Entwicklung einer Financial-Trading-Software mit der CEP-Engine Triceps

Bachelorarbeit im Studiengang Angewandte Informatik an der Fakultät IV – Wirtschaft und Informatik Hochschule Hannover

Christian Wolf

August, 2013

Verfasser
Christian Wolf
Matrikelnummer 1114750
Lenbachstraße 28
30655 Hannover

Erstprüfer
Herr Prof. Dr. rer. nat. Jürgen Dunkel
Hochschule Hannover
Fakultät IV
Abteilung Informatik
Ricklinger Stadtweg 120
30459 Hannover

Zweitprüfer
Prof. Dr. Ralf Bruns
Hochschule Hannover
Fakultät IV
Abteilung Informatik
Ricklinger Stadtweg 120
30459 Hannover

Selbstständigkeitserklärung

Hiermit erkläre ich an Eides Statt, dass ich die eingereicht Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quelle und Hilfsmittel nicht benutzt und die benutzten Werke wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Ort, Datum Unterschrift

Inhaltsverzeichnis

1 Einleitung **4**

2 EDA mit CEP als zentraler Komponente **6**

3 Vergleich von Esper mit Triceps **8**

3.1 Vorstellung von Esper und Triceps 8
3.2 Ereignisse 9
3.3 EPAs und Ereignisregeln 11
3.4 Filtern von Ereignissen 14
3.5 Sliding-Windows 15
3.6 Aggregation 17
3.7 Joins 20
3.8 Ausblick auf Triceps 2.0 22
3.9 Fazit 23

4 Implementierung einer Financial Trading-Software mit Triceps **25**

4.1 Einführung 25
4.2 Börsenindikatoren (Berechnung und Interpretation) 25
4.2.1 Trendbestimmungsindikatoren 25
4.2.2 Trendfolgeindikatoren 26
4.2.3 Oszillatoren 27
4.3 Finanzdaten (Rest-API von Yahoo-Finance) 30
4.4 Financial Trading mit Triceps 31
4.4.1 Generierung von Ereignissen (package getStockData.pm) 32
4.4.2 Table Definition(package processing.pm) 33
4.4.3 Implementierung der Aggregatsfunktionen 33
4.4.4 GUI (package GUI.pm und checkEvents.pm) 36

5 Fazit **40**

6 Anhang **42**

6.1 Minimalbeispiel 42
6.2 Financial Trading-Software 45
6.3 Perl-Guide 47

1 Einleitung

Als Financial-Trading bezeichnet man das Handeln mit Wertpapieren an der Börse. Um diesen Handel zu unterstützen, werden die Trader (Händler) von einer Vielzahl von Anwendungen unterstützt, um z. B.

- aktuelle Börsenkurse zu vergleichen,
- eine Trendbestimmung von einzelnen Kursen vorzunehmen oder auch
- ältere Kursverläufe zu nutzen, um auf zukünftige zu schließen.

Die verwendete Software muss ununterbrochen eine große Anzahl von neu auftretenden Ereignissen analysieren. Um diesen großen Strom von Informationen zu verarbeiten und nur bei relevanten Ereignissen Aktionen beliebiger Art auszulösen, wird die *Event-Driven Architecture* (EDA) verwendet, und zwar in Verbindung mit einem *Complex Event Processing* (CEP) *Engine*.

In den letzten Jahren wurden nun ständig neue CEP-Engines angeboten. Neben kostenpflichtigen Produkten wie beispielsweise Sybase von SAP [1] gibt es inzwischen auch eine Reihe kostenloser Alternativen wie Esper [8] und Triceps [6].

Ziel aller dieser Engines ist eine schnelle Verarbeitung dieser großen Datenmengen und die Bereitstellung einfacher Mittel zur Deklaration von *Ereignisregeln*; dafür stellen diese Engines eine *Event Processing Language* (EPL) bereit. Mangels etablierter Standards verfügt jede Engine über eine eigene EPL mit unterschiedlicher Syntax und Mächtigkeit.

Ein Vergleich von zwei unterschiedlichen EPLs stellt den ersten Teil dieser Bachelorarbeit dar. Verglichen werden die zwei CEP-Engines Esper und Triceps. Beide verfügen über eine umfangreiche EPL, um Ereignisregeln zu erstellen, ihre Ansätze zur Erstellung der Regeln sind aber grundlegend verschieden. Die EPL von Esper verwendet für die Erstellung der Regeln eine Syntax, die stark an die *Structured Query Language* (SQL) angelehnt ist. Regeln werden dabei in der bekannten „`SELECT ... FROM ... WHERE ...`“-Form erstellt.

Im Gegensatz dazu steht die EPL von Triceps. Anstelle einer dedizierten „Anwendungssprache“ (*domain specific language*) für die Erstellung der Regeln, gibt Triceps dem Programmierer mit einer Programmierschnittstelle (API) die Mittel zur Hand, mit denen die Regeln auf einfache Art prozedural definiert werden können. Prozedural bedeutet in diesem Zusammenhang, das die Entwicklung der Ereignisregeln mit Hilfe von Methoden, Kontrollstrukturen und Variablen erfolgt. Wird nachfolgend in dieser Arbeit von prozeduraler Definition der Regeln gesprochen, bezieht sich dies immer auf diese Definition. Diese unterschiedlichen Herangehensweisen bei der Regelerstellung werden anhand eines Minimalbeispiels detailliert verglichen und erläutert.

Im zweiten Teil der Arbeit wird die Implementierung einer Financial-Trading-Software unter Verwendung von Triceps gezeigt. Das Event-Processing von Börsendaten ist ein Paradebeispiel aus der CEP-Welt, das bereits häufig in der entsprechenden Literatur Verwendung fand und sich daher gut für eine Beispielimplementierung eignet.

Als Grundlage für die Ereignisregeln wurden sieben Börsenindikatoren ausgewählt, die auch häufig in kommerziellen Softwaresystemen zum Einsatz kommen, um den aktuellen Verlauf von Aktienkurse zu deuten und Hinweise auf den möglichen zukünftigen Verlauf zu erhalten. Ihre Berechnung wird in einem eigenen Abschnitt erläutert und auch, wie die Ergebnisse zu deuten sind. Anschließend wird eine Möglichkeit aufgezeigt, Aktienkurse von Yahoo-Finance über das REST-API auszulesen.

Im letzten Teilabschnitt ist dann die Implementierung der Software selbst Gegenstand. Es wird der Aufbau der Software und ihr Ablauf dargestellt. Dabei wird besonders erläutert, wie aus den gewonnen Aktiendaten Ereignisse erstellt und wie diese dann in der Software, unter Verwendung von Triceps, verarbeitet werden, um die Börsenindikatoren zu berechnen. Abschließend wird auf die Interpretation der Indikatoren innerhalb des Programmes eingegangen und eine Möglichkeit aufgezeigt, die Einzelereignisse zu korrelieren, um so ihre Aussagekraft weiter zu steigern. Außerdem wird die Darstellung der Ergebnisse in der GUI beschrieben.

2 EDA mit CEP als zentraler Komponente

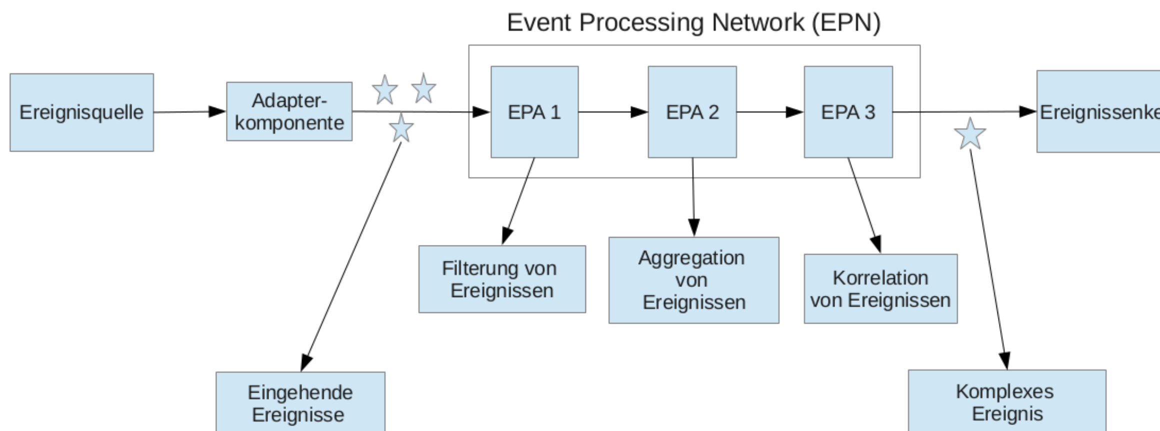

Abbildung 1: Struktur einer Event-Driven-Architektur [9]

Unter einer *Event-Driven Architecture* (EDA) versteht man eine System- oder Anwendungs-Architektur, bei der der Programmablauf nicht fest vorgegeben ist, sondern sich im Gegenteil dynamisch als Reaktion auf die eingehenden Ereignisse realisiert. Auch wenn sich die Engines von ihrer Syntax und allgemein in der Art ihrer Nutzung unterscheiden, sind die grundlegenden Elemente zur Bearbeitung der Daten doch identisch.

Abbildung 1 zeigt den Ablauf anhand der einzelnen Komponenten innerhalb einer EDA. Die Ereignisse gehen von *Ereignisquellen* aus und werden dann über die *Adapterkomponente* [9] in das *Event Processing Network* (EPN) eingespeist. Dieser kontinuierliche Zufluss von neuen Ereignissen wird Ereignisstrom (*event stream*) genannt. Die Ereignisse können dabei aus unterschiedlichen Quellen stammen, es können z. B. Daten sein, die kontinuierlich von einem Sensor eingelesen werden, Börsenticker, Netzwerkdaten oder auch vom Benutzer generierte Ereignisse.

Da CEP-Engines oft nur eine geringe Anzahl von Datenformaten für die Verarbeitung verstehen bzw. ein eigenes Format für die Darstellung von Ereignissen bereitstellen, müssen die Daten häufig vor der Verarbeitung konvertiert werden. Eingehende Daten können dabei jedes beliebige Format besitzen; Sensordaten werden z. B. häufig im XML- oder CSV-Format ausgelesen, aber auch jedes andere Format ist vorstellbar. "Die Adapterkomponente [9] ist für das Auslesen und Konvertieren der Daten in *Ereignisse* verantwortlich. Danach werden die Ereignisse dann dem *Event Processing Network* (EPN) zur Verarbeitung übergeben. Wie zu erkennen ist, besteht ein EPN aus einzelnen *Event Processing Agents* (EPA). Jeder EPA stellt dabei eine Instanz des verwendeten CEP-Engines dar. In ihnen findet unter Verwendung der EPL die eigentliche Verarbeitung der Ereignisse statt."[9]

Die Verarbeitung der Daten in den EPAs erfolgt in mehreren Schritten. Um die Anzahl der Ereignisse, die verarbeitet werden sollen, zu begrenzen, werden zunächst doppelte

Ereignisse (Dubletten) oder solche, die einfach nicht gebraucht werden, aus dem Ereignisstrom herausgefiltert. Dies geschieht in einem eigenen EPA, bei dem entsprechenden Regeln zur *Filterung* registriert sind.

Anschließend beginnt die *Aggregation* und *Korrelation* von Ereignissen. Diese beiden Punkte stellen die zentrale Aufgabe eines CEP-Engines dar. Ziel der Aggregation ist eine Zusammenfassung von einzelnen Ereignissen, die durch Operationen wie `min()`, `max()`, `avg()`, usw. erreicht wird. Die Korrelation ihrerseits identifiziert Ereignisse, die fachlich zusammenhängen und zieht aus der Art des Zusammenhangs Schlüsse auf ein aufgetretenes *komplexes* Ereignis. So hängen (in dem hier betrachteten Zusammenhang) beispielsweise alle Ereignisse, die mit dem Unternehmen Google zu tun haben, fachlich zusammen. Unterstützt wird die Verarbeitung dabei durch Mittel wie *Sliding-Windows*, welche die Speicherung großer Mengen von Ereignissen ermöglichen und *Joins*, die wie in Datenbanken Ereignisse unterschiedlicher Art anhand gleicher Attribute verbinden und dann zur Auswertung bereitstellen können.

Als Resultat der Verarbeitung können Geschäftsprozesse angestoßen, Mitteilungen verschickt oder auch neue, komplexere Ereignisse erzeugt werden. Die neu erzeugten Ereignisse können wiederum in einer EPA verarbeitet oder an anderer Stelle im Programm genutzt werden, einer sogenannten *Ereignissenke*. Eine solche Ereignissenke kann jedes Element sein, das Ereignisse empfangen kann, zum Beispiel ein Prozess der die Daten weiterverarbeitet oder eine Klasse zur Speicherung von Ereignissen in einer Datenbank.

3 Vergleich von Esper mit Triceps

Wie bereits in der Einleitung erwähnt, setzt Esper für seine EPL auf eine SQL-artige Anwendungssprache, während Triceps den prozeduralen Weg über ein API zur Erstellung der Ereignisregeln wählt (als eine *embedded DSL*).

Nachfolgend wird anhand einzelner Elemente, die in beiden Engines genutzt werden, ein Vergleich dieser beiden unterschiedlichen Ansätze angestrebt. Die Tabelle zeigt die bereits im vorigen Abschnitt erläuterten Elemente, die zur Verarbeitung von Ereignissen notwendig sind, ergänzt wurden die Namen der jeweiligen Umsetzung in den CEP-Engines Esper und Triceps.

Element/Funktion	**Triceps**	**Esper**
Ereignis	Row	POJOs, XML-Dateien
EPA	Unit	EPServiceProvider
Filterung	Label + Table	über Querys
Sliding-Windows	Table + Fifoindex	Windows (lokal o. global)
Aggregation	Eigenimplementierung in Funktionen	fertige Funktion
Joins	über Templates	mittels Query

Der Vergleich erfolgt auf Grundlage eines Minimalbeispiels, dem folgende Situation zugrunde liegt:

> Aus Börsendaten, die aus einer beliebigen Quelle stammen, müssen Ereignisse erstellt werden. Neben obligatorischen Attributen wie ID, Zeitstempel und der Quelle, aus der sie stammen, enthalten die Ereignisse den Namen des Unternehmens, zu dem die Daten gehören, und den aktuellen Aktienkurs (price).
>
> Nach der Erstellung werden die Ereignisse zur Verarbeitung einem EPA hinzugefügt. Dabei sollen nur Ereignisse berücksichtigt werden, deren aktueller Aktienkurs $\geq 600\,\$$ ist und die zum Unternehmen Google gehören. Alle anderen Ereignisse sollen aus dem Ereignisstrom herausgefiltert werden. Anschließend wird anhand des Aktienkurses der gleitende Durchschnitt (GD) der letzten 10 eingegangenen Ereignisse berechnet.

Dieses Beispiel wird in den einzelnen Abschnitten nach und nach mit beiden Engines umgesetzt; dabei werden die einzelnen Bestandteile der jeweiligen EPL erläutert und verglichen. Der Quellcode, welcher die Umsetzung des Minimalbeispiels darstellt, ist auch vollständig für beide Engines im Anhang zu finden.

Zuvor werden aber noch beide Engines kurz vorgestellt und ein Blick auf ihre aktuelle Entwicklung geworfen.

3.1 Vorstellung von Esper und Triceps

Esper wird seit 2006 unter dem Dach der Codehouse Foundation entwickelt und hat inzwischen die Version 4.9.0 erreicht.

Neben der Open Source Version gibt es eine kostenpflichtige Enterprise Edition, welche einen erweiterten Funktionsumfang gegenüber der Open-Source-Variante bietet. Außerdem wird eine .NET-Variante angeboten, die unter dem Namen NEsper (aktuell in der Version 4.8.0) zu finden ist.

Beide Varianten verfügen über eine gute Dokumentation und werden auch in mehreren Büchern verwendet, was den Einstieg weiter erleichtert.

Triceps wird von Sergey A. Babkin entwickelt und ist frei (open source) unter der LGPL-Lizenz in Version 3 verfügbar. Die Engine wird in C++ und Perl entwickelt, wobei die Perl-API auf der von C++ aufbaut und für den Programmierer eine weitere Abstraktionsschicht zur einfacheren Nutzung des Engines darstellt. Sergey A. Babkin selbst empfiehlt die Perl-Version, da diese einen höheren Komfort und Umfang gegenüber dem C++-API bietet. Die Engine wird in einem knapp 300seitigen Developer-Guide ausführlich beschrieben. In diesem werden auch häufig vorkommende Problemfälle mit CEP-Engines ausführlich erläutert und entsprechende Lösungen mit Triceps aufgezeigt.

Die Engine wurde 2012 als stabile 1.0 Version veröffentlicht, wird aber weiterhin von S. Babkin gepflegt und erweitert. Über den aktuellen Stand kann man sich in seinem Blog [5] informieren, in welchem auch schon große Teile des Developer-Guides (welcher aktuell nur für die Perl-API verfügbar ist) für die C++-API umgeschrieben wurde. Auf den aktuellen Stand der API und was zukünftig noch von dieser zu erwarten ist wird am Ende des Dokumentes noch einmal eingegangen.

3.2 Ereignisse

In Esper können Ereignisse auf mehrere Arten dargestellt werden. Im Rahmen dieses Vergleichs gehe ich auf die zwei wichtigsten ein, einmal die *Plain-Old Java Objects* (POJO) und dann die Möglichkeit, Ereignisse per XML darzustellen. Bei der POJO-Variante werden die Ereignisse in Form von Java-Klassen erstellt. Neben den Attributen müssen auch Getter-Methoden für den Zugriff auf die Attribute implementiert werden. Diese müssen im bekannten Java-Bean-Style erstellt werden. Wie in Java üblich, sind die so erstellten Klassen auch in der Lage, von Oberklassen zu erben oder Interfaces zu implementieren.

```
1 public class SuperType {
2   private int id;
3   private String date;
4   private String source;
5
6   public SuperType(...) {
7     this.id = id;
8     ....
9   }
10
11   public int getId() {
12     return id;  }
13
14   public String getDate() {
15     return date; }
```

```
16
17   public String getSource() {
18     return  source; }
19 }
```

Das Beispiel zeigt eine Superklasse, die diejenigen Elemente enthält die alle Unterklassen aufweisen sollen. Dies wären ID, Zeitstempel und Herkunft des Ereignisses. Die davon abgeleiteten Klassen enthalten dann nur noch die ereignisspezifischen Attribute. Bezogen auf das Minimalbeispiel, sieht das vollständige Ereignis unter Verwendung der zuvor erstellten Oberklasse so aus:

```
1 public class StockEvent extends SuperType {
2   private String name;
3   private float price;
4
5   public SuperType(...) { ...}
6
7   public String getName() {...}
8   public float getPrice() {...}
9 }
```

Die Erstellung eines Ereignisses erfolgt wie gewohnt über den Konstruktor der Klasse.

```
1   StockEvent event = new StockEvent(  1, "09-12-2013",
2   "yahoo-finance", "Google", 738.13 );
```

Es wird ein Ereignis mit der ID 1 und dem Datum 09.12.2013 erstellt, wobei letzteres zum leichteren Verständnis nicht als Date oder Calendar kodiert wird, sondern ein einfacher String ist. Als Quelle wurde Yahoo-Finance eingetragen, der aktuelle Aktienpreis liegt bei 738.13 $ und gehört zum Unternehmen Google.

Eine weitere Möglichkeit, die kurz angesprochen werden soll, ist die Darstellung von Ereignissen in Form von XML-Daten. Dies ist dann vorteilhaft, wenn die EPAs auch physikalisch verteilt sind. Ereignisse können so direkt z. B. per Sockets zwischen den einzelnen EPAs ausgetauscht werden. Trotz des Textformates ist aufgrund des XML-Formates die Typsicherheit der Attribute gewährleistet. Auch kann, wenn die Ereignisquelle XML exportieren kann, eine direkte Einspeisung der Daten in die Engine erfolgen.

In Triceps gibt es einen eigenen Datentyp, genannt `Row`, in dem die Ereignisse gekapselt werden müssen, um von dem Engine verarbeitet werden zu können. Zur Erstellung der Rows ist ein `RowType` notwendig. Dieser wird wie folgt erstellt:

```
1 $rtSuper = Triceps::RowType->new (
2   id  => "int64",
3   date  => "string",
4   source  => "string",
5 );
```

Die Attribute werden über Key/Value-Paare definiert, wobei Key der Name des Attributes und Value der Datentyp ist.

Ähnlich wie durch Vererbung in Java, kann auch in Triceps ein `RowType` definiert werden, der die Attribute enthält, die alle Ereignisse gemein haben sollen. Das obige Beispiel stellt mit den Attributen `ID`, `date` und `source` den gleichen Grundtyp dar, wie er bereits für Esper gezeigt wurde. Dieser kann anschließend in den weiteren `RowType`-Definitionen mit der Methode `getdef()` ausgelesen und um ereignisspezifische Attribute ergänzt werden:

```
1 $rtStockEvent = Triceps::RowType->new(
2   $rtSuper->getdef(),
3   name  => "string",
4   price => "float64",
5 );
```

Eine `Row` wird dann unter Verwendung des `RowTypes` instantiiert. Dabei werden sowohl die Attribute der Ober- als auch die der Unterklasse angegeben. Die gewählten Daten sind äquivalent zu den in Esper gezeigten.

```
1 $row = $rtStockEvent->makeRowArray(
2   1, "09-12-2013", "yahoo-finance", "Google", 738.13
3 );
```

Die Werte können wie in dem Beispiel einzeln übergeben werden oder in Form eines Arrays, welches alle Werte enthält.

3.3 EPAs und Ereignisregeln

Der EPServiceProvider übernimmt in Esper die Aufgabe eines EPAs. Von ihm können mehreren Instanzen erstellt werden, von denen jede über ein eigenes Event Processing Engine verfügt, bei dem Ereignisregeln registriert werden können. Durch die mehrfache Instantiierung des Providers können auf einfache Weise Event Processing Networks (EPN) aufgebaut und die Regeln über viele EPAs hinweg verteilt werden.

Der nachfolgende Code zeigt die Erstellung eines EPAs in Esper. Zuerst wird der `EPServiceProvider` erstellt. Dazu wird der der eine Instanz des `DefaultProviders` vom `EPServiceProviderManager` angefordert. „Über den erstellten Provider wird ein `EPAdministrator` instantiiert, der dann eine Registrierung von Ereignisregeln im Agenten ermöglicht. Dafür wird die Methode `createEPL(...)` bei der Instantiierung der `EPStatement`-Klasse verwendet. Die `EPStatement`-Klasse ist das Esper-Äquivalent zu einer Ereignisregel“ [9]. Über den Aufruf `createEPL(...)` kann das Ereignis dann in der bereits angesprochenen SQL-Form angegeben werden.

```
1 EPServiceProvider epService = EPServiceProviderManager.getDefaultProvider();
2 EPAdministrator admin = epService.getEPAdministrator();
3
4 EPStatement selectStmt = admin.createEPL(
5 select * from StockEvent where name = 'Google');
```

Diese Regel würde alle Ereignisse des Typs `StockEvent` selektieren, bei denen das Attribut den Namen von Google aufweist.

Ein neues Ereignis kann dem Agenten über den Methodenaufruf `sendEvent()` hinzugefügt werden. Die Übergabe des im vorigen Abschnitts erstellten Ereignisses sieht dann so aus:

```
1 StockEvent event = new StockEvent( 1, "09-12-2013",
2 "yahoo-finance", "Google", 738.13 );
3
4 epService.getEPRuntime().sendEvent(event);
```

Auch in Triceps wird das Konzept der EPAs umgesetzt, die dafür notwendige Komponente heißt `Unit`. Außer einem Namen, im Beispiel `TradeUnit`, muss für eine Erstellung nichts weiter angegeben werden.

```
1 $TradeUnit = Triceps::Unit->new("TradeUnit");
```

In Triceps gibt es kein direktes Äquivalent zu den für Esper gezeigten Ereignisregeln. Stattdessen werden Aufgaben wie Filterung oder Aggregation von `Tables` bzw. `Labels` übernommen, die nachfolgend erklärt werden.

`Tables` dienen in Triceps der Zustandsspeicherung. Genauer gesagt, speichern sie `Rows`, in denen ein Zustand dargestellt werden kann. Bevor eine `Table` erstellt wird muss, ähnlich wie bei den `Rows`, erst ein entsprechender `TableType` erstellt werden. Die Argumente, die für dessen Erstellung benötigt werden, sind ein `RowType` und ein Index. Der angegebene `RowType` bestimmt, welchen Typ die eingehenden `Rows` haben müssen, damit die `Table` sie speichern und verarbeiten kann. Der Index ist für die Speicherung der `Rows` innerhalb der `Table` verantwortlich. Genutzt werden kann ein Hashindex, der über ein Key/Value-Paar definiert wird. Der Schlüssel muss ein Attribut sein, das im angegeben `RowType` vorkommt (oft die `ID`), der Wert ist dann die `Row`, die den entsprechenden Schlüssel aufweist.

Das Codebeispiel zeigt die Erstellung eines `TableTypes`:

```
1 $tableType = Triceps::TableType->new($rtStockEvent)
2 ->addSubIndex("byID", Triceps::IndexType->newHashed(key => [ "id" ])
3 );
4
5 $table = $TradeUnit->makeTable($tableType, "table")
6
7 $table->insert($row);
8 $TradeUnit->makeArrayCall($table->getInputLabel(), @data);
```

Als `RowType` wird der im Abschnitt 3.2 gezeigte `$rtStockEvent`-Typ verwendet. `byID` ist der Name des Hashindexes und kann frei gewählt werden. Als Schlüssel zur Speicherung der `Rows` dient entsprechend die `ID`. Nachdem der `TableType` erstellt ist, kann er beliebig oft instantiiert werden.

Wie in Zeile 5 zu erkennen, geschieht dies mit Hilfe der Unit, die den Befehl `makeTable()` bereitstellt. Alles, was zur Erstellung noch angegeben werden muss, ist der `TableType`, welcher als Vorlage dienen soll und ein Name (hier schlicht: `table`).

Um der Table nach der Erstellung eine Row hinzuzufügen, gibt es zwei Möglichkeiten. Entweder über die Methode `insert(row)`, die in Zeile 7 dargestellt und von der Table

bereitstellt wird. Alternativ können Rows oder die Daten, aus denen eine Row entstehen soll, über das Input-Label der Table hinzugefügt werden. Der Methodenaufruf der Unit in Zeile 8, `makeArrayCall()`, benötigt neben dem Input-Label der Table dafür entweder eine fertige Row oder die Daten in Arrayform, wie es auch im Beispiel gezeigt ist. Die Grundlage für die Erstellung der Row ist der bei der Table-Erstellung angegebene `RowType`, und er wird von der Unit genutzt, um selbständig eine Row aus den übergebenen Daten zu erzeugen. Der Aufruf `getInputLabel()` des Tables sorgt dann dafür, dass die neu erstellte Row der Table hinzugefügt wird. Er wird nachfolgend noch detaillierter erklärt, weshalb an dieser Stelle nicht weiter darauf eingegangen werden soll.

Neben der `insert()`-Methode bietet die Table eine Reihe weiterer Methoden an, die genutzt werden können, um mit den gespeicherten Rows zu arbeiten. Darunter sind `deleteRow()`, `size()`, `find($row)` und Methoden, die eine Iteration über die ganze Table ermöglichen.

Eine weitere wichtige Komponente ist das *Label*. Labels bieten eine einfache Möglichkeit, Ereignisse zwischen verschiedene Tables zu verschicken.

Abbildung 2: Funktionsweise von Labels

Ihre Funktion kann mit einer `goto`-Anweisung verglichen werden. Abbildung 2 zeigt die Nutzung eines Labels. Nehmen wir an, die eingehenden Ereignisse sollen nacheinander in zwei verschiedenen Tables verarbeitet werden, im Bild als Table 1 und Table 2 bezeichnet. Nachdem die Ereignisse in der ersten Table verarbeitet wurden, müssen sie der zweiten hinzugefügt werden. Mit der Methode `insert($row)`, die die Tables bereitstellen, wurde dafür schon eine Möglichkeit gezeigt, wie Rows einer Table hinzugefügt werden können. Eleganter und einfacher ist aber das Verschicken von Ereignissen über Labels. Ist die Verarbeitung einer Row in der ersten Table abgeschlossen, weist das Out-Label, ähnlich einer `goto`-Anweisung, auf die Stelle (im Beispiel Table 2) an der die Row als nächstes verarbeitet werden soll. Triceps sorgt dann selbstständig dafür, dass die Row dem nächsten Table über das Input-Label hinzugefügt wird.

Der Methodenaufruf, um die Labels zweier Tables miteinander zu verbinden, sieht so aus:

```
1 $table1->getOutputLabel()->chain($table2->getInputLabel());
```

Zuerst wird das Label angegeben, aus dem die Rows kommen. Im Beispiel ist es das Output-Label von `table1`. Dann folgt der Aufruf der Methode `chain(...)` welcher aussagt das das Label mit dem Nachfolgenden, im Beispiel das Input-Label von `table2`, verbunden werden soll.

Alternativ kann, mittels Unit, ein Label auch per Hand erstellt und mit anderen Labels oder Tables verkettet werden.

```
1 $label = $tradeUnit->makeLabel($rowType, "name", \&selectSub, @args);
2 $label->chain(table->getInputLabel());
```

Für die Erstellung muss ein `$rowType`, ein Name für das Label und eine Referenz auf eine Funktion angegeben werden. Optional ist auch die Übergabe weiterer beliebiger Argumente möglich.

Die angegebene Funktion, im Beispiel `selectSub` wird jedesmal automatisch aufgerufen, wenn eine Row das Label erreicht. Triceps ruft intern die Methode mit mehreren Argumenten auf:

```
1 selectSub($label, $row, @args)
```

Diese Argumente sind eine Referenz auf das Label selbst, die Row, und die mit angegebenen Argumente. Innerhalb der Funktion kann dann auf die Row zugegriffen und nach Belieben damit gearbeitet werden.

Wie können Ereignisregeln, wie die im Beispiel von Esper gezeigte (`select * from StockEvent where name = 'Google'`), nun in Triceps umgesetzt werden?

Zeile zwei des obigen Codebeispiels zeigt die Verkettung des erstellten Labels mit dem Input-Label einer Table. Die angegeben Funktion `selectSub` kann zur Implementierung der Regeln genutzt werden.

```
1 sub selectSub {
2   $row = $_[1];
3   if ($row->get("name") eq 'Google') {
4      ....
5   }
6 }
```

Dafür wird einfach eine if-Klausel verwendet, die den Namen aus der Row über den `get()`-Befehl ausliest und auf Gleichheit mit „Google“ testet. Mit den Rows, bei denen die Bedingung erfüllt ist, kann dann im if-Block beliebig gearbeitet werden.

3.4 Filtern von Ereignissen

Da in einer CEP-Engine durchaus mehrere zehntausend Ereignisse pro Sekunde eingehen können, ist es sinnvoll, doppelte Ereignisse und solche, die gar nicht gebraucht werden, aus dem Ereignisstrom herauszufiltern, bevor dieser die EPAs zur Verarbeitung erreicht.

In Esper bietet sich die Erstellung eines eigenen Agenten zur Filterung entsprechender Ereignisse an. Innerhalb dieses Agenten können dann Regeln definiert und registriert werden, die alle nicht benötigten Ereignisse herausfiltern.

Die Filterregel aus dem Minimalbeispiel lautete, dass nur Ereignisse, die von Google stammen und deren aktueller Aktienkurs $\geq$ 600 \$ ist, bearbeitet werden sollen. Die Umsetzung sieht so aus:

```
1 select * from StockEvent where name = 'Google' and price >= 600
```

Die `where`-Klausel kann dabei fast alle denkbaren Bedingungen enthalten. Neben den bereits gezeigten Vergleichsoperatoren, ist es beispielsweise auch möglich, komplette Ereignistypen herauszufiltern oder Methoden (bei POJOs) aufzurufen und abhängig von dem Rückgabewert der Methode zu filtern.

Im vorigen Abschnitt wurden die Labels in Triceps vorgestellt und gezeigt, wie sie genutzt werden können, um Ereignisregeln umzusetzen. Dieses Prinzip kann auch zur Filterung von Ereignissen genutzt werden. Bevor die Ereignisse zur Verarbeitung an die Tables geschickt werden, durchlaufen Sie zuerst ein Filterlabel, das alle unerwünschten Ereignisse herausfiltert. Die entsprechenden Filterregeln werden dabei wieder über if/else-Statements umgesetzt. Wie ein Label erzeugt und mit einer Methode verknüpft wird, wurde bereits gezeigt, daher enthält der nachfolgende Code nur noch das if-Statement zur Filterung.

```
1 if ($row->get("name") eq 'Google' && $row->get("price") >= 600) {
2 ...
3 }
```

Neben den if/else-Statements kann natürlich auch jedes andere Konstrukt der Sprache Perl genutzt werden, das sich für eine Filterung eignet (beispielsweise switch/case-Statements). Eine Filterung nach dem Ereignistyp ist hingegen nicht möglich, denn diese geschieht bereits implizit durch das Label, da es nur Ereignisse desjenigen `RowType`s akzeptiert, der bei der Label-Erstellung angegeben wurde.

3.5 Sliding-Windows

Das Konzept der Sliding-Windows wird in jeder CEP-Engine umgesetzt und ist u. a. für die Aggregation von Daten notwendig.

In Esper kann unterschieden werden zwischen dem klassischen Längen-Fenster, welches wie ein FIFO-Puffer arbeitet, einem Zeitfenster, das alle eingehenden Ereignisse über einen gewissen Zeitraum speichert sowie einem Batch-Fenster. Das Batch-Fenster speichert Ereignisse nach Maßgabe einer definierten Anzahl oder eines Zeitraums und leitet diese dann „in einem Rutsch“ weiter, wenn die vorgegebene Anzahl oder der Zeitpunkt erreicht ist.

In Esper können solche Windows auf zwei Wegen erstellt werden. Zum einen gibt es lokale Windows, die sich nur auf das jeweilige Statement beziehen, in dem Sie verwendet werden, zum anderen gibt es die Möglichkeit, *Named Windows* zu deklarieren, die in vielen verschiedenen Statements verwendet werden können.

Erstere werden auf folgende Art erstellt:

```
1 select * from StockEvent.win:length(10);
```

Das Statement erstellt ein Längenfenster, das zehn Ereignisse vom Typ `StockEvent` speichert. Natürlich kann das Fenster auch direkt mit der im Beispiel geforderten Filterklausel erstellt werden:

```
1 select *
2 from StockEvent.win:length(10)
```

```
3 where (name = 'Google' and price >= 600)
```

Die `where`-Klausel kann auch hier beliebige Bedingungen zur Filterung enthalten.

Die zweite Möglichkeit, in Esper ein Fenster zu erstellen, ist das Named Window, welches global in allen Statements genutzt werden kann. Das Statement dafür sieht so aus:

```
1 create window StockEventWindow.win:length(100) as StockEvent
```

Es erstellt ein globales Fenster mit dem Namen `StockEventWindow`, welches 100 Ereignisse des Typs `StockEvent` speichert. Neue Ereignisse können über das aus SQL bekannte `insert into`-Kommando hinzugefügt werden. Dabei werden drei Arten der Ereignisübergabe unterschieden:

```
1 StockEvent event = new StockEvent( 1, "09-12-2013",
2 "yahoo-finance", "Google", 738.13 );
3 insert into StockEventWindow(event)
4 insert into StockEventWindow( 1, "09-12-2013", ...);
5 insert into StockEventWindow (select * from StockEvent)
```

Als Argument des Kommandos `insert into` kann ein fertiges Ereignis wie in Zeile 3 übergeben werden oder aber die einzelnen Attribute direkt (Zeile 4). Alternativ kann als Argument ein `select`-Statement genutzt werden, wie in Zeile 5. In der letzten Version werden alle Ereignisse vom Typ `StockEvent`, die eintreffen, dem Fenster hinzugefügt. Die Statements können auch komplexer sein und z. B. direkt eine `where`-Klausel zur Filterung enthalten.

In Triceps wird das Window-Konzept mit Hilfe von Tables umgesetzt. Dazu wird bei der Erstellung des `TableTypes` der Fifoindex verwendet:

```
1 $tableType = Triceps::TableType->new($rtStockEvent)
2 ->addSubIndex("window", Triceps::IndexType->newFifo(limit => 1010)
3 );
```

Die Erstellung gleicht weitgehend der bereits gezeigten mit dem Hashindex, nur das der Index diesmal `newFifo` ist. Als Argument bekommt der Index die Anzahl der Rows, die gespeichert werden sollen (im Beispiel 10), übergeben.

Die Windows in Triceps lassen sich mit den globalen Windows in Esper zu vergleichen. Überall im Programm, wo ein Zugriff auf die Table möglich ist, kann dieser ebenfalls genutzt werden, um über das Window zu iterieren.

Wie der Name des Indexes schon vermuten lässt, ist ein Window in Triceps das Äquivalent zum Längen-Fenster in Esper. Es gibt in Triceps aktuell kein direktes Gegenstück zum Zeit- bzw. Batch-Fenster. Dies bedeutet freilich nicht, dass entsprechende Gegenstücke nicht zu realisieren wären. Eine Möglichkeit, um in Triceps ein Zeitfenster umzusetzen, wäre es, mehr Rows zu speichern (beispielsweise die doppelte Anzahl von Rows als innerhalb des Zeitintervalls erwartet werden) und bei der Iteration nur diejenigen zu verwenden, die noch im aktuellen Zeitlimit liegen. Am Ende der Iteration könnte dann ein Methodenaufruf erfolgen, um alle Rows, die das Limit überschritten haben, aus der Table zu löschen und den Speicherplatz wieder freizugeben.

Auch ein Batch-Fenster kann in Triceps umgesetzt werden. Anstatt die Rows nach dem Eintreffen direkt über das Output-Label weiterzuleiten, werden diese dann gesammelt, bis die gewünschte Anzahl erreicht ist. Dann können innerhalb einer Schleife alle Rows direkt über den oben gezeigten `insert`-Befehl dem nächsten Table hinzugefügt werden.

3.6 Aggregation

Die Aggregation von Daten stellt in jeder CEP-Engine einen der wichtigsten Punkte dar. Dementsprechend bietet jede Engine umfangreiche Möglichkeiten, um Datenmengen zu verwalten und damit zu arbeiten.

Esper bietet zur Aggregation von Daten eine Reihe bekannter Funktionen wie `min()`, `max()`, `sum()` und `avg()` an. Da zur Aggregation immer eine gewisse Anzahl Ereignisse vorliegen muss, werden die Funktionen oft in Verbindung mit einem Window erstellt. Die im Minimalbeispiel angeführte Query zur Berechnung des Durchschnittspreises einer Aktie von Google sieht folgendermaßen aus:

```
1 select avg(price)
2 from StockEvent(name='Google').win:length(10)
```

Durch diesen Query wird ein Fenster, das zehn Ereignisse speichert, erstellt. Der Durchschnittspreis der Aktie wird anschließend anhand des Attributes `price` gebildet. Neben den Aggregatfunktionen, gibt es in Esper auch das aus SQL bekannte `group by`. Dadurch muss nicht für jedes Unternehmen ein eigener Query angelegt werden, in dem beispielsweise der Durchschnittspreis berechnet wird. Stattdessen kann beispielsweise folgendes Query genutzt werden:

```
1 select symbol, avg(price)
2 from StockEvent.win:length(10)
3 group by symbol
```

Die Berechnung des Durchschnittspreises wird für jedes Aktiengesellschaft, für das Ereignisse des Typs StockEvent eingehen, einzeln berechnet und dann weiter geschickt. Die Gruppierung erfolgt dabei auf Grundlage des Symbols, über das jedes Aktiengesellschaft identifiziert werden kann.

Wie zu erwarten funktioniert die Aggregation in Triceps in Verbindung mit den Tables. Die Funktionsweise ist dabei der mit den Labels ähnlich. Wird ein `TableType` erstellt, kann über den Aufruf `setAggregator(...)` eine beliebige Methode mit angegeben werden. Diese Methode wird jedesmal automatisch aufgerufen und ausgeführt, wenn der Table eine neue Row hinzugefügt worden ist.

```
1 $tableType = Triceps::TableType->new($rtStockEvent)
2 ->addSubIndex("window", Triceps::IndexType->newFifo(limit => 10)
3   ->setAggregator(Triceps::AggregatorType->new(
4     $rtAvgPrice, "aggrAvgPrice", \&computeAverage)
5   )
6 );
```

Im Beispiel wird dem bereits gezeigten `TableType` ein Aggregator hinzugefügt. `$rt-AvgPrice` spezifiert den `RowType`, der für die Erstellung der Ergebnis-Row verwendet wird. `aggrAvgPrice` ist der Name des Aggregators und `computeAverage` eine Referenz auf die entsprechende Methode.

Innerhalb der Methode kann sowohl direkt auf die zuletzt hinzugefügte Row zugegriffen als auch über die komplette Table iteriert werden. Bei der Iteration über die Table können dann Funktionen wie `avg()` oder `sum()` nachgebildet werden. Das nachfolgende Beispiel zeigt die Funktion `computeAverage` und stellt zugleich die Umsetzung des Minimalbeispiels dar:

```
1 sub computeAverage {
2   my $avg = 0;
3   my count = 0;
4   for (my $rhi = $context->begin(); !$rhi->isNull();
5           $rhi = $context->next($rhi)) {
6
7     $avg = $avg + $rhi->getRow()->get("price"),
8     count++;
9   }
10
11  $avg = $avg/$count;
12  ....
13 }
```

Die dargestellte Schleife ermöglicht eine Iteration über alle in der Fifo gespeicherten Rows. `$rhi` kann als ein Zeiger angesehen werden, der am Anfang auf die erste Row zeigt. Die Schleife wird solange durchlaufen, wie es Rows gibt (also 10mal) und mit jedem neuen Durchlauf wird `$rhi` eine Position weiter gesetzt. Mit dem Funktionsaufruf `getRow()` wird die Row ausgelesen, auf die `$rhi` zeigt, und mittels `get(„price“)` das Attribut `price` aus dieser Row. Dieses wird auf die Variable `$avg` aufsummiert, die anschließend durch `$count`, also die Anzahl der Schleifendurchläufe, geteilt wird.

Die große Stärke dieser Art von Aggregation liegt nun aber darin, dass auch komplexere Berechnungen realisiert werden können. So lassen sich die Ergebnisse einer Berechnung mehrfach an verschiedenen Stellen verwenden und innerhalb der Schleife mehrere unterschiedliche Berechnungen für z. B. eine Aktie tätigen. So kann vermieden werden, für jede einzelne Berechnung über die komplette Table iterieren zu müssen.

Nachdem die Berechnungen abgeschlossen sind, kann innerhalb der Funktion eine neue Row mit den Ergebnissen und allen weiteren notwendigen Informationen erstellt werden. Die neue Row kann dann entweder über den `insert`-Befehl einer Table oder über das Outputlabel, das für den Aggregator automatisch erstellt wird, zur nächsten Table weitergeleitet werden.

Da mehr als ein Index pro Table angelegt werden kann, ist eine Kombination von unterschiedlichen Indexen für eine Table möglich. So kann auch die `group-by`-Funktion in Triceps umgesetzt werden. Abbildung 3 zeigt eine beispielhafte Anordnung von Indexen, die die `group-by`-Funktionalität nachbilden. Wie zu erkennen ist, ist der erste Index ein Hashindex mit dem Aktiensymbol als Schlüssel. Der Subindex ist vom Typ Fifo, und

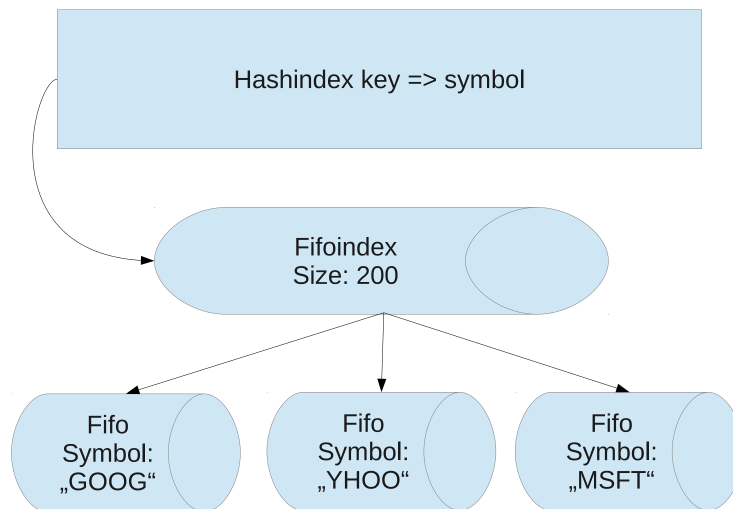

Abbildung 3: Nutzung von mehreren Indexen in Triceps

speichert bis zu 200 Rows. Eine Fifo, die 200 Rows speichert, wird dabei für jedes neue Symbol angelegt, das erkannt wird.

Das Beispiel aus Abbildung 3 würde in Codeform so aussehen:

```
1 $tableType = Triceps::TableType->new($rtStockEvent)
2 ->addSubIndex("bySymbol", Triceps::IndexType->newHashed(key["symbol"])
3     ->addSubIndex("window", Triceps::IndexType->newFifo(limit => 200)
4     )
5   )
6 );
```

Diese Art der Aggregation ist sehr leistungsfähig und ermöglicht die Implementation jeder beliebigen Berechnung. Für einfache Fälle ist aber das manuelle Implementieren einer Aggregatsfunktion, deren Iteration über die Table und die abschließende Erzeugung einer Ergebnis-Row hingegen recht umfangreich, insbesondere wenn, wie im Beispiel aus Abschnitt 3, nur ein einfacher Durchschnittspreis berechnet werden soll. Für diese Fälle bietet Triceps ein fertiges Template, den `SimpleAggregator`. Der Aufbau des Templates ähnelt dabei der SQL-Syntax in Esper.

```
1 Triceps::SimpleAggregator::make(
2   tabType => $tableType,
3   name => "aggrAvgPrice",
4   idxPath => [ "bySymbol", "window" ],
5   result => [
6     symbol => "string", "last", sub {$_[0]->get("symbol");},
7     id => "int32", "last", sub {$_[0]->get("id");},
8     price => "float64", "avg", sub {$_[0]->get("price");},
```

```
 9    ],
10    saveRowTypeTo => \$rtAvgPrice,
11    saveComputeTo => \$compText,
12 );
```

Zur Erstellung müssen ein Name für den Aggregator, ein `TableType` sowie die Namen der `TableIndex`es angegeben werden. Im Beispiel wird der `TableType` verwendet, der zuvor bei der `group-by`-Funktion erläutert wurde.

Als letztes muss noch das Format der Ergebnis-Row festgelegt werden. Die Syntax dafür ist so ähnlich wie die bei den `RowType`s gezeigte. Neben dem Namen und dem Typ des Attributs wird außerdem festgelegt, welche Art Berechnung beim jeweiligen Attribut durchgeführt werden soll. Im Beispiel wird die `ID` und das Symbol der zuletzt hinzugefügten Row (`last`) ausgelesen und als Werte für die Ergebnis-Row verwendet. Beim Attribut Preis wird die `avg`-Funktion zur Durchschnittsberechnung (als Ergebniswert) genutzt. Wird dem Aggregator eine neue Row hinzugefügt, wird automatisch der neue Durchschnittspreis anhand aller gespeicherten Rows berechnet. Eine Row mit den Ergebnissen wird nach jeder Aggregation automatisch erstellt und über das Output-Label der Aggregatfunktion zu einer Table oder einem Label weitergeleitet.

Das Template unterstützt die typischen Aggregationsberechnungen wie `sum`, `min`, `max`, `avg`, `count(*)` sowie `first` und `last` (Wert der ersten oder letzten Row im Table).

3.7 Joins

Als letztes wichtiges Mittel zur Datenverarbeitung möchte ich die Joins vorstellen, die in Esper und Triceps genutzt werden können. Die Funktionsweise ist die gleiche wie in einem Datenbanksystem, nur das anstatt von Zeilen aus Datenbank-Tabellen Ereignisse aus unterschiedlichen Ereignisströmen miteinander verbunden werden. Dabei lassen sich zwei oder mehr Ereignisströme über identische Attribute oder den gleichen Index (Triceps) miteinander verbinden.

Beide Engines unterstützen den Inner-Join, als auch die verschiedenen Outer-Join-Varianten (Left, Right und Full).

```
1 select *
2 from TickEvent.std:unique(symbol) as t, NewsEvent.std:unique(symbol) as n
3 where t.symbol = n.symbol
```

Der Query zeigt die Erstellung eines Joins in Esper. Wie zu erkennen ist, wird ein Join in Esper auf gleiche Art erstellt wie in SQL. Sowohl die Syntax. als auch die Verwendung von Aliasnamen (im Beispiel t und n) mit denen in der Where-Klausel die Ereignisse beschrieben werden ist dabei äquivalent zu SQL. Die Joins beziehen sich dabei auf die eingehenden Ereignisströme des EPA, bei dem Sie registriert sind. Ein Join muss nicht auf zwei Streams beschränkt sein, sondern kann, wie in einem Datenbanksystem, auch mehr als 2 Ereignisströme miteinander kombinieren. Bei einem *Inner-Join* wird nur dann ein neues Ereignis produziert und weitergeleitet, wenn von jedem verwendeten Ereignistyp im `from`-Teil mindestens ein Ereignis eingetroffen ist und die Bedingungen in der `where`-Klausel übereinstimmen.

Sollen auch Ereignisse weiter geleitet werden, bei denen die Bedingung nicht zutrifft bzw. es nicht alle zum Joinen nötigen Ereignisse gibt, muss ein *Outer-Join* genutzt werden. In einem Left-Outer-Join werden dabei alle Ereignisse der linken Seite weitergeleitet, zu denen es keinen Partner gibt. Entsprechend werden bei dem Right-Outer-Join alle der rechten Seite weitergeleitet und wie üblich bei einem Full-Outer-Join, die von der rechten und linken Seite.

In Triceps können Joins mit Hilfe zweier vorgegebener Templates erstellt werden. Das erste Template ist das Lookup-Template.

Das folgende Beispiel zeigt einen *Lookup-Join*. Er kann genutzt werden, um einen Join zwischen einem Label und einer Table zu erstellen. Der Join bekommt die Rows über das Label hinzugefügt. Anhand gleicher Attribute, wird in der Table dann ein Lookup durchgeführt und bei einem Treffer eine neue Row mit den Ergebnissen produziert.

```
1 $join = Triceps::LookupJoin->new(
2 unit => $TradeUnit,
3 name => "join",
4 leftFromLabel => $labelIncome,
5 rightTable => $tAccounts,
6 rightIdxPath => ["lookupSrcExt"],
7 rightFields => [ "internal/acct" ],
8 by => [ "acctSrc" => "source"],
9 isLeft => 1,
10 );
```

Die Situation ist folgende: Über das label `$labelIncome` werden dem Join permanent neue Rows hinzugefügt. Diese stammen aus unterschiedlichen Quellen und sollen in ein gemeinsames internes Format gebracht werden. Für jede Quelle ist in der Table eine Row gespeichert, die die Informationen enthält, die notwendig sind, um die jeweiligen quell-spezifischen Daten in das interne Format abzubilden.

In Zeile eins und zwei müssen wie üblich eine Unit und ein Name für den Join angegeben werden. Die Option `leftFromLabel` in Zeile drei bekommt als Argument das Label zugewiesen, über das die neuen Rows dem Join hinzugefügt werden.

Als nächstes wird die Table angegeben, in der der Lookup durchgeführt werden soll. Dieser wird auf Basis der Felder, die in Zeile 8 in der Option `by` angegeben sind, durchgeführt. Die Felder werden immer als Paare angegeben, ein Feld stammt von der linken Seite (also den neu eingehenden Rows) und ein Feld von der rechten Seite (den Rows, die in der Table gespeichert sind). Von den eingehen Rows wird das Attribut `acctSrc` für die Join-Bedingung genutzt und von den in der Table gespeicherten Rows das Attribut `source`. Gibt es einen Treffer in der Table, wird das Mapping anhand der Optionen `right-`, und `leftFields` durchgeführt. Die Optionen geben Auskunft darüber, welche Attribute übernommen werden sollen. Bei der Option `leftFields` werden die Attribute angegeben, die von den eingehenden Rows übernommen werden sollen, und bei `rightFields` entsprechend die Attribute aus der Row, die in der Table gespeichert ist. Im Beispiel sagt die Angabe `[ïnternal/acct"]`, bei der `rightFields`-Option aus, dass das Feld `internal` übernommen, aber in `acct` umbenannt werden soll. Fehlt eine der beiden Option, wie im Beispiel die `leftFields`, werden alle Felder der linken Seite

übernommen.

Die neu erstellte Row mit den Ergebnissen wird über das Output-Label, über das jeder Join verfügt, weitergeleitet.

Da die linke Seite in diesem Template immer ein Label sein muss und die Rechte ein Table mit festen Werten, werden bei diesem Template nur der Inner-, und der Left-Outer-Join unterstützt.

Das zweite Template ist das `JoinTwo`-Template. Es bietet die Möglichkeit, mehrere Tables anhand ihrer Indexe und Felder miteinander zu joinen. Dieses Template bietet alle in der Einleitung erwähnten Join-Varianten an. Die Funktionsweise ist gleich zu der im Lookup-Join gezeigten, nur das statt eines Labels eine zweite Table angegeben wird. Bei einem Match wird ein neue Row mit den Ergebnissen erzeugt und weitergeleitet. Wie auch in Esper werden bei einem Left-Outer-Join alle Ereignisse der linken Table, zu denen es kein Match gab, und bei einem Right-Outer-Join alle Ereignisse des rechten Tables ohne Match weitergeleitet, schließlich bei einem Full-Outer-Join die von der rechten und linken Table ohne einen Match.

3.8 Ausblick auf Triceps 2.0

Aktuell arbeitet Sergey A. Babkin an der Version 2.0 von Triceps. Unter anderem wird diese Version erweiterte Unterstützung für den Multithreading-Betrieb, verbessertes Scheduling, sowie eine eigene Query-Sprache, genannt *Trivial Query Language* (TQL) bieten.

Auf die TQL wird nachfolgend detaillierter eingegangen, da sie eine interessante Erweiterung von Triceps EPL darstellt.

Trivial Query Language (TQL)

Die TQL ist eine Query-Sprache und von der Idee her ähnlich der oben gezeigten von Esper. Ihr Funktionsumfang beschränkt sich aber auf das Auslesen und Bearbeiten von Rows in einer Table. Die Table-Erstellung und die Aggregation der Daten muss weiterhin in der oben gezeigten prozeduralen Weise erfolgen.

Die TQL-Syntax ist an die von Listen in der Skriptsprache TCL angelehnt und weist folgende Form auf:

```
1 {command option value}
```

Der Query wird durch eine geschweifte Klammer eingeleitet, gefolgt vom Namen des Befehls, das ausgeführt werden soll. Anschließend folgen die Optionen im Name-Value-Format. Zuerst wird der Name der jeweiligen Option angegeben und anschließend die Werte.

Ein einfaches Beispiel könnte so aussehen:

```
1 {read table tWindow}
```

Der Query liest den Inhalt der Table `tSymbol` und gibt die Ergebnisse auf der Konsole aus.

Der eigentlich Clou dieser Querys liegt darin, dass sie *Pipelining*, wie es z. B. in Shellskripten genutzt wird, unterstützen. Querys können dabei sowohl ineinander verschachtelt, als auch nacheinander implementiert werden. Der erste Query weist im Regelfall die oben gezeigte Form auf und liest den Inhalt eines Tables ein. Anschließend können mit weiteren Querys Operationen auf den Rows der Tables durchgeführt werden.

```
1 {read table tWindow} {where istrue {$%price==20}}
```

Das Beispiel liest die Daten von der `tWindow`-Table ein. Dann werden im zweiten Query die Rows selektiert, die beim Attribut Preis den Wert 20 aufweisen.

Wie bereits erwähnt, ist der Funktionsumfang der TQL aktuell noch begrenzt. Neben dem gezeigten `read`- und `where`-Befehl gibt es einen Projektionsbefehl, der einzelne Spalten einer Row ausliest, sowie einen `print`- und `join`-Befehl.

In der aktuellen Version der TQL eignet sich diese besonders gut dazu, mit Hilfe der `where`-Klausel nach Daten in Tables zu suchen. Dadurch, das Querys auch miteinander gejoined werden können, ergibt sich auch ein guter Weg, um Daten auf einfache Weise in Triceps zu filtern. Für die Zukunft ist zu erwarten, dass der Funktionsumfang weiter steigt.

Zu Testzwecken kann ein Snapshot der aktuellen Version von Triceps heruntergeladen werden (Quelle: [6]), welche die neuen Funktionen bereits enthält. Eine ausführliche Erläuterung zur Benutzung gibt es im bereits erwähnten Blog [5] von S. Babkin.

3.9 Fazit

Mit Windows, Joins und Aggregatsfunktionen bieten beide EPLs im wesentlichen die gleichen grundlegenden Möglichkeiten, Ereignisse zu sammeln, zu bearbeiten und zu aggregieren. Die Art und Weise der Erstellung von Ereignisregeln mit den entsprechenden Möglichkeiten ist allerdings sehr unterschiedlich. Wie gezeigt wurde, nutzt Esper die Syntax von SQL, dreht aber das Prinzip um. Anstatt auf bereits vorhandenen Daten mit Querys Abfragen durchzuführen, werden zuerst die Regeln erstellt und dann auf die eingehenden Daten angewendet. Wie in dem Minimalbeispiel gezeigt wurde, ist die Erstellung eines EPAs und der Ereignisregeln selbst unkompliziert und mit wenigen Zeilen Code umgesetzt. Jeder der über Grundlagen-Wissen in SQL verfügt, kann sich also schnell in die EPL einarbeiten und diese sinnvoll nutzen. Wie allerdings in SQL auch, werden die Querys schnell unhandlich und schwer lesbar, wenn sie einen komplexeren Sachverhalt abbilden sollen. Nach Möglichkeit sollte dies durch die Aufteilung in mehrere Querys vermieden werden, da sie das Programm lesbarer macht.

Der Ansatz von Triceps wirkt dagegen anfänglich etwas umständlicher. Ein EPA und die Ereignisse selbst sind auch hier schnell mit wenigen Zeilen Code erstellt. Anders sieht es dagegen aus, wenn Ereignisregeln hinzu kommen sollen. Insbesondere dann, wenn diese auch noch die eingehenden Ereignisse mit Hilfe von Fenstern sammeln und aggregieren sollen. In Esper genügt die Erstellung eines einzelnen Querys, um ein Fenster zu erstellen und die gewünschte Aggregation vorzunehmen. In Triceps muss dagegen zuerst das Konzept der Tables, mit ihren Indexen und Labels verstanden werden. Dazu kommt dann noch die Aggregation über das Template oder die `setAggregator()`-Methode selbst.

Anfänglich ergibt sich dadurch eine im Vergleich zu Esper steilere Lernkurve. Ist das Konzept allerdings erst einmal verstanden, kann es schnell umgesetzt werden, um die gewünschten Ereignisregeln mit den Möglichkeiten der Sprache Perl umzusetzen. Dies zeigt sich auch wieder im Minimalbeispiel selbst, da beide Versionen für dessen Umsetzung in etwa den gleichen Codeumfang benötigen. Vorteilhaft zeigt sich die EPL von Triceps insbesondere, wenn komplexere Berechnungen notwendig sind, da diese direkt in einer Aggregats-, oder Labelfunktion umgesetzt werden können.

Welchen Weg man präferiert, hängt vom persönlichen Geschmack bzw. Vorwissen ab. Jemand der bereits viel mit SQL gearbeitet hat, wird sich schnell in Esper zurechtfinden können, da nicht nur die Syntax sondern auch deren Bedeutung in der Regel äquivalent zu SQL ist. Jemandem, der dagegen wenig bis keine Erfahrung mit SQL gesammelt hat, dürfte der Einstieg in die EPL von Triceps leichter fallen. Nachdem man den internen Ablauf der Engine verstanden hat, können dann nämlich die Regeln auf dem gewohntem prozeduralem Weg erstellt werden.

4 Implementierung einer Financial Trading-Software mit Triceps

4.1 Einführung

Der nachfolgende Abschnitt, stellt den zweiten Teil der Bachelorarbeit dar und gliedert sich wiederum in drei Teile. Im ersten werden die Börsenindikatoren die in der Software umgesetzt worden sind vorgestellt. Dazu wird die Berechnung der Indikatoren gezeigt und erläutert wie die Ergebnisse zu deuten sind.

Anschließend soll ein kleiner Blick auf die Möglichkeit geworfen werden, Börsenkurse über die Yahoo-Finance-API auszulesen.

Im dritten Teil, wird zuerst der Aufbau und Ablauf der Financial Trading-Software grob skizziert, bevor abschließend die einzelnen Packages detailliert beschrieben und beispielhaft die Umsetzung von drei Börsenindikatoren und ihre Auswertung gezeigt wird.

4.2 Börsenindikatoren (Berechnung und Interpretation)

Die in der Software verwendeten Indikatoren können in drei unterschiedliche Gruppen eingeteilt werden:

- solche zur Trendbestimmung,
- Trendfolgeindikatoren
- und schließlich die Oszillatoren.

Die Indikatoren der Trendbestimmung geben den Verlauf der Aktie, meist über einen bestimmten Zeitraum, oder die Stärke eines Trends wieder.

Die Trendfolgeindikatoren versuchen dagegen, den aktuellen Trend eines Kurses aufzuzeigen; genauer gesagt, ob sich eine Aktie gerade im Auf- oder Abwärtstrend befindet.

Oszillatoren sind Werte, die zwischen einem Minimal- und einem Maximalwert hin und her schwanken. Ihre Aussage hängt stark vom eigentlichen Indikator ab und kann nicht allgemein getroffen werden.

Die genaue Interpretation der Werte und die Formeln zu deren Berechnungen unterscheiden sich zum Teil in den einzelnen Quellen. Daher sei an dieser Stelle auf die Quellen [2] und [7] verwiesen, die als Grundlage für die Formeln und die Interpretation ihrer Ergebnisse für diese Arbeit genutzt wurden.

4.2.1 Trendbestimmungsindikatoren

Der *Directional Movement Index* (DMI) zeigt an, ob ein Kursverlauf einen Trend aufzeigt und wenn ja, wie stark dieser ist. Der DMI nimmt dabei Werte zwischen 0 und 100 an. Nimmt der Indikator ab, ist der Trend rückläufig, nimmt er zu steigt der Trend.

Zur Berechnung werden mehrere Komponenten benötigt:

Das *positive directional movement* (+DM) und das *negative directional movement* (-DM). Sie stellen die Differenz zwischen dem aktuellen und dem gestrigen Höchst-, bzw. Tiefstkurs dar. Die Formel lautet entsprechend:

$$+\mathtt{DM} = \text{heutiger Höchstkurs} - \text{gestriger Höchstkurs} \tag{1}$$

$$-\mathtt{DM} = \text{heutiger Tiefstkurs} - \text{gestriger Tiefstkurs} \tag{2}$$

Aus den +DM/-DM werden die *Directional Indicators* (+DI, -DI) berechnet. Dazu werden das +DM und -DM aufsummiert, meistens über einen Zeitraum von 14 oder 18 Tagen, und durch die Summe aller Werte der *True Range* (TR) geteilt.

$$+\mathtt{DI} = \frac{1/n \sum_{i=1}^{n}(+\mathtt{DM})}{\text{True Range}} -\mathtt{DI} = \frac{1/n \sum_{i=1}^{n}(-\mathtt{DM})}{\text{True Range}} \tag{3}$$

$n = 14$ oder 18 Tage

Die TR berechnet sich wie folgt:

$$\begin{aligned} \mathtt{TR} = \max(&\text{heutiger Höchstkurs} - \text{heutiger Tiefstkurs}, \\ &\text{heutiger Höchstkurs} - \text{gestriger Schlusskurs}, \\ &\text{heutiger Tiefstkurs} - \text{gestriger Schlusskurs}) \end{aligned} \tag{4}$$

Anschließend berechnet sich der DMI folgendermaßen:

$$\mathtt{DMI} = \frac{abs((+\mathtt{DI}) - (-\mathtt{DI}))}{(+\mathtt{DI}) + (-\mathtt{D})} \times 100 \tag{5}$$

Der ADX stellt den gleitenden Mittelwert des DMI dar. Er wird zur Bestimmung der Trendstärke verwendet. Dabei deutet ein steigender ADX auf eine Trendphase hin, ein sinkender Wert dagegen auf eine trendlose Phase:

$$\mathtt{ADX} = 1/n \sum_{i=1}^{n}(\mathtt{DMI}_i) \tag{6}$$

- n= Anzahl Tage, die in die Berechnung einfließen sollen. Vielfach 14 oder 18 Tage.
- $\mathtt{DMI}_i$= DMI des jeweiligen Tages.

4.2.2 Trendfolgeindikatoren

Der *gleitende Durchschnitt* (GD) ist der wohl bekannteste aller Börsenindikatoren. Er stellt den Durchschnittskurs einer Aktie über einen vordefinierten Zeitraum dar. Häufig werden hierzu die letzten 200 Tage verwendet. Der GD stellt somit das arithmetische Mittel eines Aktienkurses über einen bestimmten Zeitraum dar.

Die Formel lautet:

$$\text{GD} = \frac{1}{n}\sum_{i=1}^{n}(c_i) \tag{7}$$

n = Anzahl Tage, die in die Berechnung einfließen sollen.
c_i = Schlusskurs des Tages i.

Die Envelopes sind ein weiteres Mitglied der Trendfolgeindikatoren. Sie verschieben den GD um einen definierten Prozentsatz (meistens 2.5, 5 oder 10 Prozent) nach oben und unten. Sie werden deshalb auch oberes und unteres Band genannt und bilden gleichsam einen Kanal, innerhalb dessen sich der GD bewegt. Hinweise zum Handeln entstehen, wenn der GD die obere oder untere Hälfte des Bandes berührt oder verlässt. Berühren oder Überschreiten des unteren Bandes stellt ein Kaufsignal dar, beim oberen Band entsprechend ein Verkaufssignal.

$$\text{oberes Band :}\texttt{GD} + (\texttt{GD} \cdot \text{Prozentsatz}/100) \tag{8}$$

$$\text{unteres Band :}\texttt{GD} - (\texttt{GD} \cdot \text{Prozentsatz}/100) \tag{9}$$

4.2.3 Oszillatoren

Der *Relative-Stärke-Index* kurz RSI gibt an, ob ein Markt *überkauft* oder *überverkauft* ist. Überkauft bedeutet, dass sich der Markt oder hier die Aktie bereits über einen längeren Zeitraum nach oben bewegt hat und deshalb bereits überdurchschnittlich viele Anleger in diese investiert haben. Daraufhin erfolgt oft eine Gegenreaktion, da die Anleger den hohen Preis nutzen und ihre Wertpapiere wieder verkaufen, was einen Kursverlust der Aktie zur Folge hat.

Überverkauft stellt das genaue Gegenteil dar. Der Kurs der Aktie bewegt sich auf einem sehr niedrigen Niveau. Je tiefer der Wert der Aktie sinkt, desto weniger Anleger verkaufen ihre Aktien noch. Hierauf folgt dann oft die Gegenreaktion, dass der Kurs wieder steigt, da die Anleger den niedrigen Preis nutzen und Aktien kaufen. Liegt der ermittelte RSI-Wert über 70 bzw. unter 30 gilt ein Markt demnach als überkauft oder überverkauft.

Berechnung und Beispiel:

$$\text{RSI} = (h/(h+n)) * 100 \tag{10}$$

h = Durchschnitt der Kurssteigerungen der letzten n Börsentage.
n = Durchschnitt der Kursrückgänge der letzten n Börsentage.
Empfohlen wird eine Standardeinstellung von $n = 14$ Tagen.

Beispiel Berechnung eines RSI mit 7 Tagen:

Tag	Datum	Kurs	Steigerung/Rückgang
1	02.06.13	2970	
2	03.06.13	2890	-80
3	04.06.13	2825	-65
4	05.06.13	2845	+20
5	06.06.13	3015	+70
6	07.06.13	3050	+35
7	08.06.13	2980	-70
8	09.06.13	2970	-10

h=(35+70+20)/3 = 41,66
n=(10+70+65+80)/4 = 56,25
RSI = (41,66/(41,66+56,25))*100 = 42,55

Ein weiterer Vertreter der Oszillatoren ist die *Rate of Change* (ROC). Sie oszilliert um einen Mittelwert herum und bestimmt die Stärke eines Marktes. Liegt die ROC im negativen Bereich und steigt nach oben, legt das ein mögliches Ende des Abwärtstrends nahe. Fällt sie dagegen weiter, ist dies als Fortsetzung des Abwärtstrends zu deuten.

Liegt der ROC im positiven Bereiche und steigt weiter nach oben, ist dies als Fortsetzung des Aufwärtstrends zu interpretieren. Fällt er dagegen, kann das auf ein mögliches Ende des Aufwärtstrends hindeuten.

Neben dieser Interpretation kann eine sogenannte neutrale Zone festgelegt werden. Eine mögliche Zone wäre beispielsweise zwischen 90 und 108. Steigt der ROC von unter 90 auf über 90 und betritt somit die Zone, kann dies als Kaufsignal gedeutet werden. Fällt er dagegen von einem Wert $\geq$ 108 in die Zone, kann dies als Verkaufssignal gedeutet werden.

Berechnung:

$$\text{ROC} = (c/c_x) * 100 \tag{11}$$

mit
c = Schlusskurs aktuell und
c_x = Schlusskurs vor x Tagen.
Für x wird eine Standardeinstellung von 12–30 Tagen empfohlen.

Der *Commodity Channel Index* (CCI) bewegt sich ober- bzw. unterhalb des GD und gibt die Abweichung des aktuellen Kurses vom GD wieder. Das Ergebnis seiner Berechnung ist ein Wert, der meist zwischen den sogenannte Signallinien -100 und $+100$ oszilliert.

Kaufsignale entstehen beim CCI immer dann, wenn der errechnete Wert die -100-Signallinie oder die Null-Linie von unten nach oben durchbricht. Umgekehrt entstehen Verkaufssignale, wenn der CCI von oben nach unten durch die $+100$-Signallinie bzw. durch die Null-Linie fällt.

Die Berechnung des CCI erfolgt in mehreren Teilschritten. Zuerst wird der Durchschnittskurs des Tages t berechnet. Dieser setzt sich dabei aus dem Höchst-, Tiefst- und Schlusskurs eines Tages zusammen.

$$X_t = \frac{H + T + S}{3} \tag{12}$$

H = Höchstkurs, T = Tiefstkurs, S = Schlusskurs

Als nächstes wird der GD von X_t über eine bestimmte Periode (oft 20 Tage) ermittelt. Im Falle des CCI wird allerdings nicht die Abkürzung GD verwendet, sondern MA_t.

$$\mathrm{MA}_t = 1/n \sum_{i=1}^{n} (X_t) \tag{13}$$

Anschließend erfolgt die Berechnung der Standardabweichung.

$$SX_t = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (X_t - MA_t)^2} \tag{14}$$

Schließlich berechnet sich dann der CCI folgendermaßen aus dem ermittelten X_t des aktuellen Tages und den ermittelten Werten MA_t und SX_t:

$$\mathrm{CCI} = \frac{X_t - MA_t}{0,015 SX_t} \tag{15}$$

0,015 ist dabei ein fester Faktor, der bewirkt, dass ein Großteil der Indikatorwerte zwischen -100 und +100 liegt.

4.3 Finanzdaten (Rest-API von Yahoo-Finance)

Als Quelle der Börsendaten dient die REST-API von Yahoo-Finance. Die Daten können per GET-Request vom Server geladen werden. Dabei besteht die Möglichkeit, die Daten im CSV-, JSON- oder XML-Format herunterzuladen. Für das Projekt werden die Daten im CSV-Format herunterzuladen, da in Perl die Daten nach dem Download bequem per Methodenaufruf gesplittet und weiterverarbeitet werden können.

Die URL für den Download sieht folgendermaßen aus:

```
'http://download.finance.yahoo.com/d/quotes.csv
?s=YHOO+GOOG+MSFT&f=snl1kj&e=.csv'
```

Der String kann in 4 Teile aufgesplittet werden.

1. http://download.finance.yahoo.com/d/quotes.csv
2. ?s=YHOO+GOOG+MSFT
3. &f=snl1kj
4. &e=.csv

Der 1.Teil ist die Adresse des Servers, der angesprochen werden soll mit dem Hinweis, dass Börsendaten im CSV-Format heruntergeladen werden sollen. Als nächstes werden die Namen, genauer die Symbole angegeben, unter denen die Unternehmen an der Börse gelistet sind. Es ist möglich, gleich für mehrere Unternehmen die Daten herunterzuladen. Dazu werden die Symbole mit einem Plus-Zeichen verkettet. Punkt drei gibt an, welche Daten geladen werden sollen. Jeder Buchstabe nach dem &f= steht dabei für ein Element.

In diesem Beispiel:

- s = Symbol des Unternehmens. Beispielsweise GOOG
- n = Name des Unternehmens.
- l1 = Letzter Preis der Aktie
- k = Höchster Preis der Aktie in den letzten 52 Wochen
- j = Niedrigster Preis der Aktie in den letzten 52 Wochen

Zum Schluss kommt noch der Part, der besagt, dass man die Daten im CSV-Format haben möchte. Neben den genannten Kürzeln für Name, Aktienpreis, etc. gibt es noch viele weitere, die man bei Bedarf abrufen kann. Eine genaue Auflistung der verfügbaren Kürzel findet man in [3].

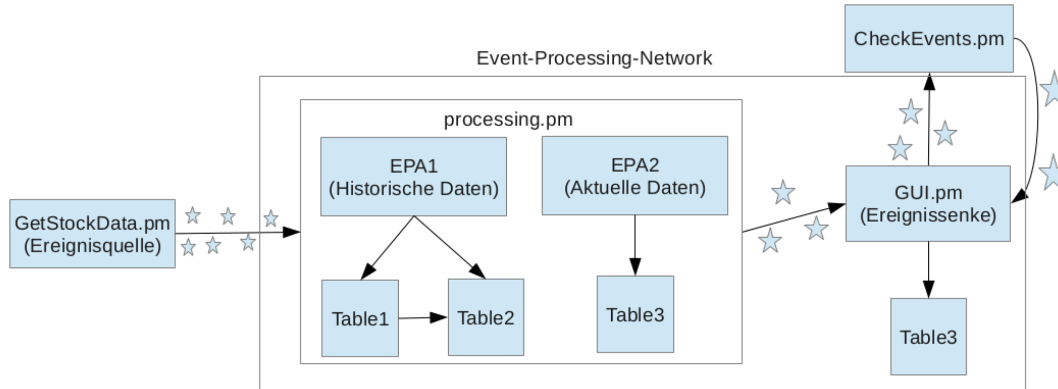

Abbildung 4: Event Processing mit Triceps

4.4 Financial Trading mit Triceps

Zur Erstellung der Software wurde die im Abschnitt 2 vorgestellte ereignisgesteuerte (event-driven) Architektur verwendet.

Abbildung 4 zeigt die Umsetzung der Architektur und den Ablauf der Financial-Trading-Software.

Das Package getStockData repräsentiert die Ereignisquelle. Über eine Queue werden die Ereignisse dem Processing-Package übergeben. Dieses bildet zusammen mit dem GUI-Package das Event-Processing-Network (EPN) der Software ab. Da es zwei unterschiedliche Arten von Ereignissen gibt (historische und aktuelle), werden im Processing-Package zwei Units als EPAs verwendet. Der erste EPA ist dabei für die Verarbeitung der historischen Daten zuständig. Die Verarbeitung ist der Übersichtlichkeit wegen auf zwei Tables aufgeteilt, die beide mit der gleichen Unit erzeugt werden. Der zweite EPA und die zugehörige Table ist für die Verarbeitung der aktuellen Daten verantwortlich.

Die Aggregations-Ergebnisse der historischen Daten, aus dem processing-Package, werden über einen prozeduralen Aufruf dem GUI-Package hinzugefügt. Dort werden die Daten in der Table des dritten EPAs gespeichert. Im letzten Schritt werden die Ereignisse dann dem checkEvents-Package übergeben, wo Sie mit Hilfe definierter Regeln interpretiert werden. Wie zu erkennen ist, gehen die Ergebnisse dieser Interpretation zurück zum GUI-Package, wo sie dem User anschließend in Textform präsentiert werden. Daneben werden die gewonnen Indikatoren auch in Form von Liniendiagrammen dargestellt. Da dies einen permanenten Zugriff auf die Daten in der Table erfordert, wurde der Code für die GUI zusammen mit dem letzten Agent in einem gemeinsamen Package erstellt. Auf diese Weise kann die GUI auf einfache Weise für die Darstellung der Liniendiagrammen über den Table iterieren und die Ergebnisse des checkEvents-Package verarbeiten.

Von den aktuellen Daten wird nur der Kurs und der Unternehmensname in einem Array gespeichert und für einen Ticker verwendet.

Die genauen Aufgaben und der Aufbau, der einzelnen Packages, wird nachfolgend

detailliert erläutert.

4.4.1 Generierung von Ereignissen (package getStockData.pm)

Wie die Börsendaten mit der Yahoo-Finance-API abgerufen werden, wurde im Abschnitt 4.3 detailliert erläutert. Im Package getStockData findet die Umsetzung dieser Abfrage statt. Bei der Abfrage muss zwischen den historischen und aktuellen Daten unterschieden werden, für die es jeweils eine eigene Methode gibt.

Bei den historischen Daten muss für jedes Unternehmen eine eigene Abfrage erfolgen, da es immer nur für ein Unternehmen möglich ist, die Daten herunterzuladen. Die aktuellen Daten können dagegen direkt in einer Anfrage für mehrere Unternehmen geladen werden. Die Verarbeitung ist in beiden Fällen identisch.

Die Antwort des Requests ist eine CSV-Datei, die eine oder mehrere Zeilen enthält. Ihr Inhalt wird in einer Variable gespeichert. Jedes Zeile stellt einen Datensatz dar, der die angeforderten Informationen wie Name, Preis, Volumen, etc. enthält, siehe nachfolgendes Beispiel:

Date,	Open,	High,	Low,	Close,	Volume,	Adj Close
2013-04-30,	24.38,	24.79,	24.36,	24.73,	10091200,	24.73

Die erste Zeile stellt die Namen der Werte dar, die zweite dann die Werte selbst.

Historischen Daten, werden häufig für einen bestimmten Zeitraum angefragt. Jede Zeile stellt dann einen Tag dieses Zeitraums dar und enthält alle angeforderten Werte des jeweiligen Tages. Der Zeitraum und das Unternehmen, für das Daten abgefragt und verarbeitet werden sollen, lässt sich dabei frei über die GUI bestimmen. Bei den aktuellen Daten enthält die CSV-Datei nur dann mehr als eine Zeile, wenn für mehrere Unternehmen Daten angefordert worden sind, dann stellt jede Zeile die Antwort mit den jeweiligen Daten für ein Unternehmen dar.

Anschließend werden die CSV-Daten aus der Variable ausgelesen und geparst. Die einzelnen Daten, werden anschließend in eine Queue zur Weiterverarbeitung gepusht.

Bevor die Daten von Triceps verarbeitet werden können, müssen aus ihnen noch Rows erstellt werden. Dies geschieht im Processing-Package, da die Queue nur mit skalaren Werten arbeiten kann, weshalb eine Übergabe von fertigen Rows nicht möglich ist. Wie zuvor gibt es auch hier wieder für beide Typen eine eigene Methode. Die Daten werden dazu in einer Schleife aus der Queue ausgelesen und bei der Row-Erstellung, um weitere Informationen wie Id und Typ (historisch oder aktuell) ergänzt. Über einen Methodenaufruf werden Sie den Units und damit auch den Tables hinzugefügt, wo sie anschließend verarbeitet werden.

Da die historischen Daten nur aus einer Quelle stammen, und bei dieser noch exakt spezifiziert werden kann, welche Informationen heruntergeladen werden sollen, bedarfs es in dem Programm keiner Filter, um Rows auszusortieren.

Für die aktuellen Daten wird ein Filter genutzt, der im nächsten Abschnitt erläutert wird.

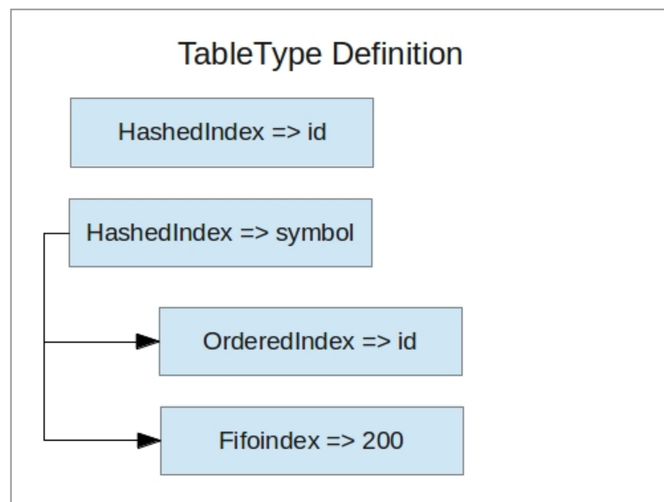

Abbildung 5: Aufbau des TableIndexes

4.4.2 Table Definition(package processing.pm)

Abbildung 5 zeigt den Aufbau des TableIndexes. Dieser ist bei allen genutzten Tables identisch.

Die beiden Hash-Indexe sind parallel angelegt. Der erste ermöglicht ein einfaches Auffinden einzelner Rows anhand ihrer `id`. Mithilfe des Subindexes vom Typ Fifo setzt der zweite Hashindex die zuvor erwähnte `group-by`-Funktion um. Der `OrderedIndex` sorgt innerhalb der Fifo für eine Sortierung der Rows anhand ihrer ID in absteigender Reihenfolge.

Wie schon im vorigen Abschnitt 4.4 erwähnt, findet die Verarbeitung der historischen Rows in zwei Tables statt. Da sie die gleichen Börsenindikatoren berechnen und fachlich zusammenhängen, sind beide Tables derselben Unit zugeteilt. Für jeden Table gibt es einen eigenen `TableType`, da den Tables unterschiedliche Aggregatsfunktionen zugeordnet sind.

Für die aktuellen Daten wird nur eine einzelne Table mit einer Aggregatfunktionen verwendet.

4.4.3 Implementierung der Aggregatsfunktionen

Da der Umfang der Aggregatsfunktionen, recht groß ausgefallen ist, wird beispielhaft nur die Implementierung der Indikatoren GD, ROC und des RSI gezeigt.

Die Berechnung wurde innerhalb des Programms auf zwei Tables und deren Aggregatfunktionen aufgeteilt. Dies spiegelt sich auch im Beispielcode wieder, welcher aus zwei Teilen besteht, die nacheinander erläutert werden.

Im ersten Teil wird der GD sowie für den ROC die Werte c (aktueller Schlusskurs) und c_x (Schlusskurs vor x-Tagen) bestimmt. Anschließend werden die ersten Teilberechnungen für den RSI durchgeführt. Dabei werden die Kursrückgänge bzw. Zunahmen zum vorigen Tag für die letzten 14 Tage vorgenommen.

Im zweiten Teil werden dann die abschließenden Berechnungen für den ROC und RSI durchgeführt. Auf eine erneute Beschreibung der Berechnungen wird an dieser Stelle verzichtet, da diese schon im Abschnitt 4.2 erläutert wurden und bei der Beschreibung des Programmcodes ebenfalls noch einmal beschrieben werden.

Alle im Beispielcode verwendeten Variablen wurden zuvor schon in der Funktion angelegt und initialisiert.

```
1 for (my $rhi = $context->begin(); !$rhi->isNull();
2                 $rhi = $context->next($rhi)) {
3
4   my $currentRow = $rhi->getRow()->copymod();
5
6   $count++;
7   #Aufsummierung des Schlusspreises der Aktie.
8   $sumGD += $currentRow->get("closePrice");
9
10
11  #Auslesen des Schluskurses vor x Tagen (cx) und des
12  #aktuellen c zur Berechnung des ROC.
13  if ($count == $cxDays ||
14  $context->groupSize() < $cxDays && $context->groupSize() == $count) {
15      $cx = $currentRow->get("closePrice");
16  } elsif($count == 1) {
17      $c = $rhi->getRow()->get("closePrice");
18  }
19
20
21  #Berechnung der RSI Daten
22  if ($count == 1) {
23    $rsiLatestTrade = $currentRow->copymod();
24  }
25
26  if ($count >= 2 && $count <= $rsiDays+1) {
27    my $closePriceLast= $rsiLatestTrade->get("closePrice");
28    my $closePriceCurrent = $currentRow->get("closePrice");
29
30    if($closePriceLast >= $closePriceCurrent) {
31      $hRsiSum = $hRsiSum + $closePriceLast-$closePriceCurrent;
32      $hLength++;
33    } else {
34      $nRsiSum = $nRsiSum + $closePriceLast-$closePriceCurrent;
35      $nLength++;
36    }
37    $rsiLatestTrade = $currentRow->copymod();
38  }
39 } #Schleifenende
40
```

```
41 #GD
42 my $gD     = $sumGD / $count;
```

`$rhi` kann als ein Zeiger angesehen werden, der am Anfang auf die erste Row zeigt. Die Schleife wird solange durchlaufen wie es Rows gibt (also höchstens 200mal) und mit jedem neuen Durchlauf wird `$rhi` eine Position weiter gesetzt. Über den Aufruf `$rhi→getRow()` erhält man Zugriff auf die Row, deren Felder man anschließend mit `get(Feldname")` auslesen kann. Da die aktuelle Row an mehreren Stellen innerhalb der Schleife benötigt wird, wird sie am Anfang der Schleife in Zeile 4 über den Methodenaufruf `copymod()` in die Variable `$currentRow` kopiert, welche anschließend zur Nutzung kommt. So muss die Row nicht jedesmal erst mit `getRow()` angefordert werden, bevor die Felder über `get()` ausgelesen werden können. Anschließend werden die einzelnen Berechnungen für die jeweiligen Indikatoren durchgeführt.

In Zeile 6 wird die Variable `$count` hochgezählt, um auf die aktuelle Anzahl der verarbeiteten Rows Zugriff zu haben. Als nächstes wird der gleitende Durchschnitt berechnet. Genauer gesagt, wird in der Variable `$sumGD` (in Zeile 8) der Schlusspreis von jedem Tag aufsummiert und dieser nach dem Ende der Schleife durch `count` geteilt. Die Berechnung des GD ist damit komplett abgeschlossen.

Ab Zeile 13 erfolgt die Berechnung von c_x für den Indikator *Rate of Change* (ROC). Für c_x wurde der 25. Tag als Stichtag festgelegt. c_x wird dann gesetzt, wenn `$count == $cxDays`, also = 25 ist. Falls es noch nicht so viele Rows gibt, wird c_x auf den Wert der letzten Row gesetzt (`$count == $context->groupSize()` gibt die aktuelle Anzahl Rows aus die es gibt).

Zum Schluss erfolgt noch die Berechnung des RSI. In der ersten if-Bedingung (Zeile 26) wird die erste Row, die in der Schleife aufgerufen wird, in der Variable `$rsiLatestTrade` gespeichert, um im nächsten Durchlauf auf diese Zugriff zu haben. Ab dem zweiten Durchlauf greift dann die zweite if-Bedingung in Zeile 30. Hier wird der Schlusspreis der zuvor gespeicherten Row sowie der aktuellen ausgelesen und der Rückgang bzw. die Zunahme zum Vortag berechnet und in der entsprechenden Variable gespeichert. Für die korrekte Berechnung von h bzw. n wird außerdem die Anzahl festgehalten, wie oft auf die beiden Werte jeweils aufsummiert wurde. Anschließend wird `$rsiLatestTrade` die Row des aktuellen Durchlaufs zugewiesen.

Nach dem die Schleife alle Durchläufe beendet hat, werden die (Teil-)Ergebnisse, in einer neuen Row gesammelt und über einen Funktionsaufruf zum zweiten Table geschickt.

Die abschließenden Berechnungen für den RSI und ROC sehen folgendermaßen aus:

```
1 #Berechnung des RSI:
2 my $hLength = $rLast->get("hLength");
3 my $nLength = $rLast->get("nLength");
4 if ($hLength > 0 && $nLength > 0) {
5   $h= ($rLast->get("h"))/$hLength;
6   $n= ($rLast->get("n"))/$nLength;
7   $rsi= ($h/($h+$n))*100
8 } elsif ($hLength > 0) {
9   $rsi= 100;
```

```
10 }
11
12
13 #ROC
14 if($rLast->get("cx") > 0) {
15   $roc = $rLast->get("c") / $rLast->get("cx")*100;
16 }
```

Aus der Ergebnis Row des ersten Tables, werden die Längen ausgelesen, also wie oft auf h bzw. n aufsummiert wurde. Dann müssen drei Fälle unterschieden werden.

Erstens: Der Standardfall, es fand eine Aufsummierung sowohl auf h als auch auf n statt; `$nLength` und `$hLength` sind damit beide größer 0. Dann wird die Berechnung durchgeführt, wie im Abschnitt 4.2.3 Beispiel 10 gezeigt.

Zweitens: Wenn es nur Kurssteigerungen gab und `$nLength` demnach Null ist, würde das Ergebnis der Berechnung immer 100 sein. Die Zahl wird deshalb direkt ohne eine Berechnung zugewiesen.

Drittens: Der dritte Fall wäre, dass es nur Kursrückgänge gab und `$hLength` entsprechend Null ist. In solch einem Fall ist das Ergebnis der Berechnung ebenfalls immer Null. Da die `$rsi`-Variable schon bei der Initialisierung auf Null gesetzt wird, braucht dieser Fall nicht in Form eines else-Statements berücksichtigt werden.

Zum Schluss erfolgt die Berechnung des ROC (Zeile 14). Dazu wird der aktuelle Kurs durch den vor x-Tagen geteilt und das Ergebnis mit 100 multipliziert.

Diese und die restlichen Indikatoren werden wieder in einer Row gesammelt und über einen Aufruf zum GUI-Package für eine letzte Auswertung und Darstellung in Liniendiagrammen geschickt. Indikatoren wie der GD, die bereits im ersten Table vollständig berechnet wurden, werden einfach aus der aktuellen Row ausgelesen und in die neue kopiert.

Natürlich wäre auch eine Aufteilung der Berechnungen auf mehr als zwei Tables möglich gewesen. Dies hätte aber jedesmal eine erneute Iteration über alle Rows umfasst, die so eingespart wurde.

Die Verarbeitung der aktuellen Ereignisse gestaltet sich einfacher. Innerhalb der Aggregationsmethode, wird anhand des neuen Ereignisses und dem davor geprüft ob sich der aktuelle Kurs einer Aktie geändert hat. Nur wenn dies der Fall ist, wird das Ereignis weiter zum GUI-Package geleitet. Die Methode stellt damit einen Filter dar, um Ereignisse die keine Änderungen im Aktienpreis aufweisen herauszufiltern.

4.4.4 GUI (package GUI.pm und checkEvents.pm)

Das GUI-Package ist für die grafische Darstellung des Programms verantwortlich. Da es die neu gewonnen Ereignisse des processing-Package empfängt, stellt es eine Ereignissenke dar.

Von den Rows mit den aktuellen Daten werden im GUI-Package der aktuelle Kurs und der Unternehmensname ausgelesen und in einem Array gespeichert. Die Daten werden anschließend zur Darstellung in einem Liniendiagramm, das den aktuellen Kursverlauf angezeigt, verwendet.

Die Ergebnisse der historischen Daten werden in einem Table gespeichert. Der Aufbau des Tables ist identisch zu dem bereits gezeigten im Abschnitt 5.

Die letzte Prüfung der Indikatoren wird im checkEvents-Package durchgeführt. Ergeben sich dabei relevante Informationen, bekommt der Nutzer in einem Textfeld auf der Hauptseite das Programms eine Nachricht ausgegeben. Es wäre natürlich auch möglich, bei signifikanten Änderungen, Geschäftsprozesse anzustoßen, den Nutzer z. B. per E-Mail oder über ein sich öffnendes Fenster darüber zu informieren. Da die Software aber historische Daten von einem Zeitraum größer als ein Jahr auswertet und entsprechend viele signifikante Ereignisse auftreten, wurde für die Darstellung der Ergebnisse ein Textfeld gewählt, in dem die Ereignisse einfach nachgelesen werden können.

Die Regeln für die Interpretation der Indikatoren im checkEvents-Package, sind den Quellen [2] und [7] entnommen. Da unterschiedliche Unternehmen unterschiedlicher Regeln bedürfen, ist es sinnvoll, diese anzupassen zu können. Dafür besteht im Programm die Möglichkeit, über eine Eingabemaske die Zahlen zur Auswertung der Indikatoren anzupassen. So gibt, für vorsichtige Anleger, das Programm schon bei kleinen Änderungen Meldungen aus, für Zocker, die dagegen auf hohe Rendite setzen, erst bei größeren Schwankungen.

Anpassungen der Regeln, können innerhalb des Programms über den Reiter Einstellungen vorgenommen werden. Dabei können Änderungen für den DMI, ADX, ROC und den RSI vorgenommen werden. Für den DMI und ADX, können die Anzahl der Punkte festgelegt werden, die der jeweilige Indikator fallen oder steigen muss, damit eine Nachricht ausgegeben wird.

Für den ROC können die untere und obere Grenze für die in Abschnitt 4.2.3 erwähnte Neutrale Zone eingestellt werden.

Auch für den RSI kann der Signalbereich geändert werden. Genauer, die Werte die ein über-, bzw. überverkaufen anzeigen. Befindet sich der RSI bereits in einem der beiden Bereiche (wenn der Wert beispielsweise <30 oder >70 ist), und fällt oder steigt weiterhin, wird ebenfalls eine Nachricht ausgegeben, dass der Wert noch weiter in den jeweiligen Bereich gefallen oder gerutscht ist. In den Einstellungen kann auch die Schrittweite festgelegt werden, ab der dafür eine zusätzliche Nachricht ausgegeben werden soll.

Die Einstellungen gelten dabei immer für alle Unternehmen und gehen beim Beenden des Programmes verloren. Zum Programmstart sind feste Werte vorgegeben.

Das checkEvents-Package beginnt mit der Auswertung der einzelnen Börsenindikatoren. Die Auswertung geschieht auf Basis des aktuellen Ereignisses, das als Argument übergeben worden ist und auch dem, das davor ausgewertet wurde. Für diesen Vergleich wird das aktuelle Ereignis am Ende der Auswertung im Package gespeichert. Die Regeln sind mit if-else-Abfragen umgesetzt worden und haben folgende Form:

```
1 if ($rocOld > 108 && $rocCurrent < 108) {
2   &{$textRef}("Der Roc hat den neutralen Bereich von oben aus betreten.
3   -> m"ogliches  Verkaufssignal.", "black")
4 }
```

Das Statement prüft ob der vorige ROC >108 und der aktuelle ROC <108 ist, also in die neutrale Zone gefallen ist. Ist dies der Fall, wird eine Referenz auf das Textfeld genutzt

und diesem eine Nachricht hinzugefügt. Das zweite Argument des Funktionsaufrufes bestimmt die Farbe des Textes (im Beispiel schwarz). Wichtige Ereignisse können so hervor gehoben werden.

Diese Art der Auswertung stellt schon eine hohe Abstraktion von den ursprünglichen Daten dar. Aus den einzelnen Börsenwerte wurden aussagekräftige Indikatoren gewonnen, die zur Entscheidung beim Aktienhandel genutzt werden können. Tatsächlich kann die Abstraktion aber noch weiter verbessert werden, indem nicht nur die Indikatoren einfach ausgewertet, sondern miteinander in Verbindung gesetzt (korreliert) werden.

Auch dies geschieht im checkEvents-Package. Wenn eine der if-Bedingungen während der Auswertung der einzelnen Indikatoren einen Treffer ergibt, wird dies mit `int`-Werten in verschiedenen Variablen vermerkt. Für die abschließende Korrelation der Daten müssen dann nicht große und komplexe if-Statements erstellt, sondern nur geprüft werden, ob alle nötigen Regeln (also die einzelnen if-Bedingungen) „gematcht“ haben.

Die Regeln haben folgende Form:

```
1 if ($rocCheck == 1 && $cciCheck == 1) {
2   &{$textRef}("ROC und CCI haben beide angeschlagen.
3   Aktien sollten gekauft werden.", "blue")
4 }
```

Das Prüfen der Variablen `$rocCheck` und `$cciCheck` auf den Wert 1 (0 für Regel ist nicht eingetroffen, 1 für Regel ist eingetroffen), ersetzt dabei eine if-Abfrage, die die nachfolgenden vier Abfragen hätte vereinen müssen:

```
1 if($cciOld < -100 && $cciCurrent > -100) {....}
2
3 if($cciOld < 0 && $cciCurrent > 0) {....}
4
5 if ($rocOld < 100 && $rocCurrent > 100) {....}
6
7 if ($rocOld < 90 && $rocCurrent >= 90) {....}
```

Die beiden CCI- und ROC-Abfragen geben, wenn Sie eintreffen, einen Hinweis darauf, dass aktuell ein Aktienkauf sinnvoll sein kann. Im Regelfall trifft immer nur eine der beiden Regeln für den CCI und ROC zu und setzt den Wert der Variablen entsprechend auf 1. Die komplexeren Regeln verbinden auf diese Weise die (Teil-)Ergebnisse der einzelnen Indikatoren. Daher gibt es auch keine Möglichkeit, in den Einstellungen die Auswertung dieser Regeln direkt zu beeinflussen. Dies kann nur über die Einstellungen der einzelnen Indikatoren selbst geschehen.

Das Auswerten der Börsenindikatoren wird über die in Abbildung 6 dargestellte Combobox angestoßen. In dieser steht der Name jedes Unternehmens, für das Daten in der Table vorliegen. Nachdem ein Unternehmen ausgewählt wurde, wird die Auswertung anhand der in der Table gespeicherte Ereignisse jedesmal aufs Neue durchgeführt, und das Ergebnis im Textfeld angezeigt.

Neben der Auswertung im checkevents-Package werden die Indikatoren auch als Liniendiagramme im Programm darstellt. Dafür gibt es verschiedene Reiter, innerhalb derer die unterschiedliche Indikatoren in grafischer Form dargestellt werden (siehe Abbildung

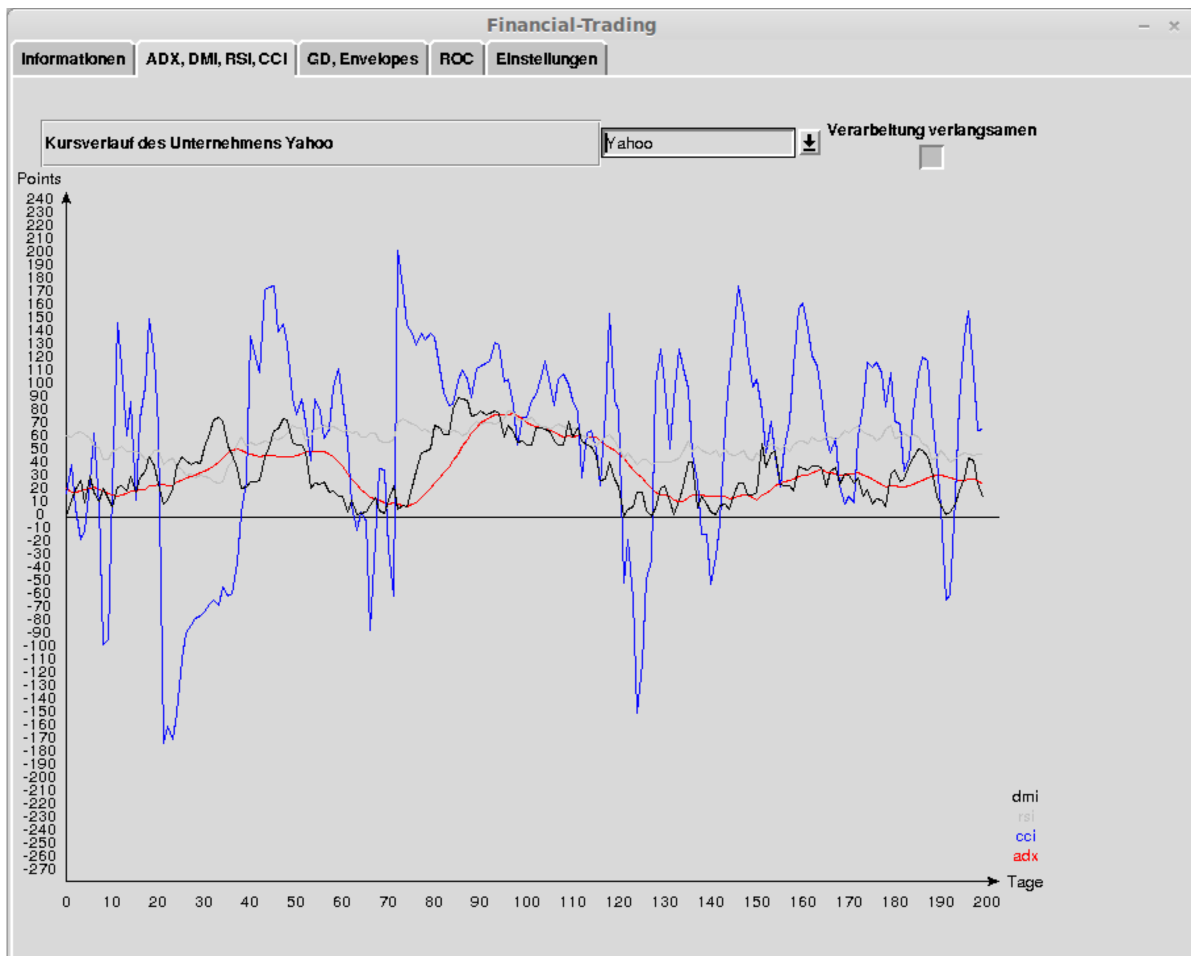

Abbildung 6: Visualisierung der Indikatoren in Diagrammform

6). Die Darstellung umfasst dabei alle in der Table gespeicherten Rows für das ausgewählte Unternehmen und gibt somit den Verlauf des jeweiligen Indikators über 200 Tage wieder.

5 Fazit

Die hier vorgestellte Implementierung gibt nur einen kleinen Ausschnitt der Realität wieder. In einem großen System werden Daten aus unterschiedlichsten Quellen verarbeitet und z. B. durch Informationen aus Datenbanksystemen ergänzt. Wie der Abschnitt 2 schon vermuten lies, bietet sich eine Verteilung der einzelnen EPAs zur Performanzsteigerung an. Dies erleichtert zugleich auch die Implementierung bzw. Erweiterung der Software um neue Ereignisregeln. Eine Erweiterung der Software durch neue Regeln kann durch die Implementierung eines neuen EPAs mit den gewünschten Regeln erfolgen. Dieser muss nach Fertigstellung nur noch in das EPN eingebunden werden. Auf diese Weise können in großen Unternehmen problemlos 100 000 Ereignisse und mehr pro Sekunde verarbeitet werden.

Wann ist der Einsatz von Triceps sinnvoll? Im Gegensatz zu Esper ist ein Einsatz in größeren Softwareprojekten nicht zu empfehlen. Der Hauptgrund dafür liegt in der Entwicklung der Engine durch nur eine Person und den dadurch nicht vorhandenen Support, wie man ihn z. B. bei Esper bekommen kann. Dies schlägt sich auch negativ auf die Updaterate von Triceps durch. Es gibt zwar regelmäßig neue Snapshots der Engine zum Downloaden, diese sind aber für einen produktiven Einsatz aufgrund vorhandener Fehler nicht zu gebrauchen. Zudem ist fraglich, wie lange die Entwicklung und Verbesserung von Triceps noch durch S. Babkin vorangetrieben werden wird. Es bestünde die Möglichkeit, das er die Entwicklung von heute auf morgen stilllegt, was bei einer Engine, die in einem großen Softwaresystem arbeiten soll, nicht akzeptabel ist.

Fairerweise sei gesagt, das Triceps auch nicht für einen solchen Einsatz entwickelt worden ist. Das Ziel ist eher, eine Möglichkeit zu schaffen, auf einfache und schnelle Weise Ideen, die das Complex-Event-Processing betreffen, austesten zu können. Dies wird durch die Ausführung der Triceps-Bibliotheken in Perl unterstützt, da die Sprache sehr flexibel ist und auch bei Fehlern toleranter als z. B. die Sprache C ist. Darüber hinaus ist Triceps für alle interessant, die sich noch nicht mit CEP beschäftigt haben, da der Developer-Guide neben dem Gebrauch der Engine auch die Konzepte des CEPs detailliert erläutert, sich mit den daraus resultierenden Problemen beschäftigt und Lösungen für diese vorschlägt. Durch den quelloffenen und umfangreich kommentierten Sourcecode wird dieser Gedanke weiter unterstützt.

Wie die Implementierung der Software zeigt, ist aber auch ein Einsatz in kleineren Programmen durchaus möglich. Die Version 1.0 von Triceps lief während der Entwicklung stabil und fehlerfrei, ebenso problemlos verlief das Kompilieren und Installieren. Durch die anstehende Veröffentlichung von Triceps 2.0, wird sich die Engine für einen solchen Einsatz auch weiter empfehlen.

Literatur

[1] *SAP Sybase Event Stream Processor.* http://www.sybase.de/products/financialservicessolutions/complex-event-processing. Version: 03. Juli 2013

[2] *Lexikon der Chartformationen.* http://www.chartundrat.de/lexikon.php#indikatoren. Version: 14. Juni 2013

[3] *Yahoo! Finance APIs.* http://code.google.com/p/yahoo-finance-managed/wiki/YahooFinanceAPIs. Version: 14. Juni 2013

[4] *Perl 5 / Perl 6 Eine Einführung mit Anwendungen der Perl 5-Modulen CGI, DBI und Modulen für XML.* RRZN, 2009

[5] Babkin, S. : *Sergey Babkin on CEP.* http://babkin-cep.blogspot.de/. Version: 30. Juni 2013

[6] Babkin, S. : *Triceps an innovative CEP (Complex Event Processing) engine.* http://triceps.sourceforge.net/. Version: 30. Juni 2013

[7] Florek, E. : *Neue Trading Dimensionen.* FinanzBuch Verlag GmbH München, 2000

[8] Foundation, C. : *EsperTech Event Stream Intelligence: Esper and NEsper.* http://esper.codehaus.org/. Version: 07. August 2013

[9] Ralf Bruns, J. D.: *Event-Driven Architecture.* Springer-Verlag Berlin Heidelberg, 2010

6 Anhang

6.1 Minimalbeispiel

Nachfolgend ist die Umsetzung des Minimalbeispiels, mit Esper und Triceps, zu finden.

Esper

```
1
2 public class SuperType {
3   private int id;
4   private String date;
5   private String source;
6
7   public SuperType(...) {
8     this.id = id;
9     this.date = date;
10    this.source = source;
11  }
12
13  public int getId() {
14    return id;  }
15
16  public String getDate() {
17    return date; }
18
19  public String getSource() {
20    return  source; }
21 }
22
23 public class StockEvent extends SuperType {
24   private String name;
25   private float price;
26
27   public SuperType(int id, String date, String source
28               String name, float price) {
29   super(id, date, source);
30   this.name = name;
31   this.price = price;
32   }
33
34   public String getName() {
35   return name; }
36   public float getPrice() {
37   return price; }
38 }
39
40
41 EPServiceProvider epService = EPServiceProviderManager.getDefaultProvider();
42 EPAdministrator admin = epService.getEPAdministrator();
```

```
43
44 EPStatement selectStmt = admin.createEPL(
45 select *
46 from StockEvent.win:length(10)
47 where (name = 'Google' and price \textgreater = 600)
48 );
49
50 EPStatement selectStmt = admin.createEPL(
51 select symbol, avg(price)
52 from StockEvent.win:length(10)
53 group by symbol
54 );
55
56 StockEvent event = new StockEvent(  1, "09-12-2013",
57         "yahoo-finance", "Google", 738.13 );
58
59 epService.getEPRuntime().sendEvent(event);
```

Triceps

```
1
2 $rtSuper = Triceps::RowType->new (
3   id     => "int64",
4   date   => "string",
5   source  => "string",
6 );
7
8 $rtStockEvent = Triceps::RowType->new(
9   $rtSuper->getdef(),
10  name  => "string",
11  price => "float64",
12 );
13
14
15 $TradeUnit = Triceps::Unit->new("TradeUnit");
16
17 $tableType = Triceps::TableType->new($rtStockEvent)
18 ->addSubIndex("bySymbol", Triceps::IndexType->newHashed(key["symbol"])
19     ->addSubIndex("window", Triceps::IndexType->newFifo(limit => 100)
20       ->setAggregator(Triceps::AggregatorType->new(
21         $rtAvgPrice, "aggrAvgPrice", \&computeAverage)
22       )
23     )
24   )
25 );
26
27 $table = $TradeUnit->makeTable($tableType, "table")
28
29 $label = $tradeUnit->makeLabel($rowType, "name", \&selectSub, @args);
```

```
30 $label->chain(table->getInputLabel());
31
32 sub selectSub {
33   $row = $_[1];
34   if ($row->get("name") eq 'Google' && $row->get("price") >= 600) {
35     ....
36   }
37 }
38
39 sub computeAverage {
40   my $avg = 0;
41   my count = 0;
42   for (my $rhi = $context->begin(); !$rhi->isNull();
43           $rhi = $context->next($rhi)) {
44
45     $avg = $avg + $rhi->getRow()->get("price"),
46     count++;
47   }
48
49   $avg = $avg/$count;
50   ....
51 }
52
53
54 $row = $rtStockEvent->makeRowArray(
55   1, "09-12-2013", "yahoo-finance", "Google", 738.13
56 );
57
58 $table->insert($row);
```

6.2 Financial Trading-Software

Dieser Bachelorarbeit liegt die zuvor beschriebene Financial Trading-Software auf CD bei. Auf dieser ist ebenfalls die verwendete CEP-Engine Triceps und eine kurze Anleitung in PDF-Form enthalten.

Die Software wurde unter der Perl-Version 5.14.2 und dem Triceps Release 1.0.1 entwickelt und getestet. Lt. Developer-Guide sollte für eine korrekte Ausführung von Triceps, Perl mindestens in der Version 5.10.0 vorliegen. Triceps erfordert für die Ausführung ein 64-Bit Linux Betriebssystem. Zur der Entwicklung wurden Linux Mint 14 & 15 (jew. 64Bit Version) genutzt.

Die Software liegt sowohl in einer fertig kompilierten Version vor, als auch in Form der erstellten Perl-Skripte bzw. Packages. Bei der kompilierten Version muss nur noch das Ausführungsrecht auf executeable gesetzt werden. Anschließend kann das Programm ohne eine weitere Installation von Paketen oder Triceps selbst ausgeführt werden.

Soll, dagegen das Perl-Skript ausgeführt werden, müssen weitere zusätzliche Pakte und Triceps selbst installiert werden. Für die Installation von Triceps, sei an dieser Stelle auf den Developer Guide (Seiten 11+12) verwiesen.

Neben der Perl Standardinstallation und Triceps wurden eine Reihe weiterer Pakete über CPAN installiert. Zur Korrekten Ausführung des Skriptes, müssen diese nachinstalliert werden. Die zusätzlich genutzten Paketen sind:

- Tk,
- Tk::BrowseEntry,
- Tk::Checkbox,
- Tk::NoteBook,
- threads,
- Thread::Queue,
- List::Util,
- Math::Complex,
- Math::Round
- LWP::UserAgent,
- HTTP::Request::Common,

Die einfachste Möglichkeit die erforderlichen Pakete zu installieren ist über das Programm CPAN, welches i. d. R. unter Linux standardmäßig installiert ist. Dieses kann über die Konsole mit dem gleich lautenden Befehl „cpan" aufgerufen werden. Über den Befehl „install + Paketname" können anschließend die einzelnen Pakete nacheinander

automatisch installiert werden. Zur korrekten Ausführung von CPAN muss sich der Nutzer vorher über die Konsole als Root identifizieren, da ansonsten die Installation abgebrochen wird.

6.3 Perl-Guide

Dieser kleine Guide, soll einen schnellen Überblick über die Sprache Perl bieten, er behandelt dabei alle Perl spezifischen Elemente die zum Verständnis der Financial Trading-Software notwendig sind.

Datenstrukturen

In Perl werden drei unterschiedliche Datenstrukturen unterschieden,

- Skalar($),
- Array(@) und
- Hash(%).

Die Auswahl der Struktur erfolgt über das Zeichen, welches sowohl bei der Deklaration als auch bei der Nutzung der Variable immer vor dem Namen angegeben werden muss.

Es gibt keine Unterscheidung von Typen wie Integer, Float oder String, usw. Jede der drei Datenstrukturen kann jeden entsprechenden Typ beinhalten. Wie der Inhalt einer Variable interpretiert wird, hängt allein vom Kontext ab.

Beispiel:

```
1 $variable = "1";
```

Bei einem String Vergleich würde Perl die eins wie angegeben als String nutzen. Wird ein Vergleich der folgenden Art durchgeführt:

```
1 if ($variable==1) {...}
```

würde Perl die 1 automatisch von einem String in den benötigten Integer konvertieren. Ein Programmfehler wird nur erzeugt, wenn eine Konvertierung nicht möglich ist und der Vergleich deswegen fehlschlägt.

Skalar: "Grundlegend, sind alle Daten in Perl Skalare. Ein Skalar beinhaltet immer nur ein Element, z. B. eine Zahl oder ein String. Auch Arrays und Hashes setzten sich aus einzelnen Skalaren zusammen und stellen somit eine Gruppierung von Skalaren dar."[4]

Arrays: Es gibt mehrere Arten ein Array zu füllen und auszulesen, die nachfolgend kurz gezeigt werden sollen:

```
1 @array = (3, 4);
2 push(@array, 1);
3 unshift(@array, 2);
4
5 pop(@array);
6 shift(@array);
```

Die Befehle push() und unshift() beinhalten zwei Parameter, der erste ist das Array zu dem ein neues Element hinzugefügt werden soll, der zweite der Wert selbst. Der Unterschied zwischen beiden ist das push das Element am Ende des Arrays hinzufügt und unshift am Anfang des Arrays. Die Befehle pop und shift lesen jew. ein Element aus dem Array welches als Parameter angegeben werden muss aus. Pop liest dabei das letzte Element aus und gibt es zurück und shift das erste.

Alternativ können einzelne Elemente, wie auch in Java, über ihren Index ausgelesen werden. Dazu muss das Array im Skalaren Kontext (mittels $) und der Angabe des Indexes angesprochen werden.

```
1 $array[index]
```

Wird ein Array im Skalaren Kontext verwendet, also so:

```
1 $length = @array;
```

beinhaltet die Variable $length anschließen die Länge des Arrays.

Hashes: Die Hashes werden in Perl gewohnt über ein Key/Value-Paar gebildet.

```
1 my %hashRows;
2 $hashRows{$row->get("name")} = $row;
```

In dem Beispiel wird ein Hash mit dem Namen hashRows angelegt. In Zeile zwei wird dem Hash anschließend eine Row hinzugefügt. Als Schlüssels dient der Name der in der Row gespeichert ist und der Wert ist die Row selbst.

Eine Besonderheit die noch gezeigt werden soll, ist ein Hash der als Wert ein Array aufweist, das wiederum selbst über mehrere Einträge verfügt.

```
1 my %hashCurrent;
2 push (@{$hashCurrent{$row->get("name")}}, $row->get("currentPrice"));
3
4 @{$hashCurrent{$name}}[index];
5
6 shift (@{$hashCurrent{$name}});
```

Der Hash wird wie gezeigt angelegt. Um diesem nun einen neuen Wert samt Array zuzuweisen wird der bereits gezeigt push-Befehl verwendet. Als Schlüssel dient wieder der Name der Row und als Wert diesmal der aktuelle Preis. Ist noch kein Eintrag im Hash unter dem Namen vorhanden, wird ein neuer erstellt. Der Befehl push sorgt dafür das als Wert ein Array angelegt und der übergebene Wert in diesem gespeichert wird.

Zeile 4 zeigt wie anschließend auf das Array zugegriffen wird. Als Schlüssel wird der Name genutzt gefolgt vom index den man aus dem Array auslesen möchte. Neben dem gezeigten push-Befehl (in Zeile 6) sind auch shift, pop und unshift möglich.

Methoden

Eine Methode wird in Perl Subroutine genannt.

```
1 sub test {
2 my ($a, $b) = @_;
3
4 my $a = $_[0];
5 my $b = $_[1];
6
7 my $a = shift;
8 my $b = shift;
9
10 }
11 test(1,2);
```

Wie zu erkennen ist wird eine Methode über das Schlüsselwort sub gefolgt vom Namen eingeleitet. Argumente müssen nicht, wie z. B. bei Java in Rundenklammern in dem Methodenkopf angegeben werden.

Der Methodenaufruf erfolgt wie gewohnt, inklusive der Parameterübergabe (siehe Zeile 11). Das Beispiel zeigt außerdem drei Möglichkeiten auf, die zwei übergebenen Argumente auszulesen.

In der ersten werden innerhalb der Klammern alle Variablen deklariert, denen ein Argument zugewiesen werden soll. Über den @_-Aufruf werden den deklarierten Variablen dann nacheinander die Argumente in der übergebenen Reihenfolge zugewiesen.

Die zweite Möglichkeit ist, auf die Paramter mit der $_[ ] Notation zuzugreifen. Die Parameter könnten auch direkt in dieser Form genutzt werden, Konvention ist aber die einzelnen Parameter wie im Beispiel Variablen zuzuweisen, da dies die Lesbarkeit des Programmcodes erhöht.

Die letzte Möglichkeit ist die Parameter mit shift auszulesen und Variablen zuzuweisen. Das erste shift liest dabei den ersten Übergabeparameter aus, das zweite den zweiten. usw...

my

Wie in dem obigen Beispiel zu sehen ist, wird jede Variabel bei der Deklaration mit dem my()-Befehl versehen. Dieser sorgt dafür, das auf die Variable immer nur in dem Scope zugegriffen werden kann, in dem Sie deklariert worden ist. So wird die Variable vor (ungewollte) Änderungen durch andere Programmteile geschützt.

Packages

In Perl werden Programme durch Packages (auch Module genannt) strukturiert. Ein neues Package wird durch den package Befehl gefolgt vom Namen des Packages eingeleitet. Innerhalb des Packages, werden die gewünschten Funktionen über Subroutinen implementiert. Die eins am Ende signalisiert Perl beim einbinden des Packages, das dieses korrekt eingebunden ist.

```
1 package Hello;
2
3 sub printHello {
```

```
4     print "Hello $_[0]\n"
5 }
6
7 1;
```

Eingebunden, wird ein Package über den Befehel use + Packagenamen.

```
1 use Hello;
2
3 Hello::printHello();
```

Die Package-Methoden können anschließend, wie in Zeile 3 gezeigt, genutzt werden.

Referenzen

Nachfolgend soll noch kurz die Syntax für die Übergabe einer Referenz gezeigt werden. Diese ist für Methoden, Objekte und Variablen identisch.

```
1 initTextRef(\&insertText)
2
3 sub initTextRef {
4   $textRef = $_[0];
5 }
```

In der ersten Zeile wird der Methode initTextRef eine Referenz auf ein zuvor erstelltes Textfeld übergeben. Die Zuweisung zu einer Variable kann dann, wie bereits bei den Methoden gezeigt, erfolgen.