren. Zur Verfügung stehen dabei die numerischen Operatoren und Vergleichsoperatoren, aber auch Operatoren für Zeichenketten sind denkbar oder gesondert definierte Methoden. Um in dieser Algebra verschiedene Entitäten von Ereignistypen

[2]Distributed Event-Based Systems: http://debs.org/debs-conferences/

zu betrachten, ist ein Operator zur Namenssubstitution vorhanden. Dieser Operator weist einem Ereignis eines Ereignistyps einen neuen Namen zu. Dieser binäre Operator könnte zum Beispiel *AS* lauten. Das folgende Muster würde demnach die Werte des Attributes *humidity* von einem Ereignis mit dem Ereignistyp *A* mit einem Ereignis vom nachfolgenden Ereignistyp *B* auf Gleichheit prüfen.

$$((A\,AS\,a)) \rightarrow (B\,AS\,b)) \wedge (a.humidity = b.humidity)$$

Aufgrund des endlosen Ereignisstromes können die zuvor beschriebenen Muster nicht auf die ganze Menge der Ereignisse angewendet werden, da diese nicht gespeichert werden können oder nach kurzer Zeit schon nicht mehr von Interesse sind. Daher kommen ***sliding windows*** zum Einsatz. Diese erlauben es, nur ein gewisses Segment des Ereignisstromes für die Mustererkennung zu betrachten. Dabei werden laut [3] und [5] zwei Fensterarten unterschieden. Das Zeitfenster berücksichtigt alle Ereignisse, die in einem zuvor festgelegten Zeitraum eintreffen. Wird hingegen eine maximale Anzahl an Ereignissen innerhalb eines Ereignisstromes betrachtet, wird von Längenfenstern gesprochen. Abbildung 3 aus [9] verdeutlicht diesen Mechanismus. Dabei können die grauen Quader als beliebige Ereignisse oder als Ereignisse, die einem Ereignismuster entsprechen, verstanden werden.

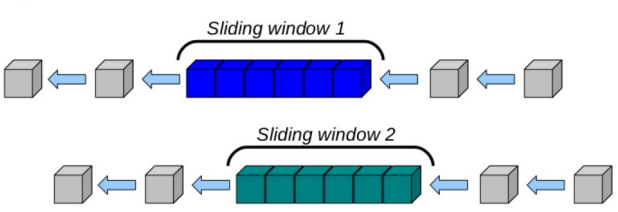

Figure 3: sliding windows

Eine exemplarische Ereignisregel mithilfe von *sliding windows* könnte folgendermaßen aussehen:

$$(A \rightarrow B)[win : time : 5min] \rightarrow C$$

Dieses Muster ist erfolgreich, wenn innerhalb eines fünfminütigen Zeitfensters ein Ereignis des Ereignistyps B auf ein Ereignis des Ereignistyps A folgt und anschließend ein Ereignis vom Ereignistyp C eintritt.
Da es möglich ist, mehrere Fensterinstanzen auf einen oder mehrere Ereignisströme parallel anzuwenden, kann mithilfe eines **Verschiebefaktors**, wie er in [5] beschrieben ist, die Schnittmenge zweier aufeinanderfolgender Fenster angegeben werden. Dabei unterscheidet [5] nochmals in die *Rolling Windows*, bei denen der Verschiebefaktor kleiner als die Länge des Fensters ist und zwei aufeinanderfolgende Fenster somit eine Schnittmenge besitzen, und *Tumbling Windows*, deren Verschiebefaktor größer oder gleich der Fensterlänge ist. Aufeinanderfolgende *Tumbling Windows* sind zueinander disjunkt. Ist der Verschiebefaktor größer als die Fensterlänge, befinden sich zwischen ihnen nicht betrachtete Ereignisse. Abbildung 4 aus [5] zeigt den Einfluss des Verschiebefaktors auf aufeinanderfolgende Fensterinstanzen.

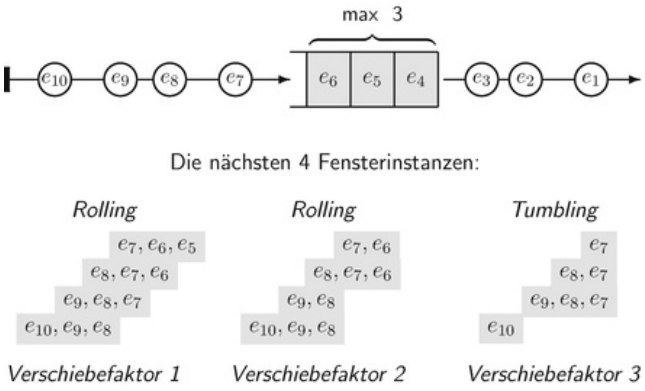

Figure 4: Verschiebefaktor

2.2.2 *Definition von Aktionen*

Wird ein Ereignismuster erkannt und ist damit die Prämisse einer Ereignisregel erfüllt, kann eine Aktion ausgeführt werden. Eine Aktion kann die Erstellung neuer Ereignisse sein oder das Ausführen von Diensten beziehungsweise Versenden von Meldungen.
Werden neue Ereignisse erzeugt, so ist von der Ereignistransformation die Rede. In [3] wird zwischen zwei Transformationen unterschieden: Den Transformationen, die keine Informationen zu einer transformierten Ereignisfolge hinzufügen, und diejenigen, die dies tun. So kann im einfachsten Fall ein Ereignis eines gewissen Ereignistyps einem anderen Ereignistyp zugeordnet werden. Aktionen in Anfragesprachen erlauben auch die Filterung von Ereignissen. Mit einer gefilterten Menge an relevanten Ereignissen kann eine Leistungssteigerung der Musterkennung erfolgen. Zudem lässt sich eine Menge an Ereignissen zu einem neuen Ereignis zusammenfassen.
Darüber hinaus kann auch der Informationsgehalt von Ereignissen verändert werden. Zum Beispiel kann im Zuge der Normalisierung eines Datenformats eine Information hinzugefügt oder entfernt werden. Das Generieren von komplexen Ereignissen erfolgt über Korrelation mehrerer einfacher Ereignisse, aus denen sich domänenspezifisches Wissen ableiten lässt. Als Beispiel soll die Erkennung einer *BruteForce*-Attacke dienen:

$$
\begin{aligned}
&CONDITION: \\
&\quad (FailedLoginAttempt\ AS\ f)[win : time : 5min] \\
&\quad \wedge \quad f.SourceIP\ =\ '' :: 1'' \\
&\quad \wedge \quad f.sum(count)\ AS\ FailCounter \\
&\quad \wedge \quad FailCounter >= 10 \\
&ACTION: \\
&\quad create \quad BruteForceAttack(SourceIP = f.SourceIP, \\
&\qquad Time = timestamp())
\end{aligned}
$$

2.3 Esper

Esper ist eine *Complex Event Processing Engine* und stellt eine Laufzeitumgebung und einen Compiler für die eigene Ereignisanfragesprache *Event Processing Language (EPL)* zur Verfügung. Der Compiler übersetzt *EPL*-Muster in Bytecode für die *Java Virtual Machine*. Esper ist ebenfalls in der Programmiersprache Java implementiert, mit Nes-

per existiert allerdings auch eine Implementierung in C#. Da nicht auf einer endlichen Menge an Daten nach Mustern gesucht wird, lässt sich Esper eher als eine invertierte Datenbank verstehen. Zu Beginn wird dem Compiler, mithilfe einer zuvor erstellten Konfiguration, das Ereignisschema mitgeteilt. Im Anschluss wird ihm ein *EPL*-Statement zur Übersetzung übergeben. (Abbildung 5).

```
EPCompiler c = EPCompilerProvider.getCompiler();
Configuration conf = new Configuration();
conf.getCommon().addEventType(PersonEvent.class);

CompilerArguments cargs =
    new CompilerArguments(conf);
EPCompiled epCompiled;
epCompiled = c.compile("@name('statement')
    select name, age from PersonEvent
    where (age/5) > 5", cargs);
```

Figure 5: Initialisierung des Compilers

Die Laufzeitumgebung verwendet sodann das zuvor übersetzte Statement und wendet es auf einen Ereignisstrom an. Ereignisse sind in diesem Fall Objektinstanzen der Klasse *PersonEvent* (siehe Abbildung 5). Ein *Listener* kann die zutreffenden Resultate verarbeiten. In diesem Beispiel (Abbildung 6) aus [1] werden die Ereignisparameter *name* und *age* ausgegeben.

```
EPRuntime rt =
    EPRuntimeProvider.getDefaultRuntime(conf);

EPDeployment deployment;
deployment =
    rt.getDeploymentService().deploy(epCompiled);

EPStatement statement =
    rt.getDeploymentService().getStatement(
        deployment.getDeploymentId(),"statement");

statement.addListener(new UpdateListener() {
public void update(EventBean[] newData,
    EventBean[] oldData, EPStatement statement,
    EPRuntime rt) {

    String name = (String) newData[0].get("name");
    int age = (int) newData[0].get("age");
    System.out.println(
        String.format("Name: %s,
            Age: %d",name, age));
}});
```

Figure 6: Initialisierung der Laufzeitumgebung

Um an die Esper-Laufzeitumgebung Ereignisse zu senden, müssen neue Ereignisinstanzen erzeugt und an die Laufzeitumgebung gesendet werden, wie in Abbildung 7 gezeigt.

```
for (int i = 0; i <= 100; i++) {
runtime.getEventService().sendEventBean(
    new PersonEvent("Peter", i), "PersonEvent");
}
```

Figure 7: Senden von Ereignissen

2.4 Die Anfragesprache EPL

Die *Event Processing Language* von Esper ist stark an *SQL* angelehnt, bietet aber Erweiterungen für die Verarbeitung von Ereignissen in einem endlosen Ereignisstrom. Ähnlich wie bei SQL ist ein Statement folgendermaßen aufgebaut:

```
select DATENAUSWAHL from EREIGNISSTROM where FILTER
```

Um Anfragen an einen Ereignisstrom zu validieren, bietet der Entwickler von Esper, EsperTech Inc., das Werkzeug *EPL*-Online[3] an. Basierend auf den Voreinstellungen in *EPL*-Online werden die nachfolgenden Beispielanfragen an den Ereignisstrom `StockTick` gerichtet. Der zugehörige Ereignistyp wurde zuvor durch das folgende EPL-Statement definiert. Die neu erzeugten Ereignisse werden als *callbacks* durch einen *Listener* entgegengenommen und sind als Ausgabe verfügbar.

```
create schema StockTick(symbol string, price double);
```

Die verwendeten Datentypen sind stark an die primitiven Datentypen der Programmiersprache Java angelehnt (`boolean, integer, long, double, float, byte`). Es existiert aber, für die Speicherung von unbegrenzt großen Zeichenketten, auch der Datentyp `string`

Für die Beispiele wird folgender Ereignisstrom verwendet.

```
StockTick={symbol='YHOO', price=65}
StockTick={symbol='IBM', price=141}
t=t.plus(2 seconds)
StockTick={symbol='IBM', price=142}
StockTick={symbol='YHOO', price=62}
t=t.plus(2 seconds)
StockTick={symbol='IBM', price=146}
StockTick={symbol='YHOO', price=63}
t=t.plus(2 seconds)
StockTick={symbol='YHOO', price=64}
StockTick={symbol='IBM', price=147}
t=t.plus(6 seconds)
```

Die Variable `t` symbolisiert die vergangene Zeit zwischen dem Eintreten von Ereignissen. Dabei sind `t = t+2000` und `t=t.plus(2 seconds)` äquivalente Ausdrücke.

[3] http://esper-epl-tryout.appspot.com/epltryout/mainform.html

Das einfachste Muster, das mithilfe von *EPL* beschrieben werden kann, ist die Filterung aller Ereignisse.

```
select * from StockTick();
```

Einfache Bedingungen lassen sich mit `where` realisieren, zur Formulierung stehen folgende Relationen bereit: =, <, >, >=, <=, ! = und `is null` sowie `is not null`. Zusätzlich lässt Esper noch eine weitere Schreibweise für Bedingungen zu. Diese zweite Notation ist äquivalent zur ersten, erhöht jedoch die Lesbarkeit in den später behandelten ereignistypbasierenden Mustern.

```
select * from StockTick() where price > 100;
select * from StockTick(price > 100);
```

Eine Aggregation über alle Ereignisse kann mittels vordefinierter Funktionen erfolgen. In *EPL* stehen neben den einfachen Funktionen wie `avg(), count(), sum()` auch statistische Methoden zur Verfügung (siehe Tabelle 10.5 in [1]).

Die nächste Anfrage zählt die Anzahl aller Ereignisse und bildet den Durchschnitt über alle Kurspreise der IBM-Aktie.

```
select count(*),avg(price) from StockTick
where symbol='IBM'
```

Die Aggregationsfunktionen können nach `select` angewendet werden, nicht jedoch innerhalb einer `where`-Bedingung. Um Aggregationsfunktionen auch als Bestandteil einer Bedingung zu verwenden, existiert die `having`-Klausel. Die `having`-Anweisung wird meistens in Zusammenhang mit der Instruktion `group_by` verwendet. Das sich anschließende Beispiel zeigt das Zusammenspiel der beiden Anweisungen. Dabei wird jeweils über einen fünf Sekunden langen Zeitraum die Summe der Aktienpreise, gruppiert nach ihrem *symbol*, berechnet. Ein Ereignis wird nur dann erzeugt, wenn die Summe einer Symbolgruppe in einem fünf Sekunden langen Intervall 400 überschreitet.

```
select symbol, sum(price) from StockTick#time(5 sec)
group by symbol having sum(price) > 400
```

Das Ergebnis dieser Anfrage (Tabelle 2) auf den zuvor festgelegten Ereignisstrom lautet demnach:

Table 2: Bedingung mit Aggregation

Symbol	Summe	Rechnung
IBM	429	141+142+146
IBM	435	142+146+147

Die zuvor gezeigten Beispiele eignen sich nicht zur Anfrage auf einem endlosen Ereignisstrom, da alle Ereignisse angefragt werden. Ein endgültiges Ergebnis würde ein Ende des Ereignisstromes voraussetzen, daher finden im Rahmen von *CEP* die in Kapitel 2.2.1 beschriebenen Fenster Verwendung. Die anschließende EPL-Anfrage setzt ein auf Zeit basierendes *sliding window* ein und berechnet den Durchschnittspreis der IBM-Aktie der letzten 6 Sekunden:

```
select avg(price) from StockTick.win:time(6 sec)
where symbol='IBM'
```

Da es sich hierbei um ein normales Zeitfenster ohne expliziten Verschiebefaktor handelt (siehe auch Abschnitt 2.2.1), werden insgesamt 7 Ergebnisse ausgegeben. Trifft ein Muster auf ein Ereignis zu, so bekommt dieses Ereignis eine Gültigkeitsdauer zugeordnet, es ist damit in das Fenster aufgenommen worden. Läuft die Gültigkeitsdauer ab, so wird das Ereignis aus dem Fenster entfernt. Tabelle 3 soll die Funktion der einfachen Zeitfenster in Esper verdeutlichen. Dabei wird sich auf den zuvor eingeführten Ereignisstrom und die letzte Anfrage bezogen.

Table 3: Fenster ohne Verschiebefaktor

Fenster	2 Sek.	4 Sek.	6 Sek.	callback
1			**141**,65	141
2		**141**,65	**142**,64	141.5
3	**141**,65	**142**,62	**146**,63	143
4	**142**,62	**146**,63		144
5	**142**,62	**146**,63	**147**,64	145
6	**146**,63	**147**,64		146.5
7	**147**,64			147

Da dieses Vorgehen wenig intuitiv erscheint, existieren in *EPL* auch Längen- und Zeitfenster mit einem fixen Verschiebefaktor. Die sogenannten `batch`-Fenster sind *Tumbling Windows*, deren Verschiebefaktor der Länge des jeweiligen Fensters entspricht. Die gleiche Anfrage mithilfe eines `batch`-Fensters verhält sich folgendermaßen.

```
select avg(price) from StockTick.win:time_batch(6 sec)
where symbol='IBM'
```

Table 4: `batch`-Fenster

Fenster	6 Sek.	callback
1	65, **141**, **142**, 62, **146**, 63	143
2	64,**147**	147

Anfragen mithilfe von einfachen Längenfenstern lassen sich ebenso leicht definieren und verhalten sich genauso wie eine Warteschlange. Es werden nur Ereignisse innerhalb des Längenfensters betrachtet, die dem Filter hinter der `where`-Anweisung entsprechen. Dabei werden durch Esper immer dann Aktionen ausgelöst, die als sogenannte *callbacks* an den *Listener* gesendet werden, wenn ein relevantes Ereignis das Fenster betritt oder verlässt. Die folgende Tabelle 5 demonstriert dieses Vorgehen.

```
select avg(price) from StockTick.win:length(4)
where symbol='IBM'
```

Table 5: Längenfenster

Fenster	1	2	3	4	callback
1	65				-
2	**141**	65			141
3	**142**	**141**	65		141.5
4	62	**142**	**141**	65	-
5	**146**	62	**142**	**141**	143
6	63	**146**	62	**142**	144
7	64	63	**146**	62	146
8	**147**	64	63	**146**	146.5

Aufeinanderfolgende `batch`-Längenfenster überschneiden sich hingegen nicht.

```
select avg(price) from StockTick.win:length_batch(4)
where symbol='IBM'
```

Table 6: `batch`-Längenfenster

Fenster	1	2	3	4	callback
1	65	**141**	**142**	62	141.5
2	**146**	63	64	**147**	146.5

Werden mehrere Bedingungen an ein Ereignis gestellt, können diese mithilfe aussagenlogischer Operatoren beschrieben werden. Die sich anschließende Anfrage ermittelt aus jeweils vier Datensätzen diejenige Kursaktualisierung, die IBM betrifft und den Kurs unter einen Wert von 144 fallen lässt.

```
select * from StockTick().win:length_batch(4)
where symbol='IBM' and price < 144
```

Um **ereignistypbasierte Muster** in *EPL* zu verwenden, wie sie in Sektion 2.2.1 erwähnt wurden, existiert das Schlüsselwort `pattern[]`. Mit dessen Hilfe können Ereignisse auf einem endlosen Datenstrom detektiert werden, ohne dass Fenster eingesetzt werden. Als Grundlage für die folgenden Beispiele soll der folgende, angepasste Ereignisstrom dienen. Dem Ereignisstrom wurde ein neuer Ereignistyp namens `NewsTick` mit der Definition:
`create schema NewsTick(symbol string, message string);` hinzugefügt.

```
StockTick={symbol='YHOO', price=65}
StockTick={symbol='IBM', price=141}
t=t.plus(2 seconds)
StockTick={symbol='IBM', price=142}
StockTick={symbol='YHOO', price=62}
t=t.plus(2 seconds)
StockTick={symbol='IBM', price=146}
NewsTick={symbol="IBM",message="good"}
StockTick={symbol='YHOO', price=63}
t=t.plus(2 seconds)
StockTick={symbol='YHOO', price=64}
StockTick={symbol='IBM', price=147}
t=t.plus(6 seconds)
StockTick={symbol='YHOO', price=71}
StockTick={symbol='IBM', price=150}
NewsTick={symbol="IBM",message="bad"}
t=t.plus(6 seconds)
StockTick={symbol='YHOO', price=77}
StockTick={symbol='IBM', price=107}
```

Um mit dem `pattern[]`-Schlüsselwort auf Ereignisströmen zu operieren, muss zuvor dessen Mechanismus zur Ereignisauswahl geklärt werden. Im Gegensatz zu Fenstern, wo aus einer unendlichen Menge eine klar über die Zeit oder Mächtigkeit definierte, endliche Untermenge betrachtet wird, kann die betrachtete Menge mithilfe des `pattern[]`-Mechanismus unterschiedlich groß in Bezug auf Zeit und Mächtigkeit sein. Die Handlungsweise hängt stark von der Anfrage ab und wird mit dem Begriff *Event Consumption* belegt. Die verschiedenen Verfahren wurden in [2] ausführlich vorgestellt und in [5] in Zusammenhang mit Esper gebracht.
Betrachtet wird eine ereignistypbasierte Regel, welche zwei Ereignistypen mithilfe des schon beschriebenen Sequenzoperators in Kontext setzt (`pattern[A->B]`). Ereignistyp A wird dann als Initiator und B als Detektor bezeichnet. Ein einfaches Beispiel, angewendet auf den neu eingeführten Ereignisstrom, lautet wie folgt. (Dabei soll die Yahoo-Kurskorrektur gefunden werden, die zeitlich auf eine IBM-Kurskorrektur folgt.)

```
select b from pattern[a=StockTick(symbol='IBM') ->
b=StockTick(symbol='YHOO')];
```

Das einzige Ergebnis dieser Anfrage ist das Detektorereignis `StockTick={symbol='YHOO', price=62}`. Das Initiatorereignis ist die erste auftretende IBM-Kurskorrektur `StockTick=symbol='IBM', price=141`. Wird ein Initiatorereignis gefunden, startet Esper einen dedizierten *Event Processing Agent (EPA)* (siehe [3]), der alle eintreffenden Ereignisse in zeitlicher Reihenfolge analysiert. Trifft der *EPA* auf das Detektorereignis, wurde das Muster erkannt und, der angefragte Datensatz wird als Rückgabe bereit gestellt. Auch andere logische Operatoren dürfen in diesen Anfragen Verwendung finden. Wird statt des Sequenzoperators `->` die Konjunktion `and` verwendet, ändert sich das Detektorereignis zu `StockTick={symbol='YHOO', price=65}` aufgrund der Kommutativität dieses Operators.
Aus dieser Darstellung heraus ist erkennbar, dass in diesem Beispiel lediglich ein *EPA* gestartet wurde. Zudem geht aus dem Ereignisstrom hervor, dass dieses Muster noch auf weitere Ereignisse zutreffen könnte. Um Esper anzuweisen, zusätzliche *EPA* zu starten, muss der Operator `every` eingesetzt werden. Die folgende Anfrage verdeutlicht exemplarisch die Anwendung des *Unrestricted Consumption Mode* in EPL.

```
select a.price,b,price from
pattern[every a=StockTick(symbol='IBM') ->
every b=StockTick(symbol='YHOO')];
```

Im *Unrestricted Consumption Mode* wird für jedes Initiatorereignis ein eigener *EPA* gestartet, zudem wird jedes mögliche Detektorereignis in die Ergebnismenge einbezogen. Die Ergebnismenge für diese Anfrage ist in Tabelle 7 vereinfacht als Tupel der Ereignisparameter `price` (a_n, b_n) dargestellt.

Table 7: *Unrestricted Consumption Mode*

a_n	b_1	b_2	b_3	b_4	b_5
141	62	63	64	71	77
142	62	63	64	71	77
146		63	64	71	77
147				71	77
150					77

Das sich anknüpfende Beispiel demonstriert den *Continous Consumption Mode*. In diesem Modus wird ebenfalls für jedes Initiatorereignis ein eigener *EPA* gestartet, allerdings terminiert dieser beim ersten Entdecken eines Detektorereignisses, was die Ergebnismenge stark einschränkt, wie in Tabelle 8 gezeigt.

```
select a.price,b.price from
pattern[every a=StockTick(symbol='IBM') ->
b=StockTick(symbol='YHOO')];
```

Table 8: *Continous Consumption Mode*

a_n	b_1	b_2	b_3	b_4	b_5
141	62				
142	62				
146		63			
147				71	
150					77

Sehr ähnlich ist auch der sich anschließende Modus, bei dem der Operator `every` vor einem Detektorereignis anstatt vor dem Initiatorereignis platziert wird. Es wird daher lediglich ein *EPA* gestartet; dieser aggregiert alle weiteren Detektorereignisse, wie in Tabelle 9 abgebildet. Er wird im Folgenden *Continous Detection Mode* genannt.

Table 9: *Continous Detection Mode*

a_1	b_1	b_2	b_3	b_4	b_5
141	62	63	64	71	77

Der letztmögliche, durch den Operator `every`, darstellbare Modus eignet sich für Mustererkennungen von hochfrequent eintretenden Ereignissen, welche auch mehrfach vorkommen können. Durch das erste Initiatorereignis wird eine *EPA*-Instanz gestartet. Trifft diese Instanz auf ein Detektorereignis, wird dieses als *callback* an den *Listener* gesendet und die Instanz beendet. Im Anschluss wird nach einem weiteren Initiatorereignis gesucht und eine neue *EPA*-Instanz gestartet. Trifft diese auf ein weiteres mögliches Initiatorereignis, ohne vorher auf ein Detektorereignis gestoßen zu sein, wird dieses Initiatorereignis überlesen und erst mit dem Auffinden eines Detektorereignisses beendet. Dieser Vorgang führt bei nachfolgender Anfrage zu dem Ergebnis in Tabelle 10.

```
select a.price,b.price from
pattern[every (a=StockTick(symbol='IBM') ->
b=StockTick(symbol='YHOO'))];
```

Table 10: *Regular Consumption Mode*

a_1	b_1	b_2	b_3	b_4	b_5
141	62				
142					
146		63			
147				71	
150					77

Alle zuvor beschriebenen Beispiele können auch mit Fenstern kombiniert werden. Mit deren Hilfe lassen sich zum Beispiel Untersuchungszeiträume festlegen. Auch der Speicherbedarf kann durch eine Zeit- oder Längenlimitierung begrenzt werden. Die sich anschließende Anfrage soll als komplexeres Beispiel dienen. Es soll untersucht werden, ob eine negative Nachrichtenmeldung den Aktienkurs der IBM-Aktie nachteilig beeinflusst.

```
select c from pattern[a=StockTick(symbol='IBM') ->
b=NewsTick(symbol='IBM' and message='bad') ->
c=StockTick(symbol='IBM' and
price < a.price)].win:time_batch(5 min);
```

In diesem Beispiel wird neben einer kontextabhängigen Bedingung (`price < a.price`), welche den Preis des zweiten Detektorereignisses mit dem Preis des Initiatorereignisses vergleicht, auch ein Zeitfenster eingesetzt, welches den zu betrachtenden Zeitraum auf 5 Minuten einschränkt.

Um **komplexe Ereignisse** zu erzeugen, ist es notwendig, neue Ereignisse in den gleichen oder einem anderen Ereignisstrom einzufügen. Für diesen Zweck bietet *EPL* die `insert into`-Anweisung an. Diese benötigt als Argument einen Ereignistyp und optional die verwendeten Ereignisattribute diesen Typs. Dazu wird ein neuer Ereignistyp eingeführt: `create schema TargetEvent(symbol string, average double);`.
Die sich anschließende Anfrage verwendet zusätzlich zur `intert into`-Anweisung auch noch sogenannte Gruppenfenster. Durch diese Fenster wird die Anfrage deutlich übersichtlicher; für jede Bedingung, die dem Schlüsselwort `groupwin(CONDITION)` mitgeteilt wird, wird eine separate Längen- oder Zeitfensterinstanz gestartet.

```
insert into TargetEvent(symbol, average)
select symbol, avg(price)
from StockTick#groupwin(symbol)#length_batch(6);
```

Es wird jeweils der Durchschnittspreis der letzten sechs, zu einer Firma gehörenden Aktie berechnet und als komplexes Ereignis vom Ereignistyp `TargetEvent` in den Strom eingefügt. Diese komplexen Ereignisse können ebenfalls wieder angefragt und zur weiteren Analyse genutzt werden.

```
select * from TargetEvent();
```

Die Ergebnismenge dieser Anfrage lautet:

```
{symbol='YHOO', average=67.0}
{symbol='IBM', average=138.83333333333334}
```

2.5 SQL-Erweiterungen in EPL

Neben den reinen EPL-basierenden Ereignismustern bietet Esper auch die Möglichkeit, Muster mithilfe des SQL-*Match-Recognize*-Mechanismus zu beschreiben. *Match-Recognize* ist als *Row-Pattern-Recognize* seit 2016 Bestandteil des SQL-Standards [6]. Im Gegenteil zur *truth-value-based-evaluation* von EPL verwendet *Match-Recognize* einen *sequence-based*-Ansatz. Bei der *truth-value-based-evaluation* reicht bereits ein Ereignis, um die Bedingungen des folgenden Ereignismusters zu erfüllen:

```
select a,b from pattern[a=StockTick(price<100)
and b=StockTick(price>20)];
```

Um das Muster zu erfüllen, ist intuitiv ein Ereignis `a`, mit einem Preis von unter 100, und ein Ereignis `b`, mit einem Preis größer 20, zu erwarten. Tatsächlich erfüllt das Ereignis `symbol='YHOO', price=65` das EPL-Ereignismuster. Der *Match-Recognize*-Mechanismus benötigt zur Erfüllung eines äquivalenten Ereignismusters hingegen zwei Ereignisse in vorgegebener Reihenfolge. Das nachfolgende Beispiel zeigt ein solches Ereignismuster.

```
select * from StockTick
match_recognize (
partition by symbol
measures A.price as a_price, B.price as b_price
pattern (A B)
define
A as A.price < 100,
B as B.price > 20 )
```

Ohne präzise auf die *Match-Recognize*-Syntax einzugehen, sind doch einige funktionale Unterschiede wichtig. Der größte Unterschied ist offensichtlich der Aufbau des eigentlichen Ereignismusters (`pattern`). Während EPL-Operatoren der Aussagenlogik spezielle Operatoren zur Verwaltung von Fenstern und ankunftsabhängige Mechanismen wie den Sequenzoperator einsetzen, werden Ereignismuster bei *Match-Recognize* durch reguläre Ausdrücke beschrieben. EPL ist in der Lage, in einem Ereignismuster auf mehrere Ereignistypen zu reagieren, während die SQL-Erweiterung nur einen Ereignistyp als Eingabe erlaubt. In EPL ist es zudem möglich, die Abwesenheit von Ereignissen mithilfe des Negationsoperators festzustellen. Aufgrund des Fehlens eines ähnlichen Operators in der *Match-Recognize*-Erweiterung von Esper ist die Suche nach ausbleibenden Ereignissen schwieriger. Tabelle 11 gibt einen Überblick über die Operatoren, um reguläre Ausdrücke mittels *Match-Recognize* zu formulieren.

Table 11: ReGex Operatoren

Operator	Beschreibung	Beispiel
()	Gruppierung	(A B)
ohne	Reihung (B folgt A)	A B
|	Alternative	A | B
Quantifier		
$*$	kein oder mehrfaches Auftreten	A*
$+$	ein oder mehrfaches Auftreten	A+
?	kein oder einmaliges Auftreten	A?
$\{m, n\}$	zwischen n- und m-malig	A3,4
$\{n, \}$	n-malig oder mehr	A5,
$\{n\}$	genau n-maliges Auftreten	A5
$\{, m\}$	kein bis m-maliges Auftreten	A,7

3. AUSBLICK

Die vorgestellten Anfragen stellen die Grundlage zur Formulierung komplexer Ereignismuster dar. Welche Strategie zur Ereigniserkennung gewählt wird, hängt ganz vom Szenario ab, in dem das Ereignismuster zum Einsatz kommen soll. Aus diesem Grund bietet die Sprache *EPL* noch eine Vielzahl weiterer Möglichkeiten, um Ereignismuster zu formulieren. Zum Beispiel kann durch Auslagerung von Kontextinformationen eine Anfrage übersichtlicher gestaltet werden, die Limitierung der Ausgabe kann die Leistung einer Anfrage verbessern und es können Unterabfragen verwendet werden. Zudem existieren noch weitere Fenstertypen.
Bevor ein komplexeres Ereignismuster einsetzt wird, ist es erforderlich, das Muster einer Leistungsbetrachtung zu unterziehen. In [1] existiert ein ganzes Kapitel zu diesem Thema. Laut [4] eignet sich *CEP* auch zur Visualisierung von Daten, ein interessanter Ansatz könnte der in [8] beschriebene sein.

APPENDIX

A. REFERENCES

[1] http://esper.espertech.com/release-8.0.0/reference-esper/pdf/esper_reference.pdf. *EsperTech Inc.*, 2018.

[2] R. Adaikkalavan and S. Chakravarthy. Seamless event and data stream processing: Reconciling windows and consumption modes. In *International Conference on Database Systems for Advanced Applications*, pages 341–356. Springer, 2011.

[3] R. Bruns and J. Dunkel. *Complex event processing: komplexe Analyse von massiven Datenströmen mit CEP*. Springer-Verlag, 2015.

[4] M. Eckert and F. Bry. Complex event processing (cep). *Informatik-Spektrum*, 32(2):163–167, 2009.

[5] U. Hedtstück. *Complex Event Processing: Verarbeitung von Ereignismustern in Datenströmen*. Springer-Verlag, 2017.

[6] ISO/IEC TR19075-5 - Row Pattern Recognizion in SQL. standard, International Organization for Standardization, 2016.

[7] D. Luckham. Event processing glossary-version 2.0, http://complexevents.com/wp-content/uploads/2011/08/EPTS_Event_Processing_Glossary_v2.pdf. *Event Processing Technical Society*, 2011.

[8] R. T. Perry, C. Kutay, and F. Rabhi. Using complex events to represent domain concepts in graphs. In *Information Science and Applications*, pages 303–311. Springer, 2015.

[9] T. Surdilovic. https://www.slideshare.net/slideshow/embed_code/key/bjwF5BbwNY8U0u. *RedHat via SlideShare*, 2011.

[10] K. Vidačović. Marktübersicht real-time monitoring software - event processing tools im Überblick. *Fraunhofer-Institut für Arbeitswirtschaft und Organisation IAO*, 2010.